Advances in Industrial Control

Ying Bai
Zvi S. Roth

Classical and Modern Controls with Microcontrollers

Design, Implementation, and Applications



EXTRAS ONLINE

Springer

Advances in Industrial Control

Series editors

Michael J. Grimble, Department of Electronic and Electrical Engineering, University of Strathclyde, Glasgow, UK
Antonella Ferrara, Department of Electrical, Computer and Biomedical Engineering, University of Pavia, Pavia, Italy

Advances in Industrial Control is a series of monographs and contributed titles focusing on the applications of advanced and novel control methods within applied settings. This series has worldwide distribution to engineers, researchers and libraries.

The series promotes the exchange of information between academia and industry, to which end the books all demonstrate some theoretical aspect of an advanced or new control method and show how it can be applied either in a pilot plant or in some real industrial situation. The books are distinguished by the combination of the type of theory used and the type of application exemplified. Note that "industrial" here has a very broad interpretation; it applies not merely to the processes employed in industrial plants but to systems such as avionics and automotive brakes and drivetrain. This series complements the theoretical and more mathematical approach of Communications and Control Engineering.

Indexed by SCOPUS and Engineering Index.

Publishing Ethics: Researchers should conduct their research from research proposal to publication in line with best practices and codes of conduct of relevant professional bodies and/or national and international regulatory bodies. For more details on individual ethics matters please see:
https://www.springer.com/gp/authors-editors/journal-author/journal-author-helpdesk/publishing-ethics/14214

More information about this series at http://www.springer.com/series/1412

Ying Bai · Zvi S. Roth

Classical and Modern Controls with Microcontrollers

Design, Implementation, and Applications

Springer

Ying Bai
Department of Computer Science
and Engineering
Johnson C. Smith University
Charlotte, NC, USA

Zvi S. Roth
Department of Electrical
and Computer Science
Florida Atlantic University
Boca Raton, FL, USA

Additional material to this book can be downloaded from http://extras.springer.com.

ISSN 1430-9491 ISSN 2193-1577 (electronic)
Advances in Industrial Control
ISBN 978-3-030-01381-3 ISBN 978-3-030-01382-0 (eBook)
https://doi.org/10.1007/978-3-030-01382-0

Library of Congress Control Number: 2018958364

© Springer Nature Switzerland AG 2019
This work is subject to copyright. All rights are reserved by the Publisher, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, reuse of illustrations, recitation, broadcasting, reproduction on microfilms or in any other physical way, and transmission or information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar methodology now known or hereafter developed.
The use of general descriptive names, registered names, trademarks, service marks, etc. in this publication does not imply, even in the absence of a specific statement, that such names are exempt from the relevant protective laws and regulations and therefore free for general use.
The publisher, the authors and the editors are safe to assume that the advice and information in this book are believed to be true and accurate at the date of publication. Neither the publisher nor the authors or the editors give a warranty, express or implied, with respect to the material contained herein or for any errors or omissions that may have been made. The publisher remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

This Springer imprint is published by the registered company Springer Nature Switzerland AG
The registered company address is: Gewerbestrasse 11, 6330 Cham, Switzerland

This book is dedicated to my wife, Yan Wang, and to my daughter, Susan (Xue) Bai.

Dr. Ying Bai

I owe everything that I know in Control Systems to many teachers and colleagues that I have had over the years. I chose to dedicate this book to my first teachers from the Technion, Israel Institute of Technology, Prof. Julius Preminger and Prof. Eliezer Schoen, who introduced me to this field and are responsible for the first spark of interest that later became a career long passion.
Dr. Preminger later became my M.Sc. thesis advisor. I worked as a Teaching Assistant for Dr. Schoen many times and learned a lot from our weekly planning sessions. The lecture notes from that period still look fresh and relevant and I used some of this material in this book.
Dr. Schoen introduced all his students to dynamic simulations. At that time, in the early 1970s, the tool of choice was the text-based CSMP. These simulation tools became lifelong friends.

Dr. Zvi S. Roth

Series Editor's Foreword

The subject of control systems engineering is viewed very differently by researchers and those that practice the craft. The former group develops general algorithms with a strong underlying mathematical basis while for the latter concerns over the limits of equipment and plant downtime dominate. The series *Advances in Industrial Control* attempts to bridge this divide to some extent and hopes to encourage technology transfer.

The rapid development of new control theory and technology has an impact on all areas of control engineering and applications. There are new control theories, actuators, sensor systems, communication and computing methods, design philosophies, and, of course, new application areas. This provides some justification for a specialized monograph series but there is another more important reason. The development of relevant control theory needs to be stimulated and driven by the needs and challenges of applications. A focus on applications is also essential if the different aspects of the control design problem are to receive sufficient attention.

There is a lot of work on the analysis and synthesis problems in control systems engineering but much less on the problems of control design. The path from a control loop specification to a design, which satisfies requirements is often paved with uncertainties and confusion. The series provides an opportunity for researchers to present an extended exposition of new work on industrial control, raising awareness of the substantial benefits that can accrue, and the challenges that can arise, and dealing with this important issue of control-systems design.

The authors of this text have been working on the subject of control design and implementation over many years, and contributed to the literature in what is a rapidly changing environment. This book is concerned with the technology used when implementing control systems, which is a difficult topic to present when the technology and programming methods change every decade. However, there are some underlying principles that underpin current and probably future systems that the authors have covered. In fact, a good proportion of the material covers control fundamentals but often linked to the practical problems of implementation in digital devices.

The problems of sensor systems, actuators, and numerical implementation in microcomputing devices are often avoided in theoretical studies but this text covers the latter in great detail. At the same time, it provides an overview of some of the most useful tools in classical control theory and covers some topics in model-based and more advanced control. The MATLAB® scripts are useful for students and the Homework and Labs sections are valuable for instructors.

This text covers material that is often not considered in the huge volume of papers and books on control engineering. It is, therefore, a welcome addition to the series on *Advances in Industrial Control*.

Glasgow, UK
August 2018

Michael J. Grimble

Preface

Today, the most popular and powerful control technologies applied in our routine life are automatic and intelligent controls. Each individual living in our world, more or less, is under the effects or takes advantages of digital controls. Examples of such control techniques implemented in our society include automobiles, intelligent and graphic calculators, room cleaning robots, washers/dryers, HVAC systems, refrigerators, ATMs, iPads, iPhones, vending machines, and so on. Almost all aspects of our life are affected by either classical or modern control technologies.

In order to catch up, understand and master these control techniques, a complete and solid understanding and study process are necessary to all of us, of course, including college students. A good and practical textbook is prerequisite for this kind of study and learning process.

However, it is very difficult or even impossible to find a good or matched textbook for this objective from the current book market. Most control textbooks just discussed or covered popular control theories with some simulations. These kinds of textbooks cannot enable students to fully understand and apply what they learned to develop real control systems with only theory and simulations experience. A complete and detailed development chain is absolutely necessary to enhance students' learning and understanding abilities about classical and modern industrial control techniques and technology. This is the target objective of this textbook.

The authors of the book tried to provide a fully and completed package to cover both classical and modern control technologies with a real microcontroller system TM4C123GXL. The package includes not only solid theoretical introductions and simulations about traditional and modern control techniques, but also a set of implementation components, which contain the following:

(1) A powerful and latest microcontroller—TM4C123GH6PM (Texas Instruments),
(2) A Tiva™ TM4C123GXL LaunchPad™ Evaluation Board (EVB),
(3) An EduBASE ARM Trainer,
(4) A Keil® MDK μVersion-5 Integrated Development Environment (IDE)

With the help of these hardware and software as well as practical application notes with real examples, students can design, develop, and build some real and actual control systems by developing practical programming codes to control microcontrollers to perform real time controls to some motor systems. All example projects in the book have been compiled, built, and tested. To help students to master the main techniques and ideas, three appendices are also provided to facilitate the students to overcome some possible learning curves.

In addition to home works attached at each chapter, two sets of practical projects are also involved in this book, Class Projects and Lab Projects, to enable students to effectively learn and build some actual control program for real control systems. A complete set of home work solutions is provided for instructors. Moreover, a set of complete teaching materials built in MS PPT is also provided for instructors to facilitate their teachings.

You are welcome to have any question or comment for this book.

Charlotte, USA Dr. Ying Bai
Boca Raton, USA Dr. Zvi S. Roth

Acknowledgements

The first and most special thank to Dr. Ying Bai's wife, Yan Wang, and we could not finish this book without her sincerely encouragement and support.

Many thanks should be given to the Editors Oliver Jackson, Meertinus Faber and Subodh Kumar who made this book available to public. You could not find this book from the market without their deep perspective and hard working. The same thanks are extended to the editor team of this book. Without their contributions, it is impossible for this book to be published.

Finally but not the last, the thanks should be forwarded to all people who support us to finish this book.

Dr. Ying Bai

Over my 36 years of academic career at Florida Atlantic University I enjoyed tremendous amount of academic freedom, thanks to the many department chairs and Deans that I have encountered, which allowed me to compile over the years a large portfolio of undergraduate and graduate courses. This helped me a lot in my writing. Much of the contents came from my lecture notes in the courses of Control Systems 1, Control Systems 2, Nonlinear Systems, and Biosystems Modeling and Control.

I was fortunate to have many outstanding students. For many years, I have collected the best students solutions in each of my courses. Much of this material was very helpful for my case studies and for the homework problems with solutions that I selected for this book. Let me gratefully mention the students whose works I have used: Dr. Kasra Vakilinia, Dr. Ivan Bertaska, Claude Lieber MD, Benjamin Coleman, Dennis Estrada, and more.

Last but not least, many thanks to my loving wife Eva, who freed me up from many important chores, so that I could focus on finishing the book, and to my children who showed active interest and encouraged me to go on.

Dr. Zvi S. Roth

Contents

1 Introduction and Overview to This Book 1
1.1 The TM4C123GH6PM Microcontroller Development Tools and Kits 2
1.2 Outstanding Features About This Book 2
1.3 Who This Book Is For 3
1.4 What This Book Covered 4
1.5 How This Book Is Organized and How to Use This Book 6
1.6 How to Use the Source Code and Sample Projects 7
1.7 Instructors and Customers Supports 9

2 Overview of the Fundamentals of Control Systems 11
2.1 Introduction 11
2.2 The Open-Loop Control System 11
2.3 The Closed-Loop (Feedback) Control System 14
2.4 Introduction to Laplace Transformation 16
2.4.1 The Laplace Transform 18
2.5 The Transfer Functions and Block Diagrams 20
2.6 The Poles and Zeros 21
2.7 The Linear Time-Invariant Control Systems 24
2.7.1 The Time-Invariant System 24
2.7.2 The Linear System 25
2.8 The Nonlinear Time-Variant Control Systems 27
2.9 The Continuous Control Systems 29
2.10 The Discrete and Digital Control Systems 31
2.11 Summary 33

3 Introduction to Tiva C MCU LaunchPad™—TM4C123G 35
3.1 Introduction 35
3.2 Introduction to the Hardware Package 36

3.2.1 Overview TM4C123GH6PM Microcontroller Unit (MCU) . . . 37
3.2.2 Introduction to Tiva™ C Series LaunchPad™ TM4C123GXL EVB . . . 60
3.2.3 Introduction to EduBASE ARM® Trainer . . . 67
3.2.4 Introduction to Other Related Peripherals and Interfaces . . . 69
3.3 Introduction to the Software Package . . . 70
3.3.1 Overview ARM Microcontroller Development Kits . . . 70
3.3.2 Download and Install Development Suite and Specified Firmware . . . 73
3.3.3 Introduction to Integrated Development Environment—Keil® MDK . . . 73
3.3.4 Introduction to TivaWare™ for C Series Software Suite . . . 85
3.3.5 Tiva™ C Series LaunchPad™ Debug Adaptor and Debug Adaptor Driver . . . 90
3.4 Development Steps to Build a Basic Project with Keil® MDK μVersion5 IDE . . . 91
3.5 Summary . . . 98

4 System Mathematical Models and Model Identifications . . . 101
4.1 Introduction . . . 101
4.2 Mathematical Models for Most Popular Systems . . . 102
4.2.1 Models for Mechanical Rotational Systems . . . 103
4.2.2 Models for Electrical Circuit Systems . . . 103
4.2.3 Models for Electromechanical Systems . . . 104
4.3 Relationship Between Poles and Step Response . . . 106
4.3.1 Time-Domain Specifications for Step Responses . . . 108
4.4 Some Important Properties Related to System Models . . . 110
4.4.1 The Final Value Theorem . . . 110
4.4.2 Steady-State Errors . . . 111
4.4.3 Stability . . . 113
4.4.4 Linear Approximation of Physical Systems . . . 118
4.4.5 Some Important MATLAB Functions Used in System Models . . . 121
4.5 System Model Identification . . . 132
4.5.1 Identifying Models from Experimental Data . . . 133
4.6 Introduction to MATLAB® System Identification ToolBox™ . . . 134
4.6.1 Process Models . . . 135
4.6.2 System Identification Tool (GUI) . . . 137
4.6.3 Nonlinear ARX Models . . . 140

4.7 Case Study: Identification of a Real DC Motor Mitsumi 448 PPR Motor . . . 141
4.7.1 The Mitsumi 448 PPR Motor and Rotary Encoder . . . 141
4.7.2 QEI Modules and Registers in the TM4C123GH6PM MCU System . . . 145
4.7.3 PWM Modules in the TM4C123GH6PM MCU System . . . 153
4.7.4 Calibration of the Rotary Encoder to Identify the Feedback Path . . . 173
4.7.5 Identify the Dynamic Model for Our Target DC Motor . . . 181
4.8 Summary . . . 188

5 Classical Linear Control Systems—PID Control Systems . . . 195
5.1 Introduction to PID Control Systems . . . 195
5.2 Design PID Controllers with Root Locus Method . . . 199
5.2.1 Introduction to Root Locus Method . . . 199
5.2.2 Sketching Root Locus . . . 203
5.2.3 General Considerations of Designing Compensator with Root Locus . . . 209
5.2.4 Design of a Phase Lead Compensator with Root Locus Method . . . 211
5.2.5 Design of a Phase Lag Compensator with Root Locus Method . . . 216
5.2.6 Use Computer-Aided Methods to Design Compensators with Root Locus . . . 219
5.3 Design PID Controllers with Frequency Method . . . 228
5.3.1 The Major Properties of the Frequency Response Method . . . 231
5.3.2 General Considerations of Designing Compensator with Bode Plot . . . 246
5.3.3 Design of a Phase Lead Compensator with Bode Plot Method . . . 248
5.3.4 Design of a Phase Lag Compensator with Bode Plot Method . . . 251
5.3.5 Design of a PID Controller with Bode Plot Method . . . 253
5.3.6 Use Computer-Aided Methods to Design Compensators with Bode Plot . . . 255
5.3.7 The Nyquist Stability Criterion . . . 269
5.4 Design PID Controllers with State-Space Method . . . 274
5.4.1 State-Space Variable . . . 275
5.4.2 System Transfer Functions and Eigenvalues in State Space . . . 276

5.4.3 Block Diagram and State-Space Design 281
5.4.4 Control Law Design for Pole Placement 283
5.4.5 Estimator Design 287
5.4.6 Compensation—Combined Control and Estimation 290
5.4.7 Introduce the Reference Input 292
5.4.8 Integral Control 294
5.4.9 Controllability and Observability 295
5.4.10 Pole Placement Using State Feedback Control 297
5.5 Case Study: Design and Build a PID Controller for a DC Motor System 298
5.5.1 The Continuous and Discrete PID Closed-Loop Control System 298
5.5.2 Build the Control Software to Implement the PID Controller 301
5.6 Summary 304

6 Practical Nonlinear Control Systems 323
6.1 MATLAB Simulink Basics for Analysis of Nonlinear Systems 324
6.1.1 Simulink Model of a Pendulum 324
6.1.2 Simulink Model of a Master and Dog Line-of-Sight Tracking 329
6.1.3 Case Study: Aircraft Landing Modeling and Control 332
6.2 More Advanced Aspects of MATLAB Simulink 341
6.2.1 Multiple Simulink Runs of Two-Species Growth Models 341
6.2.2 Case Study: Parameter Identification in a Spread of Infectious Diseases and Determination of Immunization Control Strategy 350
6.3 Example of Control Systems with Simulator in the Loop: Glucose–Insulin Dynamics 361
6.3.1 Case Study: System Identification and Open-Loop Control of Diabetic Patient's Blood Glucose Level 365
6.4 Simple Relay-Based Control Systems 377
6.4.1 How Not to Do Relay Control 379
6.4.2 Relay Control with Position and Velocity Feedback—The Sliding Phenomenon 381
6.4.3 Minimum-Time Relay Control—Bang–Bang Control 394

6.5 Sliding Mode Control . . . 398
6.5.1 Tracking Control of First-Order Systems with Modeling Uncertainties . . . 399
6.5.2 SMC Control Going Beyond First-Order Systems . . . 409
6.6 Control Design Based on Synthesis of Lyapunov Functions . . . 420
6.6.1 Brief Review of Lyapunov Stability Theory . . . 420
6.6.2 Control Design Based on Synthesis of Lyapunov Functions . . . 425
6.7 Summary . . . 431
References . . . 435

7 Fuzzy Logic Control Systems . . . 437
7.1 Introduction to Fuzzy Logic Control Technology . . . 438
7.2 Fuzzy Sets . . . 440
7.2.1 Classical Sets and Operations . . . 440
7.2.2 Mapping of Classical Sets to Functions . . . 441
7.2.3 Fuzzy Sets and Operations . . . 443
7.2.4 The Comparison Between the Classical and Fuzzy Sets . . . 445
7.3 Fuzzifications and Membership Functions . . . 445
7.4 Fuzzy Control Rules . . . 447
7.4.1 Fuzzy Mapping Rules . . . 448
7.4.2 Fuzzy Implication Rules . . . 449
7.5 Defuzzifications and Lookup Table . . . 450
7.5.1 Mean of Maximum (MOM) Method . . . 450
7.5.2 Center-of-Gravity (COG) Method . . . 451
7.5.3 The Height Method (HM) . . . 451
7.5.4 The Lookup Table . . . 451
7.5.5 The Offline and Online Defuzzifications . . . 455
7.6 Architectures of Fuzzy Logic Control Systems . . . 455
7.6.1 Fuzzy Proportional–Derivative (PD) Controller . . . 457
7.6.2 Fuzzy Proportional Integrated (PI) Controller . . . 458
7.7 Introduction to MATLAB Fuzzy Logic Toolbox . . . 458
7.7.1 Fuzzy Inference Functions . . . 459
7.7.2 Fuzzy Logic Toolbox Graphical User Interface Tool (FLTGUIT) . . . 464
7.8 Implementation of Fuzzy Logic Controls with Fuzzy Logic Toolbox . . . 472
7.8.1 Applications of FLTGUIT for a DC Motor System . . . 472
7.8.2 Applications of Fuzzy Logic Controls with Fuzzy Functions . . . 479

7.9 Tuning of the Fuzzy Logic Controller 485
7.10 Implementation of Fuzzy Logic Controls with Simulink® 487
7.11 Comparison Between the PID Controller and the Fuzzy Logic Controller 490
7.11.1 Simulation Studies 491
7.11.2 Experimental Studies 501
7.12 Summary 506

8 Fuzzy Logic Controller Design in TM4C123G MCU System 513
8.1 Introduction 513
8.2 Implementation of Fuzzy Logic Closed-Loop Control Systems 514
8.2.1 The Fuzzification Process 514
8.2.2 Design of Control Rules 516
8.2.3 The Defuzzification Process 516
8.3 Apply the Fuzzy Logic Controller to the DC Motor Control System 519
8.3.1 Build the Fuzzy Logic Control Project Fuzzy Control 522
8.4 Case Study 527
8.4.1 Build the Fuzzy Logic Control Project FZControl 532
8.5 Apply Online Fuzzy Inference System to Obtain the Real-Time Outputs 537
8.5.1 Build a Visual C++ Interface Project to Access MATLAB FIS 538
8.5.2 Build a Complete Project to Get Online Fuzzy Output 543
8.6 Summary 546

9 Interval Type-2 Fuzzy Logic Controllers 549
9.1 Introduction to Type-1 and Interval Type-2 Fuzzy Sets 549
9.2 Interval Type-2 Fuzzy Sets Operations 551
9.3 Interval Type-2 Fuzzy Logic System 552
9.4 Karnik–Mendel (KM) Algorithms 554
9.5 An Example of an Interval Type-2 Fuzzy Logic System 557
9.6 Design and Build an IT2 Fuzzy Logic Controller for a DC Motor System 559
9.6.1 Build an IT2 Fuzzy Logic Control System for a DC Motor System 559
9.7 Simulation Study for This IT2 Fuzzy Logic Control System 564
9.7.1 Create a Simulation Block Diagram Model for FIS—IT2FL_LTS 565
9.7.2 Simulation Results for IT2FL_LTS Model 567

9.8 Simulation Study for Comparison Among PID, Type-1 and IT2 FLC . . . 568
9.8.1 Simulation Study Comparison for Systems with Normal Time Delays . . . 568
9.8.2 Simulation Study Comparison for Systems with Long Time Delays . . . 570
9.8.3 Simulation Study Comparison for Systems with Random Noises . . . 572
9.8.4 Simulation Study Comparison for Systems with Model Uncertainty . . . 574
9.9 Summary . . . 575
References . . . 579

Appendix A: Download and Install Keil MDK-ARM 5.24a IDE . . . 581

Appendix B: Download and Install TivaWare SW-EK-TM4C123GXL Package . . . 587

Appendix C: Download and Install Stellaris ICDI and Virtual COM Port . . . 593

Index . . . 603

About the Authors

Dr. Ying Bai is a Professor in the Department of Computer Science and Engineering at Johnson C. Smith University. His special interests include: Intelligent controls, soft-computing, mix-language programming, fuzzy logic controls, robotic controls, robots calibrations, and fuzzy multi-criteria decision-making.

His industry experience includes positions as software and senior software engineers at companies such as Motorola MMS, Schlumberger ATE Technology, Immix TeleCom, and Lam Research.

During recent years, Dr. Bai has published more than 50 academic research papers in IEEE Trans. Journals and International conferences. He also published fourteen (14) books with publishers such as Prentice Hall, CRC Press LLC, Springer, Cambridge University Press, and Wiley IEEE Press in recent years. The Russian translation of his first book entitled *Applications Interface Programming Using Multiple Languages* was published by Prentice Hall in 2005. The Chinese translation of his eighth book entitled *Practical Database Programming with Visual C#.NET* was published by Tsinghua University Press in China at the end of 2011. Most books are about interfacing software programming, serial port programming, database programming, and fuzzy logic controls in industrial applications as well as microcontroller programming and applications.

Dr. Zvi S. Roth received his B.Sc. and M.Sc. both in Electrical Engineering from the Technion Israel Institute of Technology in 1974 and 1979, respectively, and his Ph.D. in Systems Engineering from Case Western Reserve University in 1983. He thereafter joined the faculty of the Florida Atlantic University (FAU) College of Engineering and Computer Science, where he is presently a Professor in the Department of Computer and Electrical Engineering and Computer Science.

Drs. Benjamin Mooring, Zvi S. Roth, and Morris Driels wrote the book *Fundamentals of Manipulator Calibration* (John Wiley 1991).

Dr. Roth served as Director of the State of Florida funded FAU Robotics Center from 1985 to 1993. Between 1985 and 1998, the FAU Robotics Center graduated over 30 Ph.D. students. In 1996, Drs. Hanqi Zhuang and Roth wrote the book *Camera-Aided Robot Calibration* (CRC Press).

Dr. Roth served as Chair of the EE Department from 1993 to 1997, and from 2005 to 2008 as Director of the State of Florida funded Florida-Israel Institute. Additional research interests that Dr. Roth has developed over the years include Automation Design for Biotechnology, Submicron Analog Electronic Circuits Design, and Modeling of Biological and Physiological Systems. He published over 50 refereed papers in journals and conferences and graduated 9 Ph.Ds.

Trade Marks and Copyrights

- Arm® is a trademark and a product of Arm Limited.
- Cortex® is a trademark and a product of Arm Limited.
- Keil® is a trademark and a product of Arm Limited.
- µVersion® is a trademark and a product of Arm Limited.
- Arm7™ is a trademark and a product of Arm Limited.
- Arm9™ is a trademark and a product of Arm Limited.
- ULINK™ is a trademark and a product of Arm Limited.
- MATLAB® is a trademark and product of The MathWorks®, Inc.
- Simulink® is a trademark and product of The MathWorks®, Inc.
- System Identification Toolbox™ is a trademark and product of The MathWorks®, Inc.
- Control System Toolbox™ is a trademark and product of The MathWorks®, Inc.
- Fuzzy Logic Toolbox™ is a trademark and product of The MathWorks®, Inc.
- Neural Network Toolbox™ is a trademark and product of The MathWorks®, Inc.
- Texas Instruments™ is a trade mark of the Texas Instruments Incorporated.
- Tiva™ is a trademark and product of the Texas Instruments Incorporated.
- TivaWare™ is a trademark and product of the Texas Instruments Incorporated.
- LaunchPad™ is a trademark and product of the Texas Instruments Incorporated.
- Code Composer Studio™ is a trademark and product of the Texas Instruments Incorporated.
- Stellaris® is a trademark and product of the Texas Instruments Incorporated.

Chapters 3 and 4 used the following copyrighted materials from Wiley IEEE Press:

- *Practical Microcontroller Engineering with ARM Technology*, 1st Ed., 2016, (Ying Bai). Reused materials include: Sects. 2.6.1–2.6.2, 2.6.3.3, 2.7 and 2.8 in Chap. 2. Sections 3.2–3.4 in Chap. 3, Sects. 7.6.1–7.6.3, 7.6.4.1–7.6.4.2, 7.6.5.1, 7.6.6.1–7.6.6.3, 7.6.7–7.6.8 in Chap. 7, Sects. 10.3–10.5 in Chap. 10.

All of these reused materials are reproduced with the permission of John Wiley & Sons, Inc., 2016.

Chapter 7 used the following copyrighted materials from Springer:

- *Advanced Fuzzy logic Technologies in Industrial Applications* (Book Title) Chapter 2: Fundamentals of Fuzzy Logic Control—Fuzzy Sets, Fuzzy Rules and Defuzzifications, 2006, pages: 17 ~ 35, Ying Bai, Hanqi Zhuang and Dali Wang, all of these reused materials are reproduced with permission of Springer, 2006.

Chapter 1
Introduction and Overview to This Book



Have been working and teaching in classical and modern controls with different control strategies and technologies for many years, more and more advanced control theories and techniques have been developed and reported in recent years. However, it always felt that there is a gap that existed between the control theories and practical control implementations, and this gap is not easy to be covered by most of the current control books, including textbooks, from the book markets.

Most of the control books followed a traditional writing style; first, the control theories and principles are introduced and presented. Then, some control problem-solving techniques are discussed, and finally, some simulations are provided to confirm the introduced control theories with some examples. That is the style for most control books deal with the classical and modern control technologies and implementations.

However, students need to get much more detailed and actual control knowledge as well as techniques for most popular and practical control systems, and furthermore to utilize the knowledge and techniques, they learned from classes to personally and practically design a complete control system with actual control objectives or targets to fully understand what they have learned. In this way, students can fully study and understand control theories and techniques better since only theory studies and system simulations are not enough to enable them to totally understand those control strategies.

Based on the above ideas, this book is planned and prepared. The objective of this book is to provide students a way or a bridge to cross over the gap mentioned above to enable them to combine control theories they learned from classes with popular practical control targets together to actually design, build, simulate, and finally develop real control programs to perform practical real-time closed-loop control strategies to real control targets to get optimal control systems.

For that purpose, a real and actual controller or a microcontroller system is needed as a tool to enable students to develop control algorithms and programs on it to realize the actual control functions for control targets or plants. To help students to better design and build those control programs, a set of related software tools is

© Springer Nature Switzerland AG 2019

Y. Bai and Z. S. Roth, *Classical and Modern Controls with Microcontrollers*,
Advances in Industrial Control, https://doi.org/10.1007/978-3-030-01382-0_1

also provided to enable them to effectively design and build those control algorithms and programs to perform real-time controls on plants.

Among the most popular microcontroller systems, a product or a microcontroller unit (MCU) produced by Texas Instruments™, TM4C123GH6PM, with its evaluation board, TM4C123GXL, is adopted in this book as the hardware. This MCU is a 32-bit microcontroller unit providing multiple control functions. The related software tools include an Integrated Development Environment (IDE), **Keil® ARM®-MDK μVersion®5.24a** and TivaWare™ SW-EK-TM4C123GXL Software Driver Package.

1.1 The TM4C123GH6PM Microcontroller Development Tools and Kits

In this book, we concentrate on a typical and popular ARM® Cortex®-M4 MCU evaluation system built by Texas Instruments™ called TM4C123GXL, in which two ARM® Cortex®-M4 MCUs, TM4C123GH6PM, are utilized. The related development tools and kits can be categorized into two parts: the hardware part and the software part.

The hardware part includes:

- Tiva™ LaunchPad TM4C123GXL Evaluation Board (EVB), TM4C123GXL.
- EduBASE ARM® Trainer (contains most popular peripherals and interfaces).
- Some other related peripherals, such as DC Motors, CAN interfaces, and D/A converters.

The software part includes:

- Integrated Development Environment Keil® ARM®-MDK μVersion®5 (IDE).
- TivaWare™ SW-EK-TM4C123GXL Software Driver Package.
- Stellaris In-Circuit Debug Interface (ICDI).

Appendices A–C in this book provide detailed information and directions for downloading and installing these software tools in your host computers.

1.2 Outstanding Features About This Book

1. Both classical and modern control theories and control techniques are introduced and presented with real examples, including PID controls, type-1 fuzzy logic controls and interval type-2 fuzzy logic controls.
2. Both linear and nonlinear control theories and techniques are introduced and discussed with various MATLAB® Control Toolboxes, including Nonlinear Control Toolbox, as well as related control functions and tools.

3. For all control strategies, classical and modern controls, linear and nonlinear controls, and related simulation studies are provided with MATLAB® Simulink® to provide readers a real and complete picture about the control performances of those introduced control techniques.
4. Both control theories and actual hands-on control program development procedure are provided step by step with detailed illustrations to enable students to practically build control algorithms and programs on a real microcontroller system, TM4C123GXL, to perform real-time control for DC motor closed-loop control systems. Furthermore, students can verify their control performance by plotting step responses for those closed-loop control systems.
5. More than 12 real class projects are provided in this book with detailed and line-by-line explanations and illustrations to enable readers to learn and understand both classical and modern control strategies easily and faster.
6. Totally 16 real lab projects are provided in this book to enable students to practically design and build various control strategies to make them understand better for both classical and modern control strategies more effectively.
7. A complete set of home works and lab projects is attached after each chapter. This enables students to understand what they learned better by doing something themselves.
8. A complete set of answers to all home works and labs is provided for the instructors.
9. A complete set of MS PowerPoint teaching slides are provided for the instructors to make them teaching this book easily and conveniently.
10. Appendices A–C provide a complete set of instructions and directions to enable users to download and install development tools and kits easily and faster.
11. Good textbook for college students, good reference book for control programmers, software engineers, and academic researchers.

1.3 Who This Book Is For

This book is designed for college students majored in Electrical and Computer Engineering (ECE), Information System Engineering (ISE), Electrical Engineering (EE) and Computer Engineering (CE) for courses such as Automatic Controls, Digital Controls, or Modern Controls, as well as software programmers who want to develop practical and commercial control programming and related development tools. Fundamental knowledge and understanding on C language programming are assumed.

1.4 What This Book Covered

This book is composed of nine chapters with an easy study way to enable students to learn classical and modern control technologies effectively and practically. Each chapter contains home works and exercises as well as lab projects to enable students to perform necessary exercises to improve their learning and understanding for the related materials and technologies.

Chapter 1: Provide an overview and introduction about the book with highlights on outstanding features and organizations of the book.

Chapter 2: Fundamental and basic control technologies, including the classic and modern control strategies, are discussed and introduced in this chapter. Both the open-loop and closed-loop control systems are introduced with some actual examples. Some popular techniques applied in analyzing linear closed-loop control systems are introduced, which include Laplace transform and block diagram. Some important properties that are closely related to Laplace transform, such as poles and zeros, are discussed and analyzed with real examples.

Chapter 3: Provide a fundamental and basic introduction to one of the updated microcontroller units (MCU) LaunchPad™, Tiva™ C Series TM4C123G made by Texas Instruments™. This chapter provides not only the theoretic introductions and discussions about modern control technologies with simulations but also practical and hands-on techniques with actual control programming codes. This combination will enable students to easily and successfully implement theoretic knowledge to real process (DC motors) to build a complete control system. Therefore, it is important to have a clear picture about the configurations on hardware and software about this LaunchPad before we can dig a little deeper.

Chapter 4: Discussed most popular mathematical models for the most often used systems in our world. Different methods are introduced to set up, analyze, and identify the system models to enable users to successfully design desired control systems based on those models.

Chapter 5: The main topic of this chapter is concentrated on the PID control system design and analysis. Three popular designing methods, root locus, Bode plot, and state space, are introduced and discussed in details with quite a few examples. Three popular computer-aided design tools, SISOTool, PIDTuner and MATLAB® Simulink®, are discussed with some actual example systems. The Nyquist stability criterion is also introduced and discussed with some useful MATLAB functions.

Chapter 6: This chapter focuses on nonlinearities that are either unavoidable or inherent in the controlled process model, such as motion kinematics that causes trigonometric functions to appear in the process model as it results directly from the motion geometry. Other nonlinearities discussed in this chapter are those that are added intentionally to the control loop, such as relays, in order to achieve control performance, for certain specific command inputs, which may be superior to that obtained via linear control means. Intensive simulation studies for various nonlinear systems are performed and presented with MATLAB® Simulink® and Nonlinear

Control Toolbox®. Most popular nonlinear control strategies, including single-relay, multi-relay, SISO and MIMO sliding mode controls, Lyapunov stability theorem, and synthesis of Lyapunov functions, are discussed and presented with quite few of real examples.

Chapter 7: Provides detailed discussions about fuzzy logic control system (FLCS) and its implementations. The discussions include fuzzy sets, membership functions, fuzzy control rules, and defuzzification processes. Detailed introductions about the crisp and fuzzy sets, fuzzy set operations, fuzzy rule reasoning, and defuzzifications (including the center of gravity—COG method) are presented and illustrated with some real examples. Some simulation studies are presented to compare a PID controller and an FLC with different time delays, disturbance noises, and uncertain system dynamic models to illustrate the advantage performances of an FLC over a PID controller. A spectrum analysis is performed to provide evidence that an FLC is a better robust controller.

Chapter 8: The major topic on this chapter is the implementation of fuzzy logic controller (FLC) on any closed-loop control system by using actual microcontroller with its programming technologies. Three main operational steps on how to successfully implement a fuzzy logic controller on any closed-loop system are introduced and discussed: Fuzzifications, Control Rule Designs, and Defuzzifications. An actual DC motor closed-loop control system is used as an example to illustrate how to design and implement an FLC on that system with real C programming control codes step by step. A mapping technique between a real fuzzy logic output lookup table and a 2D matrix in C Programming language is discussed with detailed illustrations and example coding process. An online fuzzy output strategy is introduced and implemented by designing a user interface (UI) between the VC++ and MATLAB via MATLAB Engine library method to enable the VC++ to access the FIS built in the MATLAB environments.

Chapter 9: The main topic discussed in this chapter is an interval type-2 fuzzy logic control (IT2 FLC) strategy applied in most popular control systems, including linear and nonlinear systems. An introduction about the differences between a type-1 fuzzy set and an interval type-2 fuzzy set is presented to provide readers a clear and complete picture about these two kinds of fuzzy sets and inference systems. Following this introduction, more detailed and deeper discussions about the interval type-2 fuzzy set (IT2 FS) operations and IT2 fuzzy inference system (FIS) are presented in Sects. 9.2 and 9.3. A simulation study is discussed to help users to familiarize them with the MATLAB Simulink tool to perform simulation studies with an IT2 FLC. A comparison study is presented to compare three control strategies: PID controllers, type-1 and IT2 FLCs with simulation study. Various control factors and situations are simulated to illustrate the advantages of type-1 and IT2 FLC over PID controllers, which includes the time delay, long time delays, random noises, and model uncertainty.

- **Appendix A**—Provides instructions on downloading, installing, and configuring the Keil® MDK-ARM® 5.24a IDE.
- **Appendix B**—Provides instructions on downloading and installing TivaWare™ for C Series software package SW-EK-TM4C123GXL.
- **Appendix C**—Provides detailed directions on downloading and installing Stellaris ICDI debugger and Virtual COM Port on the host computer.

1.5 How This Book Is Organized and How to Use This Book

Chapters 2, 4–7 provide fundamental and professional introductions and discussions about the classical and modern control technologies applied in all aspects in our world today. While Chaps. 3, 8, and 9 provide detailed introductions and discussions about the specified microcontroller TM4C123GH6PM and microcontroller evaluation board TM4C123GXL with Keil MDK-ARM 5.24a IDE as well as software packages used to develop and build professional control programs for closed-loop control systems. Therefore, these Chaps. 3, 8, and 9 can be skipped since they are optional materials for some readers or students who only need basic and fundamental control theories and applications on classical and modern controls. This can be considered as Level I group.

For some readers or students who have already learned some basic and fundamental control knowledge, they can directly go ahead to Level II group, which includes Chaps. 3–8. For those readers, Chap. 2 provides a basic introduction to most popular control models and systems and Chap. 9 presents an Interval Type-2 Fuzzy Logic System (IT2 FLS) which can be skipped since the former had been studied and the latter is a complicated updated fuzzy control technology.

Based on the organization of this book we described above, this book can be used in two categories such as Level I and Level II, which is shown in Fig. 1.1.

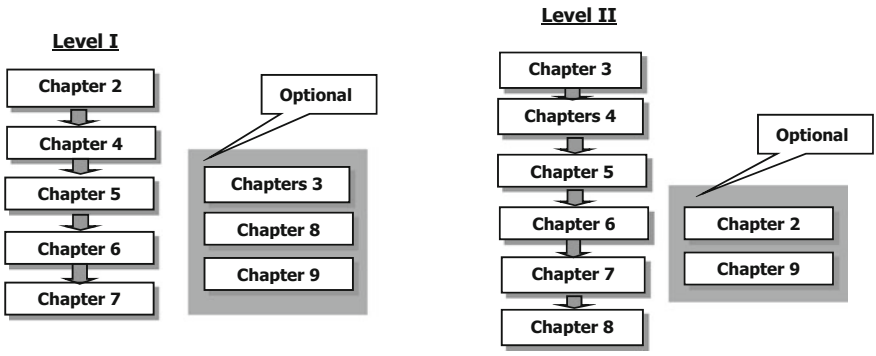


Fig. 1.1 Two study levels in the book

For undergraduate college students or beginning control programmers, it is highly recommended to learn and understand the contents of Chaps. 2, 4–7 since those are fundamental knowledge and techniques used in the classical and modern controls in most implementations (Level I). For the materials in Chaps. 3–8, they are not only related to the basic classical and modern controls but also related to specified microcontroller closed-loop control programming (Level II). This level is matched to graduate college students.

1.6 How to Use the Source Code and Sample Projects

All projects in the book are divided into two parts: the Class Projects and the Lab Projects. All source codes for those projects are available in the book. However, all class projects are available to both instructors and students, but the lab projects are only available to instructors since students need to build these lab projects themselves. All sources codes for these projects have been debugged and tested, and they are ready to be executed in any TM4C123GXL EVB.

All class projects are categorized to associated chapters that are located at the folder **C-M Control Class Projects** that is located at http://extras.springer.com/2019/978-3-030-01381-3. You need to use either the book **ISBN**, the book **Title**, or the **Author** name to access and download these projects into your computer and run each project as you like. To successfully run those projects on your computer, the following conditions must be met:

1. Tiva™ LaunchPad TM4C123GXL Evaluation Board (EVB), TM4C123GXL must have been installed and connected to your host computer.
2. EduBASE ARM® Trainer (contains most popular peripherals and interfaces) must have been installed and connected to the TM4C123GXL EVB.
3. The Keil™ MDK-ARM® 5.24a and above IDE must be installed in your computer.
4. The TivaWare™ SW-EK-TM4C123GXL Software Driver Package should be installed in your computer, and this package must be installed if you want to use any API function provided by TivaWare Peripheral Driver Library.
5. The Stellaris In-Circuit Debug Interface (ICDI) has been installed in your host computer.

Refer to Appendices A ~ C to complete Steps 3 ~ 5.

All book related teaching and learning materials, including the class projects, lab projects, appendices, faculty teaching slides, and homework solutions, can be found from the associated folders located at the Springer Book Support site, as shown in Fig. 1.2.

These materials are categorized and stored at different folders in two different sites based on the teaching purpose (for instructors) and learning purpose (for students):

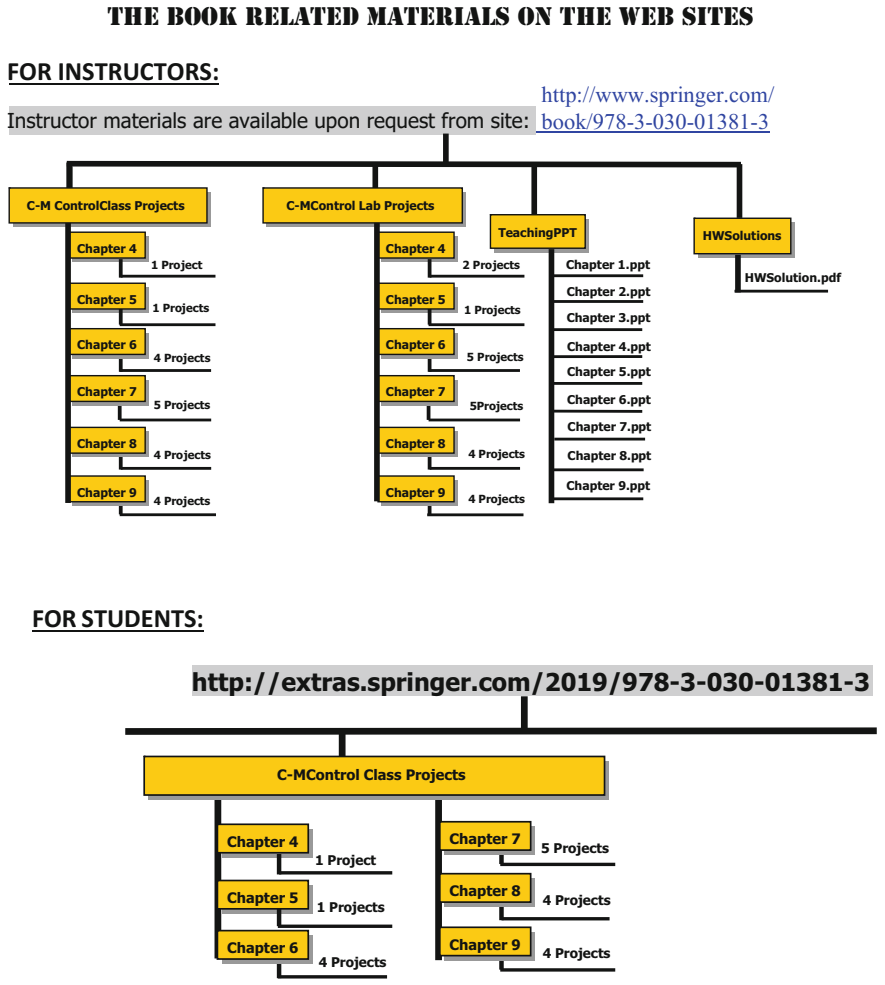


Fig. 1.2 Book related materials on website

For Instructors:

1. **C-M Control Class Projects** Folder: Contains all class projects for different chapters.
2. **C-M Control Lab Projects** Folder: Contains all lab projects included in the homework sections in different chapters. Students need to follow the directions provided in each homework lab section to build and develop these lab projects themselves.
3. **TeachingPPT** Folder: Contains all MS-PPT teaching slides for each chapter.
4. **HWSolutions** Folder: Contains a set of complete solutions for the home works and labs developed and used in the book. The solutions are categorized and stored at the different chapter subfolder based on the book chapter sequence.

For Students:

1. **C-M Control Class Projects** Folder: Contains all class projects in different chapters. Students can download and run these class projects in their host computers after a suitable environment has been set up (refer to conditions listed above).

1.7 Instructors and Customers Supports

The teaching materials for all chapters have been extracted and represented by a sequence of Microsoft PowerPoint files, each file for one chapter. The interested instructors can find those teaching materials from the folder **Teaching PPT** that is located at the site http://www.springer.com/book/978-3-030-01381-3 and those instructor materials are available upon request from the book listing on www.springer.com.

A set of complete homework solution is also available upon request from the book listing on the same site.

E-mail support is available to readers of this book. When you send e-mail to us, please send all questions to the e-mail address: ybai@jcsu.edu.

Detailed structure and distribution of all book-related materials in the Springer site, including the teaching materials for instructors and learning materials for students, are shown in Fig. 1.2.

Chapter 2
Overview of the Fundamentals of Control Systems



2.1 Introduction

Two popular control systems are open-loop and closed-loop control systems. A ***system*** can be defined as an integrated body or an element. For example, a company, a vehicle, a motor, a human being body, or even our Earth, can be considered as a system. A system can also be called a process or a plant. A ***control system*** should contain a controller to perform various control functions to the system to get the desired outputs. Therefore, a complete control system should be composed of inputs, the controller, the system or a process, and the system outputs.

Most control systems in our world are closed-loop control systems, but some of them belong to open-loop control systems.

2.2 The Open-Loop Control System

A typical open-loop control system is a microwave oven and its operational procedure can be illustrated in Fig. 2.1.

By setting the desired time interval (set value), which can be considered as the input, the timer, and the heater (controller) can start and delay the desired period of time to enable the heater to heat the food (process) to the appropriate temperature (output).

The reason we call this kind of control system as an ***open-loop system*** is that the output has no influence or effect on the control action of the input signal. In other words, in this open-loop control system, the output is neither measured nor feedback for comparison with the input. Therefore, an open-loop system is expected to faithfully follow its input command or set point regardless of the final result. This control system is also called a non-feedback control system.

© Springer Nature Switzerland AG 2019

Y. Bai and Z. S. Roth, *Classical and Modern Controls with Microcontrollers*,
Advances in Industrial Control, https://doi.org/10.1007/978-3-030-01382-0_2

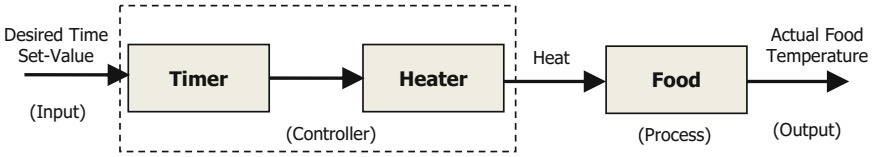


Fig. 2.1 An actual microwave oven heating process

Furthermore, an open-loop system has no knowledge about the output condition, such as the food temperature, so it cannot self-correct any errors it could make when the set value drifts, even if this results in large deviations from the set value.

Another disadvantage of an open-loop system is that they cannot handle disturbances or changes in the conditions which may reduce its ability to complete the desired task. For example, the microwave door opens and the heat is lost. The timing controller continues regardless of the full-time interval but the food is not heated at the end of the heating process. This is because there is no information feedback to maintain a constant temperature that is equal or closed to the input (set value).

One of the possible problems is that the open-loop system errors, such as environmental temperature changing or timer fault operations, can disturb the food heating process and, therefore, requires extra supervisory attention of a user such as an operator. The problem with this anticipatory control approach is that the user needs to monitor the process temperature frequently and take any corrective control action whenever the food heating process is deviated from its desired value of the set value.

Another typical open-loop control system example is a DC motor fan control system, as shown in Fig. 2.2.

In this control system, the control objective is to set up a desired speed value (set value) via the remote control panel to enable the motor to rotate at this speed with no errors. The speed of rotation of the motor will depend upon the input set value that can be converted to a voltage supplied to the motor amplifier. The value of the output voltage could be proportional to the set value on the remote control panel.

If the set value in the remote control panel is high speed, a full positive voltage will be supplied to the amplifier, representing that a high rotation speed should be obtained from the motor and fan. Likewise, if the set value in the remote control

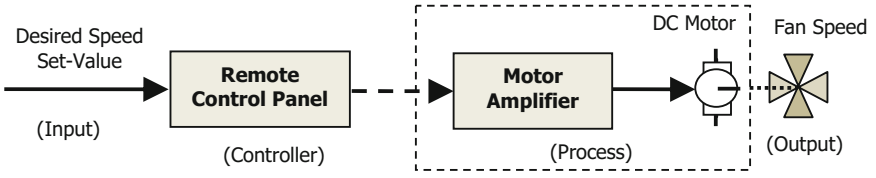


Fig. 2.2 A DC motor open-loop control system

panel is low speed, a low voltage will be supplied to the motor to represent that the motor should rotate in a slow speed. It is clearly desirable that the output speed of the motor should be identical to the set value on the remote control panel, giving the overall gain of the system as unity.

However, the individual gains of the remote control panel, amplifier, and motor may vary over time with changes in supply voltage or temperature, or the motors load (fan) may increase representing external disturbances to the open-loop motor control system. This kind of disturbance could be reduced by periodically monitoring the relationship between the set value and the motor running speed, and by increasing or decreasing the set value manually.

The advantages of this open-loop motor fan control is that it is low-cost and simple and easy to implement, thus making it ideal for use in well-defined systems where the relationship between the input and the output are direct and not influenced by outside disturbances. The shortcoming of this kind of control system lies in the fact, which is that the output (motor rotating speed) has no any knowledge of the input (set value) on the remote control panel. The motor rotating speed may or may not be equal to the set value. In other words, the control system has no any controllability on the output of the system at all! Therefore, this type of open-loop system is not practical since variations or disturbances that existed in the system do affect the speed of the motor, and therefore, this open-loop control system is not reliable in our real world.

However, open-loop control is useful and economic for well-defined systems where the relationship between input and the output can be reliably modeled by a mathematical formula. For example, determining the voltage to be fed to an electric DC motor that drives a constant load, in order to achieve a desired rotating speed would be a good application. But if the loads were not predictable and became excessive, the motor's speed might vary as a function of the loads and not just the input voltage and an open-loop controller would be insufficient to ensure repeatable control of the velocity.

Based on the above discussions, an open-loop control system has the following potential problems:

- There is no comparison between actual input and desired output values.
- Has no self-regulation or self-control action over the output value.
- Each input setting determines a fixed output value for the controller.
- Cannot reduce or overcome all variations or disturbances coming from the external conditions.

In order to solve these potential problems and improve the performance of the entire control system, we need to take a look at another type of control system, closed-loop, or feedback control system.

2.3 The Closed-Loop (Feedback) Control System

A **closed-loop control system** that is also known as a *feedback control system* is a control system which uses the concept of an open-loop system as its forward path has one or more feedback loops (hence its name) or paths between its output and its input. The reference to *feedback* simply means that some portion of the output is returned back to the input to form part of the systems excitation.

Closed-loop systems are designed to automatically achieve and maintain the desired output by comparing it with the actual input condition. It does this by comparing a part or the entire feedback from the output and the desired input to generate an error signal, which is the difference between the output and the reference input. In other words, a closed-loop system is a fully automatic control system in which its control action being dependent on the output in some way.

We still use our previous microwave oven as an example, and it can become a closed-loop control system by adding a feedback path with a sensor that is used to detect the actual temperature of the food and a comparator that is used to compare the feedback output with the input (set value). An error signal can be obtained by comparing the feedback output and the input, and this error signal can be used as input to the controller to automatically adjust the output to make it to the desired input set value, or make the output as closely equal to the input as possible.

This sensor would monitor the actual temperature of the food and compare it with or subtract it from the input reference. The error signal is then amplified by the controller, and the controller output makes the necessary correction to the heating system to reduce any error. For example, if the food is not hot enough, the controller may increase the temperature or the heating time. Likewise, if the food is very hot, then the controller may reduce the temperature or stop the process so as not to overheat or burn the food.

The closed-loop configuration is defined by the feedback signal, derived from the sensor or a thermometer in our food heating system. The magnitude and polarity of the error signal would be directly related to the difference between the required heating temperature and actual food temperature.

The term **closed-loop control** always means the use of a feedback control in order to reduce any errors between the output and the input. Just because of this feedback, it distinguishes the major differences between an open-loop and a closed-loop system. The accuracy of the output thus depends on the feedback path, which in general can be made very accurate and within electronic control systems and circuits.

From this simple example, it can be found that a closed-loop system has many advantages over open-loop systems. The primary advantage of a closed-loop feedback control system is its ability to reduce a system's sensitivity to external disturbances and can automatically correct or modify the error to make the output as equal to the input as possible.

Another point to be noted for this feedback control system is that the feedback path or the sensor must be able to provide the two following functions:

1. Direct a part or the entire output back to the input to allow the comparator to compare the input and the output to get a difference or an error signal as an updated input to the controller.
2. Convert the output type to the input type to make this comparison possible.

As shown in our food heating closed-loop control system in Fig. 2.3, the sensor not only provides a feedback from the output to the input, but also needs to convert the output type (food temperature) to the input type (timer setup value). By adjusting the timer setup value, the food temperature can be modified and changed to the desired temperature. Otherwise, the comparison between the input and the output cannot be executed because of the different types on input and output.

Let us have a closer look at the open-loop DC motor control system shown in Fig. 2.2 and modify it as a closed-loop control system. The modified result is shown in Fig. 2.4.

In order to feedback the output (motor or fan rotating speed) to the input (voltage applied on the motor amplifier), we need to install a tachometer on the rotating axis of the DC motor to detect the rotation speed of the motor and convert it to a voltage that is proportional to the rotating speed of the motor. A tachometer, also known as a tacho-generator and it is simply a permanent magnet DC generator that gives a DC output voltage proportional to the speed of the motor.

Any external disturbances to this closed-loop motor control system such as the motor's load increasing would generate a difference in the actual motor speed and the input voltage set value. This difference would create an error signal, for which the controller would automatically respond to adjusting the motor's speed based on

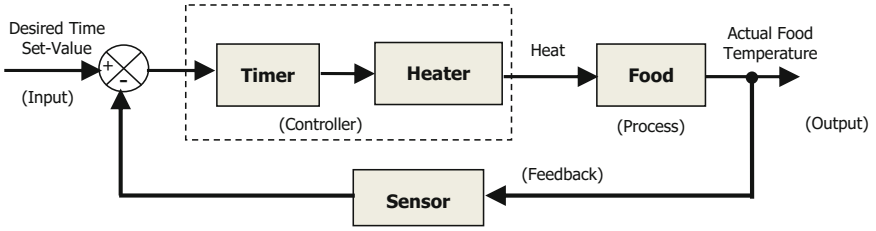


Fig. 2.3 A closed-loop microwave oven control system

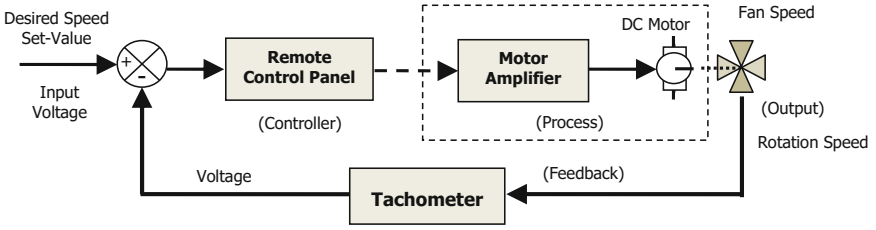


Fig. 2.4 A DC motor closed-loop control system

the amplitude of that error signal. Then, the controller works to try to minimize the error signal, with zero error indicating actual speed which equals to the set value.

Compared to the open-loop control system, closed-loop control systems have many advantages over open-loop systems. One advantage is that the use of feedback makes the system response relatively insensitive to external disturbances and internal variations in system parameters such as temperature or variation on elements. It is thus possible to use relatively inaccurate and cheaper components to obtain the accurate control of a given process or plant.

Based on the discussions above, we can conclude that a closed-loop control system is better than an open-loop control system with the following advantages:

1. Reducing errors by automatically adjusting the systems input.
2. Improving stability of an unstable system.
3. Reducing the systems' sensitivity to enhance robustness against external disturbances or internal variations to the process.
4. Producing a reliable and repeatable performance.

Now that we have a clear picture about two configurations of the control systems, next let us discuss a useful tool to make our control system analysis and study easier and simpler.

2.4 Introduction to Laplace Transformation

As we mentioned at the beginning of this chapter, most systems in our real world belong to nonlinear time-variant (NLTV) systems. It is very difficult and maybe impossible to analyze this kind of systems directly. An easy way to do that is to approximate those NLTV systems to the linear time-invariant (LTI) systems since we can use superposition principle to analyze them, and therefore make our study easier and simpler. The key issue is that the superposition principle only works for LTI systems.

As we know, most LTI systems can be presented by a sequence of differential equations. In order to simplify these differential operations, we need to use Laplace transform to convert those differential equations to the algebraic equations to make our analysis easier.

Let us start from an *RC* low-pass filter circuit as shown in Fig. 2.5.

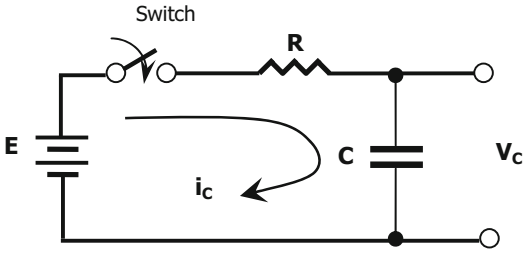


Fig. 2.5 A *RC* low-pass filter

The output voltage V_O is exactly the voltage drop on the capacitor V_C. Based on the Kirchhoff's Voltage Law (KVL), we have the relationship between the input E and the output V_O is described as:

$$E = V_R + V_C = i_C R + V_C = CR\frac{\mathrm{d}V_C}{\mathrm{d}t} + V_C \tag{2.1}$$

or

$$CR\frac{\mathrm{d}V_C}{\mathrm{d}t} + V_C = E \tag{2.2}$$

This is a first-order nonhomogeneous differential equation with generous and special solutions. The complete solution for this equation is

$$V_C(t) = E\left(1 - \mathrm{e}^{-\frac{t}{RC}}\right) \tag{2.3}$$

However, by using the Laplace transform, this first-order differential equation can be simplified to an algebraic equation

$$V_C(s) = \frac{E(s)}{sRC + 1} \tag{2.4}$$

The key issue is that after using the Laplace transform ($\boldsymbol{L}$), a derivative operation can be converted to a multiplication operation such as

$$L\left\{\frac{\mathrm{d}f(t)}{\mathrm{d}t}\right\} = sF(s) \tag{2.5}$$

Another little complicated example is a simple spring-mass-damper mechanical system as shown in Fig. 2.6. Based on Newton's second law of motion, the input–output relationship of this system can be described as

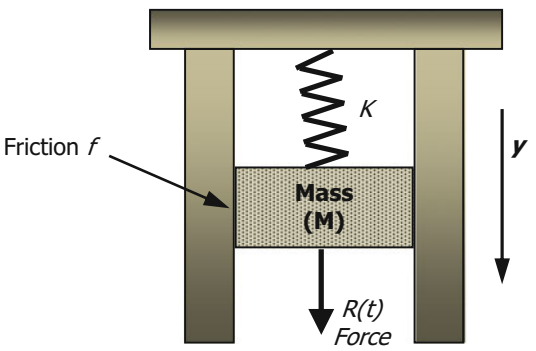


Fig. 2.6 A simple spring-mass-damper mechanical system

$$M\frac{\mathrm{d}^2y(t)}{\mathrm{d}t^2}+f\frac{\mathrm{d}y(t)}{\mathrm{d}t}+Ky(t)=r(t) \tag{2.6}$$

When $r(t) = 0$, $y(0^-) = y_0$, and $y'(0^-) = 0$, the Laplace transform of this second-order differential equation can be written as

$$Y(s)=\frac{(Ms+f)y_0}{Ms^2+fs+K}=\frac{p(s)}{q(s)} \tag{2.7}$$

It can be found from these two examples that the input–output relationships of a complicated system represented by a differential equation can be significantly simplified by using the Laplace transformation method.

Let us have a closer look at the Laplace transformation method.

2.4.1 The Laplace Transform

As shown from above two examples, the Laplace transform method can be used to transfer more complicated differential equations to relatively simple algebraic equations for all systems that can be described by linear differential equations. In other words, the Laplace transform method substitutes the relatively easily solved algebraic equations for more difficult differential equations. In fact, a Laplace transform $F(s)$ is an integration process for the differential equation $f(t)$

$$F(s)=\int_{0^-}^{\infty}f(t)\mathrm{e}^{-st}\mathrm{d}t=L\{f(t)\} \tag{2.8}$$

The prerequisite condition of using the Laplace transform integration is that the integration for $f(t)$ must be converged

$$\int_{0^-}^{\infty}|f(t)|\mathrm{e}^{-\sigma t}\mathrm{d}t<\infty \tag{2.9}$$

where σ is a real positive number. If the magnitude of $f(t)$ can be absolutely converged, which means that if $|f(t)| < Me^{\alpha t}$ for all positive t, the integral will converge for $\sigma > \alpha$. The σ is called the abscissa of absolute convergence.

The inverse Laplace transform can be expressed as

$$f(t)=\frac{1}{2\pi j}\int_{\sigma-j\infty}^{\sigma+j\infty}F(s)\mathrm{e}^{st}\mathrm{d}s \tag{2.10}$$

Table 2.1 Most important Laplace transformation pairs

$f(t)$	$F(s)$
Step function u(t)	$\frac{1}{s}$
e^{-at}	$\frac{1}{s+a}$
$\sin \omega t$	$\frac{\omega}{s^2+\omega^2}$
$\cos \omega t$	$\frac{s}{s^2+\omega^2}$
$e^{-at}f(t)$	$F(s+a)$
t^n	$\frac{n!}{s^{n+1}}$
$f^{(k)}(t) = \frac{\mathrm{d}^k f(t)}{\mathrm{d}t^k}$	$s^k F(s) - s^{k-1}f(0^-) - s^{k-2}f'(0^-) - \cdots - f^{(k-1)}(0^-)$
$\int_{-\infty}^{t} f(t)\mathrm{d}t$	$\frac{F(s)}{s} + \frac{1}{s}\int_{-\infty}^{0} f(t)\mathrm{d}t$
Impulse function δ(t)	1

The inverse Laplace transformation is usually obtained by using the Heaviside partial fraction expansion. This kind of transformation is useful for system analysis and design due to the effect that each characteristic root or eigenvalue can be clearly observed.

Generously, the Laplace variable s can be thought as the differential operator such as

$$s \equiv \frac{\mathrm{d}}{\mathrm{d}t} \tag{2.11}$$

Likely, we can consider the integral operator

$$\frac{1}{s} \equiv \int_{0^-}^{t} \mathrm{d}t \tag{2.12}$$

The regular operational procedure of using the Laplace transformation to transfer and derive the input–output relationship of a control system includes the following steps:

- Develop the differential equations based on the real control system model.
- Transfer the differential equations to the algebraic equations with the Laplace transformation.
- Solve the resulting algebraic equations of the variable of interest.

Table 2.1 lists the most important Laplace transform pairs.

When using the Laplace transformation method to transfer a differential equation to a related algebraic equation, the following points must be kept in mind:

- The control system to be transferred must be described by a linear differential equation. This means that each coefficient for each differential item must be a constant and must be an invariant variable with the time.
- Each item in the differential equation must be absolutely converged.

2.5 The Transfer Functions and Block Diagrams

One of the most popular applications of using Laplace transform method on control systems is the ***transfer function***. The so-called transfer function is exactly a ratio between the Laplace transform of the output variable to the Laplace transform of the input variable, with all initial conditions assumed to be zero. In other words, by using the transfer function, we can get a clear relationship between the output and the input for a control system in the s domain.

When applying a transfer function for any control system, the following points must be remembered:

- The transfer function can only be used for a linear or stationary system, or for a control system that can be described by a linear differential equation. Because a nonstationary system, often called a time-varying system, includes some time-varying coefficients, for which the Laplace transform method may not work or may not be available.
- A transfer function only provides the input–output relationship of a system. Thus, the transfer function itself may not contain any detailed information about the internal structure of the system and its behavior.

By using the Laplace transform method, the transfer function for our *RC* low-pass filter circuit can be expressed as

$$G(s) = \frac{V_C(s)}{E(s)} = \frac{1}{sRC + 1} \tag{2.13}$$

Regularly, we use $G(s)$ to represent a transfer function.

The transfer function for our simple spring-mass-damper system can be written as

$$G(s) = \frac{Y(s)}{R(s)} = \frac{1}{Ms^2 + fs + K} \tag{2.14}$$

After using the transfer function, the input–output relationship of a control system can be clearly and easily described by a ratio of an output Laplace transform and an input Laplace transform. In many control systems, the system is composed of several components and these components are connected as a chain with one input coming from the previous output, therefore, the equation of the entire control system can be written as a concatenating sequence of each separate unit. In order to

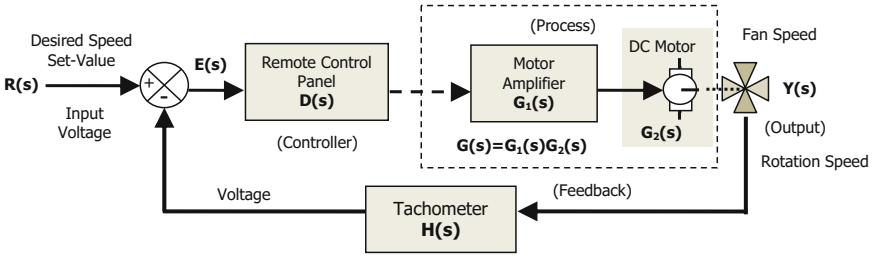


Fig. 2.7 The block diagram of our DC motor closed-loop control system

represent these relationships, a graph called a ***block diagram*** is always used to simplify those components. For example, for our DC motor closed-loop control system, its block diagram can be expressed as shown in Fig. 2.7.

Both the motor amplifier and the DC motor can be expressed as two separate processes, $G_1(s)$ and $G_2(s)$, respectively, in a series format. This series connection can be combined with a single process $G(s) = G_1(s)\ G_2(s)$. The controller can be mapped to the $D(s)$ and the feedback path can be represented by $H(s)$. The Laplace transform of the input and the output can be written as $R(s)$ and $Y(s)$, respectively.

Based on Fig. 2.7, the output and the input relationship can be expressed as

$$Y(s) = D(s)G(s)E(s) \tag{2.15}$$

$$E(s) = R(s) - H(s)Y(s) \tag{2.16}$$

Place (2.16) into (2.15), we have

$$Y(s) = D(s)G(s)[R(s) - H(s)Y(s)] \tag{2.17}$$

Therefore, the output and input ratio are

$$\frac{Y(s)}{R(s)} = \frac{D(s)G(s)}{1 + D(s)G(s)H(s)} \tag{2.18}$$

Equation (2.18) is a typical transfer function for a closed-loop control system. The product $D(s)G(s)$ in the numerator can be called the open-loop or the forward gain, and the product $D(s)G(s)H(s)$ is called the loop gain.

2.6 The Poles and Zeros

By using the Laplace transfer and transfer functions, the analysis and study of control systems become easier and simpler. Going a little deeper, we can find some interesting properties for a transfer function.

For example, we have a process transfer function $G(s)$ represented as

$$G(s) = \frac{2s+1}{s^2+3s+2} = \frac{2(s+0.5)}{(s+1)(s+2)} = \frac{3}{s+2} - \frac{1}{s+1} = \frac{p(s)}{q(s)} \tag{2.19}$$

To make numerator $p(s)$ equal to zero, we have $s = -0.5$. Likely to make denominator $q(s)$ equal to zero, then we get $s_1 = -1$ and $s_2 = -2$. The s-value to make the numerator equal to zero is called zeros and the s-values to make denominator equal to zero are called poles. Both poles and zeros can be presented in s-plane with horizontal axis as real and vertical as an imaginary axis, as shown in Fig. 2.8.

If we use the inverse Laplace transform method, the process function in the time domain is

$$L^{-1}\{G(s)\} = h(t) = 3e^{-2t} - e^{-t} \quad (t \geq 0) \tag{2.20}$$

In fact, for a more complicated control system, more poles and zeros may be involved in the transfer function and make the analysis more difficult. But regularly the poles, exactly the locations of the poles, in a transfer function play more important roles and they can determine the primary characteristics or properties for a control system.

The locations of the poles in the s-plane uniquely determine the primary properties of a control system. As shown in Fig. 2.9, if the pole is located in the origin, the response or the output of the system is a unit response $u(t)$. If the poles are located along the real axis, the system response is a real function. Depending on the poles' locations either in the left or right s-plane, the response would be decayed exponentially when the poles are in the left s-plane. The further to the left or more far away from the origin, the faster the decay to be occurred. However, if the poles are in the right s-plane, the inverse situation happened and the system response would be diverged exponentially. The further to the right or the more far away from the origin, the faster the divergent to be occurred.

Since the system would be decayed or converged exponentially as the poles are in the left s-plane, so we called those systems as stable systems. Likely, the system

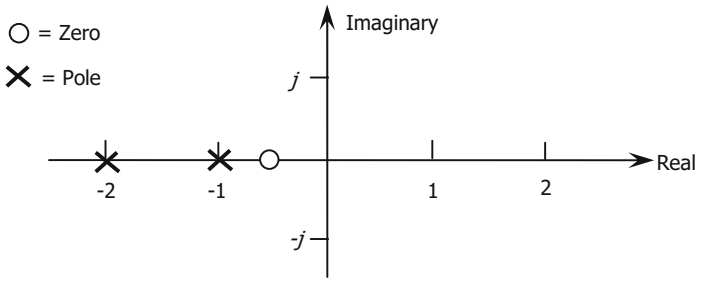


Fig. 2.8 Zeros and poles in a s-plane

responses would be diverged exponentially when the poles are in the right *s*-plane, those systems are called unstable systems since the response amplitudes would become large and large, and finally, a periodic or nonperiodic oscillation would occur to the system to make the system unstable.

When the poles are located at the imaginary axis, a stable oscillation would occur with the definite amplitude. The upper to the origin or the real axis, the higher frequency of oscillations can be obtained.

A more complicated situation is that a pair of complex conjugate poles that are located in neither real nor imaginary axes such as a pole $\boldsymbol{A}$ and its complex conjugate $\boldsymbol{A}^*$ as shown in Fig. 2.9. Similarly, depending on the location of the pole or its complex conjugate pole, the system would be a stable system if the pole is located at the left *s*-plane and a decayed oscillation is occurred since the amplitude of that oscillation would be decayed exponentially, and finally converged to infinitive small value. Otherwise, the system is an unstable one if the pole is at the right s-plane since a diverged oscillation would occur.

As an example, our *RC* low-pass filter circuit is a stable system since only one pole is applied on that system and it is located at the left *s*-plane, $s = -1/\mathrm{RC}$.

It looks like that zeros do not play the same important roles as poles did for a transfer function. However, a useful usage of zeros is to make an unstable system become a stable system by canceling or compensating unstable poles located at the right *s*-plane with zeros located at the same locations.

To get the poles for our example DC motor closed-loop control system, let denominator equal to zero such as $1 + D(s)G(s)H(s) = 0$.

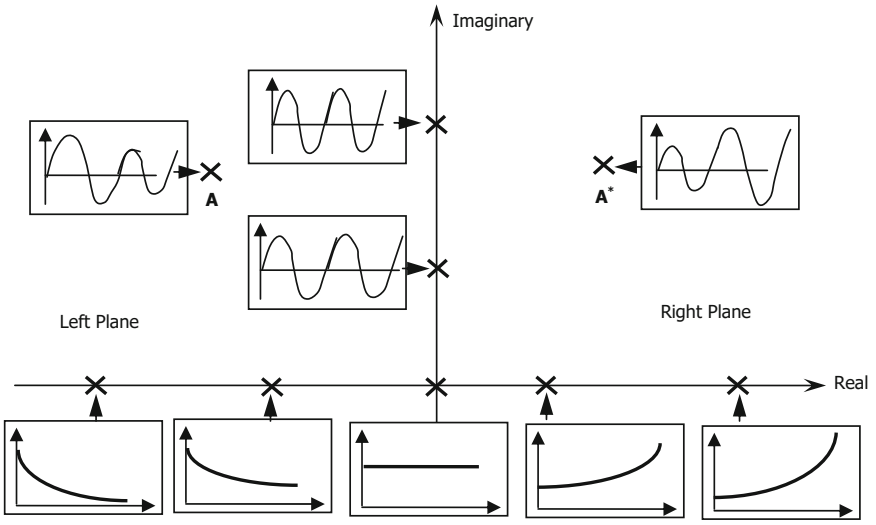


Fig. 2.9 System responses associated with the poles' locations

2.7 The Linear Time-Invariant Control Systems

As mentioned in the last section, transfer functions only work for linear time-invariant (LTI) system, and cannot work for linear time-variant (LTV) or nonlinear systems. In this section, we will have a deep discussion about the LTI system. First, let us concentrate on the time-invariant system.

2.7.1 The Time-Invariant System

The output $y(t)$ for a given input $x(t)$ can be defined as

$$y(t) = f[x(t)] \tag{2.21}$$

This function can be represented by a notation defined as

$$x(t) \mapsto y(t) \tag{2.22}$$

A continuous system is called time-invariant system when the input is delayed by t_0, then the output is also delayed by the same amount. This can be expressed by the following notation:

$$x(t - t_0) \mapsto y(t - t_0) \tag{2.23}$$

A block diagram view of a time-invariant property is shown in Fig. 2.10.

In the top branch, the input is first delayed by t_0, and then it is applied to the system to get the output $z(t)$. In the lower branch, the input is first applied to the system to get the output $y(t)$, and then delay that output by t_0. If we can confirm that both outputs, $z(t)$ and $y(t - t_0)$ are equal, then we can say that this system is a time-invariant system.

Let us use this method to test a square-law system $y(t) = [x(t)]^2$ to see whether it is a time-invariant system.

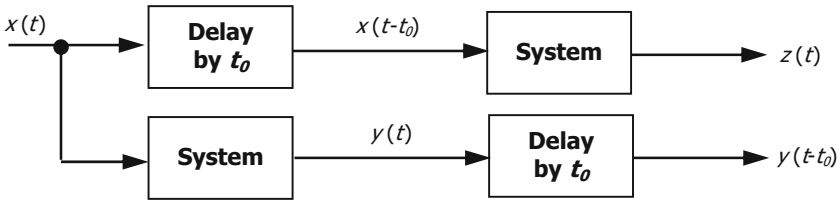


Fig. 2.10 Testing block diagram for time-invariant system

For the top branch, we get the output of this square-law system as $z(t) = [x(t - t_0)]^2$. For the lower branch, we have $y(t - t_0) = [x(t - t_0)]^2 = z(t)$. Therefore, the square-law system is a time-invariant system.

Another example is the time-flip system $y(t) = x(-t)$. Similarly, for the top branch, we get $z(t) = x(-t - t_0)$. For the bottom branch, we have $y(t - t_0) = x[-(t - t_0)] = x(-t + t_0)$. Since $z(t) \neq y(t - t_0)$, therefore, the time-flip system is not a time-invariant system.

By using the same method, one can test whether the system $y(t) = tx(t)$ is a time-invariant system or not.

2.7.2 The Linear System

A linear system is defined based on the following facts (Homogeneity and Additivity):

- If two subsystems $x_1(t) \mapsto y_1(t)$and $x_2(t) \mapsto y_2(t)$,
- Then $x(t) = \alpha x_1(t) + \beta x_2(t) \mapsto y(t) = \alpha y_1(t) + \beta y_2(t)$.

This linearity definition must be true for any kinds of constants α and β. This definition states that if the input is composed of a sum of scaled sequences, then the output is also a sum of scaled outputs corresponding to the individual input sequences. A block diagram view of the linearity property is shown in Fig. 2.11.

In the top branch, two inputs, $x_1(t)$ and $x_2(t)$, are inputs to the system to get two outputs $y_1(t)$ and $y_2(t)$, respectively. Then, both outputs are scaled by multiplying two constants, α and β, to get the sum or the output $z(t)$. In the lower branch, two inputs, $x_1(t)$ and $x_2(t)$, are first scaled and summed, and then fed into the system to get the output $y(t)$. If we can confirm that both outputs, $z(t)$ and $y(t)$, are identical, then the system is a linear system.

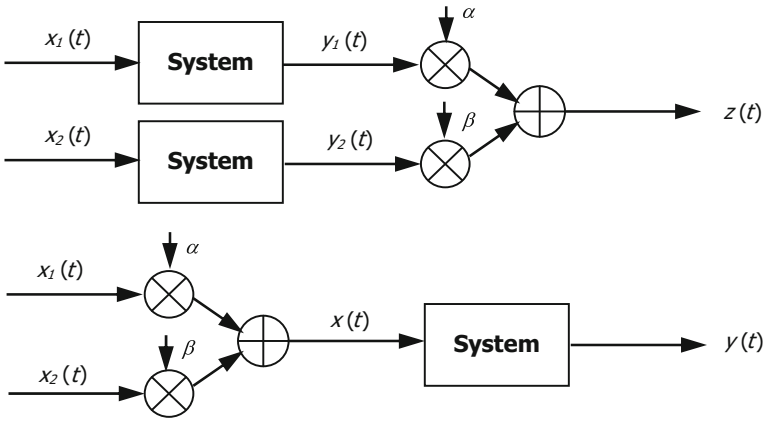


Fig. 2.11 Testing linearity property block diagram

The linearity condition is equivalent to the superposition principle, which means that if the input is a sum of two or more scaled sequences, the output can be found from each sequence acting alone and then by adding those separate scaled outputs together. In some situations, this combined condition can be divided into two conditions, assumed that $\alpha = \beta = 1$

$$x(t) = x_1(t) + x_2(t) \mapsto y(t) = y_1(t) + y_2(t) \tag{2.24}$$

and

$$x(t) = \alpha x_1(t) \mapsto y(t) = \alpha y_1(t) \tag{2.25}$$

Now, let us test some systems to see whether they are linear systems or not. For example, for the square-law system, $y(t) = [x(t)]^2$. Applying it to the top branch, we have

$$z(t) = \alpha[x_1(t)]^2 + \beta[x_2(t)]^2$$

Then, apply two inputs to the lower branch, we get

$$y(t) = [\alpha x_1(t) + \beta x_2(t)]^2 = [\alpha x_1(t)]^2 + 2\alpha\beta x_1(t)x_2(t) + [\beta x_2(t)]^2$$

Therefore, it is clearly indicated that $z(t) \neq y(t)$, so the square-law system $y(t) = [x(t)]^2$ is not a linear system.

There are some other definitions for a linear system such as a system that can be described by a linear differential equation belongs to a linear system. The so-called linear differential equations are those equations from which each coefficient for each differential item is a constant, and each of them cannot be a function of the time or it does not depend on the time.

Some typical analysis tools used for LTI systems include Laplace transform, Fourier transform in the continuous domain, and Z transform in the discrete domain. Traditional design methods used for LTI systems contain the frequency method with Bode plot, Nyquist stability criterion method, root locus method, and state space method. We will discuss these analysis and design methods in the following related chapters.

In addition to those traditional analysis and design methods, some modern methods and tools are also available such as MATLAB® Control System Toolbox™, MATLAB PID Control system tuning tool, and MATLAB® Simulink® used for simulation purpose.

Comparing to the LTI system, the LTV systems are those systems whose coefficients are time-varying parameters and they are not constants. This means that at each different moment, the coefficients for a differential equation have different values, and the exact value is dependent on the time at that moment. Some linear time-variant systems may behave more like nonlinear systems, if the system changes quickly or significantly different between measurements.

The following time-varying systems cannot be modeled by assuming that they are time-invariant systems:

- Aircraft—Time-variant characteristics are caused by different configurations of control surfaces during takeoff, cruise and landing as well as constantly decreasing the weight because of consumption of fuel.
- The Earth's thermodynamic response to incoming solar irradiance varies with time due to changes in the Earth's albedo and the presence of greenhouse gases in the atmosphere.
- The human vocal tract is a time-variant system with its transfer function at any given time depends on the shape of the vocal organs. As with any fluid-filled tube, resonances (called formants) change as the vocal organs such as the tongue and velum move. Mathematical models of the vocal tract are therefore time-variant, with transfer functions often linearly interpolated between states over time.
- Linear time-varying processes such as amplitude modulation occur on a time scale similar to or faster than that of the input signal. In practice, amplitude modulation is often implemented using time-invariant system nonlinear elements such as diodes.
- The discrete wavelet transform, often used in modern signal processing is time-variant because it makes use of the decimation operation.

A more challenging control system is nonlinear time-variant system, and some special methods and techniques must be utilized to handle those systems.

2.8 The Nonlinear Time-Variant Control Systems

Nonlinear control theory deals with systems that are either nonlinear or time variant, or both. Nonlinear control systems are those control systems where nonlinearity plays an important role, either in the process (plant) or in the controller itself. In fact, nonlinear control theory is concerned with the analysis and design of nonlinear control systems.

Nonlinear control theory covers a wide range of systems in which the superposition principle cannot be applied. The nonlinear control theory applies almost all real-world systems since all real control systems are exactly nonlinear. These nonlinear systems can be expressed by using nonlinear differential equations.

Generally, a nonlinear system can be defined as

$$\begin{aligned} x'(t) &= f(t, t_0, x, x_0) \\ x(t_0) &= x_0 \end{aligned} \tag{2.26}$$

where f is a nonlinear function of the time, x is the system state, and x_0 is the initial condition. If the initial condition is known, the function can be simplified as

$$x'(t) = f(t, x) \tag{2.27}$$

The general solution of this equation is given by:

$$x(t) = x_0 + \int_{t_0}^{t} f(\tau, x)\mathrm{d}\tau \tag{2.28}$$

A block diagram of a typical nonlinear feedback control system is shown in Fig. 2.12.

A typical nonlinear system is the trajectory between the output current and the output voltage of a Metal-Oxide Semiconductor Field-Effect Transistor (MOSFET), or $i_D \sim v_{DS}$, as shown in Fig. 2.13. The area on the left of the dash line is called a non-saturation region since the current–voltage relationship is a nonlinear curve. However, on the right of that dash line, it is called the saturation region because the current looks like a constant no matter how large the value is on the voltage.

For a general nonlinear control system, we can divide our study into two sections; Analysis section and Design section.

The following methods or tools can be used for analysis of a nonlinear control system:

- Phase Plan Method
- Limit Cycle and Describing Function
- Small Gain Theorem
- Circle Criterion
- Popov Criterion
- Center Manifold Theorem
- Lyapunov Stability Analysis

The following methods and tools are widely implemented in the design of nonlinear systems:

- Local Linearization and Gain Scheduling
- Feedback Linearization
- Backstepping

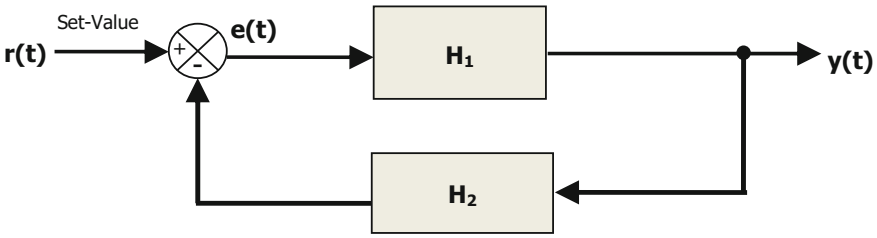


Fig. 2.12 A closed-loop negative feedback nonlinear control system

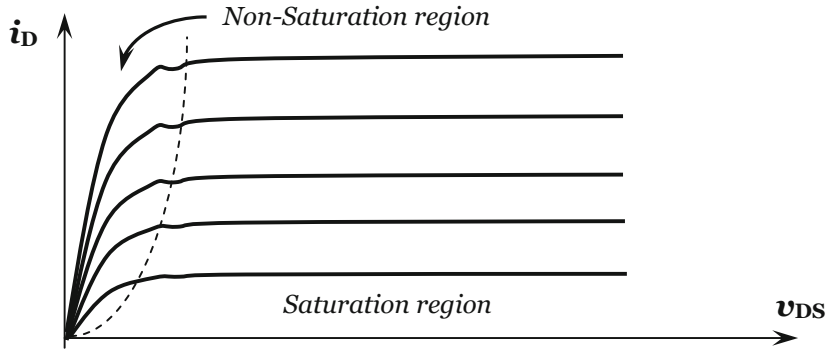


Fig. 2.13 A nonlinear relationship between i_D and v_{DS}

- Lyapunov Redesign
- Sliding Mode Control
- Nonlinear Damping

In recent years, some new modern control strategies have been developed to help and assist the design of nonlinear control systems, and these techniques include:

- Optimal Control
- Model Predictive Control
- Adaptive Control
- Fuzzy Logic Control
- Neural Network Control

In addition to these methods, some useful and powerful modern control toolboxes built by MathWorks® Inc, such as Control System Toolbox™, MATLAB PID Tuner, MATLAB® Simulink®, Fuzzy Logic Toolbox™, Neural Network Toolbox™, and MATLAB SISO Tool, are very popular in designing and developing controllers for modern control systems. We will have detailed discussions about some of these analysis and design methods in Chaps. 5 and 7.

2.9 The Continuous Control Systems

So far, all control systems we have discussed belong to continuous control systems, which means that all control functions $f(t)$ are continuous functions of the time t. Based on the basic definition in mathematics, a continuous function can be defined as:

If a function is called continuous, which means that at any moment of the time for a given interesting range, the function corresponding to that moment has a definite value, and both left and right limitation values are equal to the function value at that moment.

Interpreting this definition to equations, as shown in Fig. 2.14, it can be said If $\lim_{t \to t_0^-} f(t) = \lim_{t \to t_0^+} f(t) = f(t_0)$ $(t \in R)$, then the function $f(t)$ is a continuous function in R.

For any continuous function, both variables such as t and the function $f(t)$ must be continuous and a corresponded Laplace or Fourier transform exists.

Based on this definition, all systems in our real world are continuous systems. All of our previous examples, such as *RC* low-pass filter, simple spring-mass system, DC motor control system, are continuous systems.

A point to be noted is that continuous signals can be real mathematical functions, but some transforms applied on signals can produce complex signals that have both real and imaginary parts. For example, in our Fig. 2.9, a pair of complex conjugate poles, $\boldsymbol{A}$ and $\boldsymbol{A}^*$, are located at the s-plane, and these two poles have both real and imaginary parts. The physical meaning for these two poles is two oscillation signals with the same frequency, but one pole located in the left s-plane is converged exponentially and the other one pole located in the right s-plane is diverged exponentially.

Almost all real systems in our world belong to continuous systems, including our previous example systems, such as *RC* low-pass filter, simple spring-mass and DC motor closed-loop control system. The continuous systems can also be called analog systems, and analog controllers should be designed for those analog systems to perform control functions. However, today most or maybe all control jobs are handled by computers. As we know, computers are digital logic devices and they can only handle or process digital signals or systems, not analog or continuous signals or systems. Therefore, we need a converter to convert analog signals to digital signals before we can input them to computers to enable computers to process them.

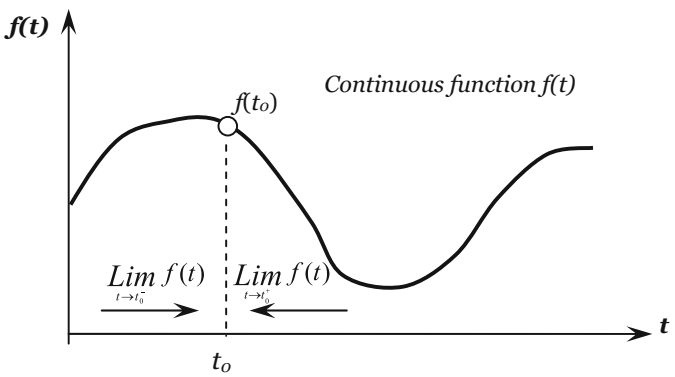


Fig. 2.14. A continuous function $f(t)$

2.10 The Discrete and Digital Control Systems

In order to enable computers to work as a controller to perform desired control jobs, any continuous or analog signal needs to be converted to discrete or digital signal since computers can only recognize and process digital signal due to its hardware properties. As we know, the basic component to make a computer system is MOSFET, and we can integrate millions or even billions of MOSFETs in a very small semiconductor chip with Very Large-Scale Integrated (VLSI) circuit technique to get powerful control functions. Any MOSFET, when working as a digital logic gate or flip-flop, it can only respond to two values, either 0 or 1, or either logic Low or logic High, or either False or True. Just because of this property, the computers can only response or recognize these two logic values (binary numbers), or a combination of these binary values.

A useful component used to perform this analog-to-digital signal conversion is called an Analog-To-Digital Converter (ADC), which is composed of a sample-hold circuit and an ADC, as shown in Fig. 2.15.

A MOSFET and a capacitor make up a sample-hold circuit that is a zero-order holder. The MOSFET works as a switch to periodically on and off to charge and discharge the capacitor, and the capacitor works as a data holder. This sample-holder circuit is used to sample and hold the analog input during each cycle time T_S when the MOSFET is turned on to connect the input to charge to the capacitor to get an input sample. Since the input to the ADC cannot be changed or modified during the A/D conversion time period, this hold circuit is necessary.

The output of the ADC is a sequence of discrete signals with holds sampled values during the conversion time period. Now, this kind of signal is not a continuous signal anymore since only sampled values are located at each sampling point $4\sin(nTs)$ with all other values being zero. Also, the values in the horizontal axis are not continuous values. This kind of signal is called discrete signal, but not digital signal.

There is some small difference between a discrete signal and a digital signal, which means that:

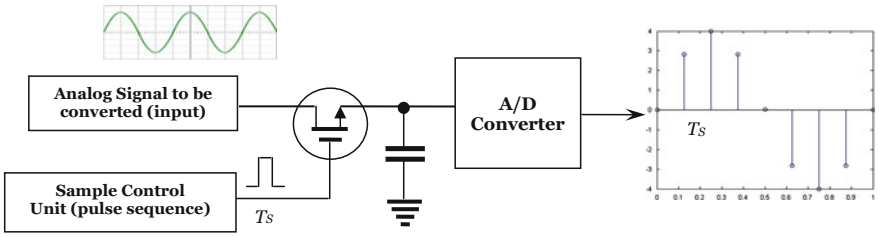


Fig. 2.15 An illustration of a typical ADC

- A discrete signal cannot be called a digital signal since its values are analog values or continuous values, see the output of Fig. 2.15. For example, when the horizontal value or the sampling time is 0.1, the sampled output is 3, and when $Ts = 0.3$, the sampled output is 4.0.
- The digital signal only contains digital or binary values. In order to convert this discrete signal to digital signal, one needs to perform the digitalization process to convert all analog amplitudes into digital values. For example, when $Ts = 0.1$, the output value should be 00000011 (decimal 3), and as $Ts = 0.3$, the output should be digitalized to 00000100 (decimal 4). Here, an 8-bit binary value is assumed and adopted.

When using a computer to work as a digital controller, we need both Analog-To-Digital Converter and Digital-To-Analog Converter since the control target is a continuous plant.

For example, for our DC motor closed-loop control system, in order to use a computer to replace the remote controller, we need to add an ADC after the error detection section to convert the analog error signal into a corresponding digital signal, and fed into the computer who works as a digital controller. Also, we need to add a Digital-to-Analog Converter (DAC) on the output of the computer to convert it back to the analog control signal to be applied on the motor amplifier and the DC motor since both of them are analog components, and only analog or continuous signals can be accepted by them. A complete control system using a computer as a digital controller is shown in Fig. 2.16.

As we know, the Laplace and Fourier transforms are only working for continuous signals, and they cannot work for discrete or digital signals. In order to handle discrete and digital signals, we need to use Discrete Laplace or Discrete Fourier transforms to process those signals. A useful transform called z-transform can also worked for discrete and digital signals.

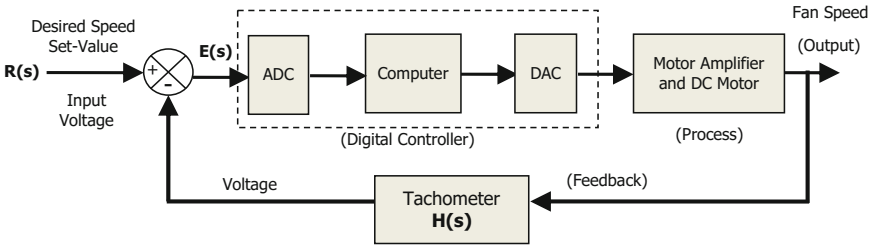


Fig. 2.16 A complete closed-loop control system with a digital controller

2.11 Summary

Fundamental and basic control technologies, including the classic and modern control strategies, are discussed and introduced in this chapter.

Starting Sect. 2.2, the open-loop and closed-loop control systems are introduced with some actual examples. The advantages of two control systems are discussed and analyzed with real examples. Also, the significant differences between these two control systems are highlighted and emphasized to provide readers with a clear picture of these control systems.

Some popular techniques applied in analyzing linear closed-loop control systems are introduced, which include Laplace transform and block diagram. Some important properties that are closely related to Laplace transform, such as poles and zeros, are discussed and analyzed with real examples in Sect. 2.6.

Following those sections, some important and popular control terminologies, such as linearity and time-invariant, are introduced with linear time-invariant (LTI) systems. The nonlinear and linear time-variant control systems are discussed in Sect. 2.8. Some important and popular analysis and design methods used for nonlinear control systems are listed and discussed in this section, including both traditional and modern control methods.

In Sects. 2.9 and 2.10, the continuous systems, discrete systems, and digital systems are discussed. The difference between the discrete signal and digital signal are highlighted with examples.

Homework

2.1 What is the open-loop control system? What is the closed-loop control system? What is the major difference between them?

2.2 Indicate which are open-loop or closed-loop control systems for the following systems.

(a) A man driving a vehicle.
(b) A solider shooting a target.
(c) The water level controlled by a float and valve.
(d) The room temperature control system—Air Conditioner.
(e) A microwave heating process.
(f) A teaching and learning process.
(g) A washing and drying process.

2.3 For those closed-loop control systems shown in 2.2 above, draw related block diagrams and indicate

(a) The reference input (set value)
(b) The controller
(c) The process (plant)

(d) The feedback component
(e) The system output

2.4 A closed-loop control system has a transfer function $G(s) = \frac{2s+1}{s^2+2s+5}$.

(a) How many poles and zeros are in this system? What are the values?
(b) Is the system stable or not? Why?

2.5 For each of the following systems, determine whether or not the system is (1) Linear (2) Time-invariant.

(a) $y(t) = x(t)\cos(200\pi t)$
(b) $y(t) = x(t) - x(t-1)$
(c) $y(t) = |x(t)|$
(d) $y(t) = Ax(t) + B$ (A and B are constants)

2.6 Draw a block diagram of a home heating system, and identify the function of each component and input and output signals.

2.7 An ADC can convert an input analog voltage from 0 to 12 V with a resolution of 5 mV. Determine how many bits the ADC must have?

Chapter 3
Introduction to Tiva C MCU LaunchPad™—TM4C123G



3.1 Introduction

One of Texas Instruments' products, Tiva™ C Series MCU—TM4C123G, is a LaunchPad™ that contains typical and popular ARM® Cortex®-M4 MCUs system. A related Evaluation Kit or Board (EVB), TM4C123GXL, which is built with two ARM® Cortex®-M4 microprocessors or MCUs, TM4C123GH6PM, is used to support and assist the development process with that MCU system. In this book, we will use this EVB combined with an EduBASE ARM Trainer to build and develop actual control system projects.

The entire and complete development tools and kits can be categorized into two parts; the ***Hardware Package*** and the ***Software Package***.

The ***hardware package*** includes

- TM4C123GH6PM Microcontroller Unit (MCU).
- Tiva™ C Series LaunchPad™ TM4C123GXL Evaluation Board (EVB).
- EduBASE ARM® Trainer (contains most popular peripherals and interfaces).
- Some other related peripherals and interfaces, such as DC Motors, Pulse Width Modulation (PWM) module and Quadrature Encoder Interface (QEI).

The ***software package*** includes

- Integrated Development Environment Keil® ARM®-MDK µVersion®5.24a (IDE).
- TivaWare™ SW-EK-TM4C123GXL Software Driver Package.
- Stellaris In-Circuit Debug Interface (ICDI).

Appendices A∼C in this book provide detailed information and directions for downloading and installing these software tools in your host computers. A complete configuration or function block diagram of this develop environment is shown in Fig. 3.1.

© Springer Nature Switzerland AG 2019

Y. Bai and Z. S. Roth, *Classical and Modern Controls with Microcontrollers*,
Advances in Industrial Control, https://doi.org/10.1007/978-3-030-01382-0_3

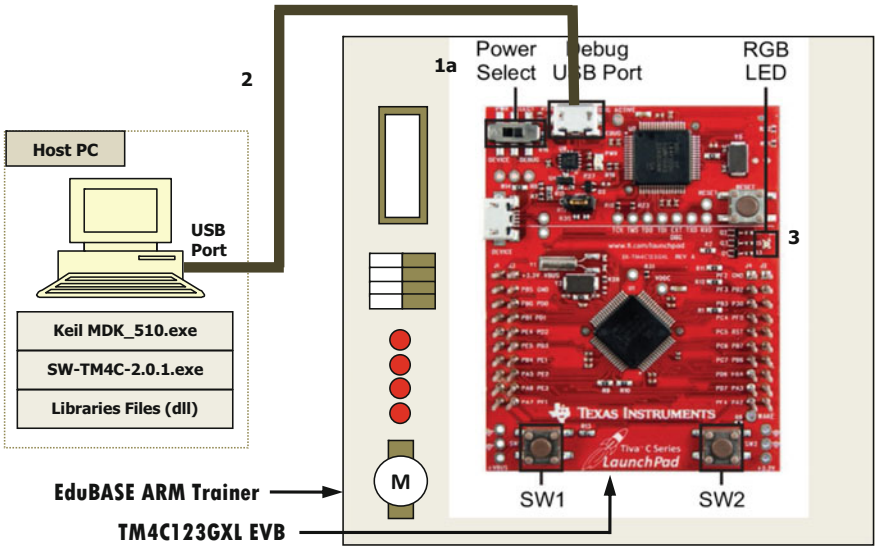


Fig. 3.1 The complete development environment. *Courtesy* Texas Instruments

To connect the Tiva™ C Series LaunchPad™ evaluation board TM4C123GXL to the host computer, follow the operations listed below (Fig. 3.1):

1. Setup the Tiva C Series TM4C123GXL evaluation board by
   a. Switching the **Power Select** to the right for the **Debug Mode**
2. Connect the USB cable from the USB Port in the host PC to the **Debug USB Port** on the Tiva C Series TM4C123GXL board, as shown in Fig. 3.1.
3. Insert the TM4C123GXL EVB into the EduBASE ARM Trainer via Booster Pack XL Interface Connectors J1 ~ J4 in the EVB with J14 and J19 on the EduBASE ARM Trainer, as shown in Fig. 3.1.

Next let us have a closer look at these hardware and software packages. First let's concentrate on the hardware package.

3.2 Introduction to the Hardware Package

The hardware package is composed of four major parts

1. TM4C123GH6PM Microcontroller Unit (MCU), including on-chip memory and General Purpose Input and Output (GPIO) ports.
2. Tiva™ C Series LaunchPad™ TM4C123GXL Evaluation Board (EVB).

3. EduBASE ARM® Trainer (contains most popular peripherals and interfaces).
4. Some other related peripherals and interfaces, such as DC Motors, Pulse Width Modulation (PWM) module, and Quadrature Encoder Interface (QEI).

We will discuss these parts one by one in this sequence. For DC Motor, QEI and PWM, will be discussed separately on different sections later with different chapters. Let us start from the TM4C123GH6PM MCU.

3.2.1 *Overview TM4C123GH6PM Microcontroller Unit (MCU)*

The TM4C123GH6PM Microcontroller Unit (MCU) is a high-performance embedded controller with multiple functions and advanced features. Unlike ARM® Cortex®-M4, this MCU contains quite a few components, such as on-chip memory and some on-chip peripherals as well as various peripheral device interfaces, and integrates them into this chip. The main components embedded in this MCU include (Fig. 3.2)

- A 32-bit ARM® Cortex®-M4F Processor Core with a Floating-Point Unit (FPU). The CPU speed is 80 MHz with 100 DMIPS performance.
- On-Chip Memory Devices include
  - 256-KB single-cycle Flash memory
  - 32-KB single-cycle SRAM
  - 2-KB EEPROM
  - Internal ROM loaded with TivaWare™ for C Series software
- A System Timer (SysTick).
- Six physical General-Purpose Input–Output (GPIO) ports or blocks.
- Two high-speed 12-bit Analog to Digital Converters (ADCs) up to 1 MSPS.
- Three independent Integrated Analog Comparators (IACs) and 16 Digital Comparators.
- Two Control Area Network (CAN) 2.0 A/B controllers.
- Optional full-speed USB 2.0 OTG/Host/Device.
- Six 16/32-bit General-Purpose Timer (GPTM) and 6 32/64-bit Wide GPTM blocks.
- Two PWM modules with each having four PWM generator blocks and a control block and this makes a total of 16 Pulse Width Modulation (PWM) outputs.
- Two Watchdog Timers (WDTs).
- Serial communication with eight UARTs, six I^2Cs, four Serial Peripheral Interface (SPI) or Synchronous Serial Interface (SSI).
- Two Quadrature Encoder Interface (QEI) modules.
- Intelligent low-power design power consumption as low as 1.6 μA.

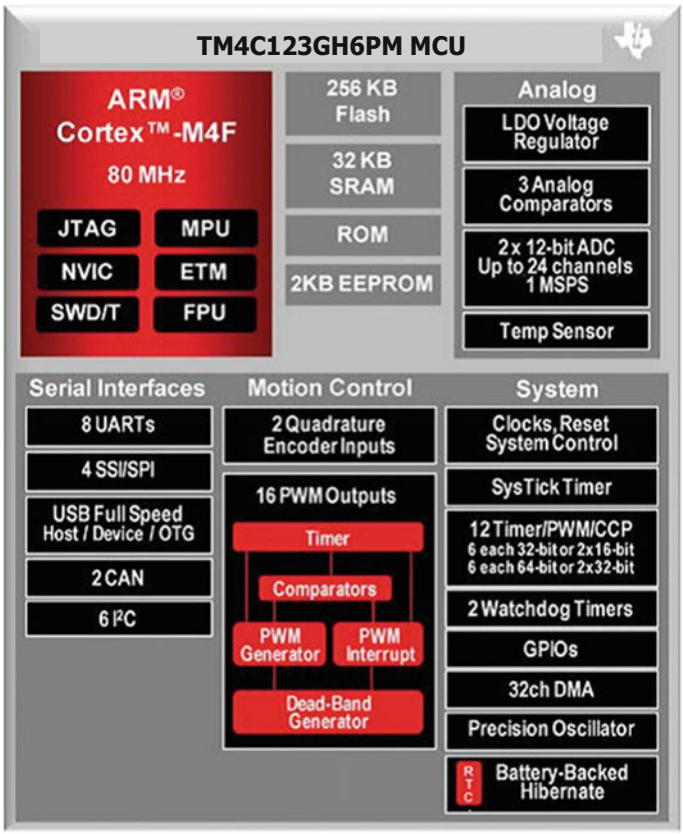


Fig. 3.2 Block diagram of TM4C123GH6PM MCU. *Courtesy* Texas Instruments

A block diagram of the TM4C123GH6PM MCU is shown in Fig. 3.2.

In this book, we need to use TM4C123GXL EVB that is composed of two MCUs with the same type, TM4C123GH6PM, and GPIO ports or blocks to interface some peripherals, such as QEI modules and PWM module, to drive DC motors and get feedback signals. Therefore in this chapter, we will concentrate on these components. We will discuss these in the following sequence:

- TM4C123GH6PM MCU on-chip memory
- TM4C123GH6PM GPIO Ports

As for the QEI and PWM modules, we will discuss them in Chap. 4 because of the space limitation in this chapter. First let us take care of the on-chip memory map.

3.2.1.1 TM4C123GH6PM Microcontroller On-Chip Memory Map

The TM4C123GH6PM MCU contains the following memory devices on this chip:

- 256 KB Flash ROM memory
- 32 KB SRAM
- 2 KB EEPROM
- An internal ROM

A detailed memory map for TM4C123GH6PM MCU is shown in Fig. 3.3.

All on-chip memory devices are controlled by the related control registers, such as Flash Control registers, ROM Control registers, SRAM Control registers, and EEPROM Control registers. These registers are located in the associated memory spaces.

The **256 KB Flash ROM** Memory is used to store the user's program codes and exception vector tables. The exception and interrupt vector tables are located at the lower memory space starting from 0x0000.0000. To perform any programming for this flash memory, the Tiva™ C Series devices provide a user-friendly interface with three registers. All erase or program operations are handled via these three registers: **Flash Memory Address (FMA)**, **Flash Memory Data (FMD)**, and **Flash Memory Control (FMC)**.

The **2 KB EEPROM** module provides a well-defined register interface to support accesses to the EEPROM with both a random access style of read and write as well as a rolling or sequential access scheme.

The **Internal ROM** is a new memory device in TM4C123GH6PM MCU and this unit can be preprogrammed with the following software and programs:

- TivaWare™ Driver Libraries, including the Peripheral Devices Library, USB Library, Graphical Library, and Sensor Hub Library.
- TivaWare™ Boot Loader

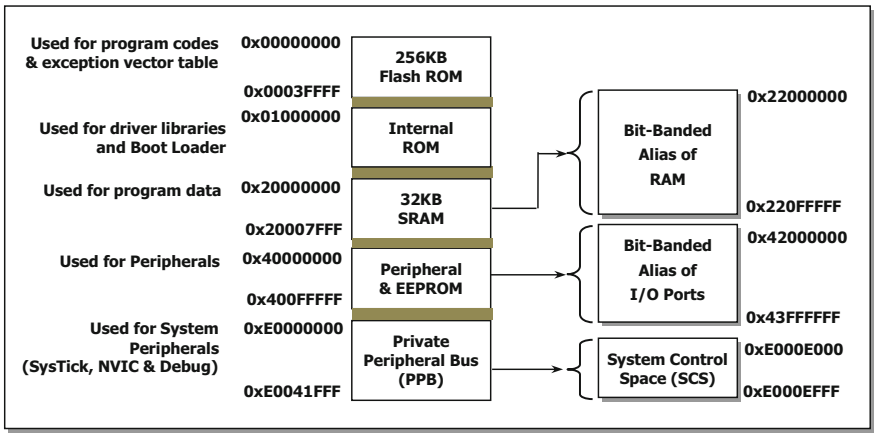


Fig. 3.3 Memory map for TM4C123GH6PM MCU

- Advanced Encryption Standard (AES) cryptography tables.
- Cyclic Redundancy Check (CRC) error-detection functionality.

The TivaWare™ Boot Loader is used to download code to the Flash memory of a device without the use of a debug interface. When the core is reset, the user has the opportunity to direct the core to execute the ROM Boot Loader or the application in Flash memory by using any GPIO signal in Ports A-H as configured in the **Boot Configuration (BOOTCFG)** register.

AES is ideal for applications that can use prearranged keys, such as setup during manufacturing or configuration.

The CRC technique can be used to validate correct receipt of messages (nothing lost or modified in transit), to validate data after decompression, to validate that Flash memory contents have not been changed, and for other cases where the data needs to be validated.

The **Peripheral** area is used to map all on-chip I/O devices and interfaces to external I/O devices to be used in the system. The main on-chip peripheral devices include

- Watchdog Timers
- Six 16/32-bit Timers and six 32/64 Timers
- Two Analog to Digital Converters (ADCs)
- Analog Comparators

The main peripheral interfaces used for the external I/O devices include

- General-Purpose Input Output (GPIO) Ports A ~ F.
- Synchronous Serial Interfaces (SSI), SSI0 ~ SSI3.
- UART0 ~ URAT7
- CAN0 ~ CAN1
- USB

One point to be noted is that each GPIO Port can be accessed through one of the two bus apertures: (1) The Advanced Peripheral Bus (APB) is backward-compatible with previous devices, (2) The Advanced High-Performance Bus (AHB). The AHB offers the same register map but provides better back-to-back access performance than the APB bus. Note that each GPIO module clock must be enabled before the registers can be programmed. There must be a delay of three system clocks after the GPIO module clock is enabled before any GPIO registers are accessed.

A more detailed memory map for TM4C123GH6PM MCU is shown in Table 3.1.

Basically, all peripherals in the TM4C123GH6PM can be divided into the following groups:

(1) System Peripherals
(2) On-Chip Peripherals
(3) Interfaces to External Parallel Peripherals
(4) Interfaces to External Serial Peripherals

Table 3.1 Memory map for TM4C123GH6PM MCU

Start address	End address	Descriptions
256 KB On-Chip Flash ROM		
0x0000.0000	0x0000.003C	Exceptions Vector Table
0x0000.0040	0x0000.00A8	Interrupts Vector Table
0x0000.00B0	0x0000.0268	System Controls
0x0002.0000	0x0003.FFFF	User Codes
Internal ROM		
0x0100.0000	0x01FF.FFFF	Tiva Driver Libraries, Boot Loader, AES and CRC
32 KB SRAM		
0x2000.0000	0x2000.7FFF	User Data
0x2200.0000	0x23FF.FFFF	Bit-band alias of bit-banded on-chip SRAM starting at 0x2000.0000
Peripherals		
0x4000.0000	0x4000.0FFF	Watchdog Timer 0
0x4000.1000	0x4000.1FFF	Watchdog Timer 1
0x4000.4000	0x4000.4FFF	GPIO Port A (APB Aperture)
0x4000.5000	0x4000.5FFF	GPIO Port B (APB Aperture)
0x4000.6000	0x4000.6FFF	GPIO Port C (APB Aperture)
0x4000.7000	0x4000.7FFF	GPIO Port D (APB Aperture)
0x4000.8000	0x4000.BFFF	SSI0 ~ SSI3
0x4000.C000	0x4001.3FFF	UART0 ~ UART7
0x4002.0000	0x4002.3FFF	$I^2C0 \sim I^2C3$
0x4002.4000	0x4002.4FFF	GPIO Port E (APB Aperture)
0x4002.5000	0x4002.5FFF	GPIO Port F (APB Aperture)
0x4002.8000	0x4002.9FFF	PWM0 ~ PWM1
0x4002.C000	0x4002.DFFF	QEI0 ~ QEI1
0x4003.0000	0x4003.5FFF	16/32-Bit Timer 0 ~ 16/32-Bit Timer 5
0x4003.6000	0x4003.7FFF	32/64-Bit Timer 0 ~ 32/64-Bit Timer 1
0x4003.8000	0x4003.8FFF	ADC0
0x4003.9000	0x4003.9FFF	ADC1
0x4003.C000	0x4003.CFFF	Analog Comparators
0x4004.0000	0x4004.1FFF	CAN0 ~ CAN1 Controllers
0x4004.C000	0x4003.FFFF	32/64-Bit Timer 2 ~ 32/64-Bit Timer 5
0x4005.0000	0x4005.0FFF	USB
0x4005.8000	0x4005.DFFF	GPIO Port A ~ Port F (AHB Aperture)
0x400A.F000	0x400A.FFFF	2 KB EEPROM and Key locker

(continued)

Table 3.1 (continued)

Start address	End address	Descriptions
0x400F.9000	0x400F.9FFF	System Exception Module
0x400F.C000	0x400F.CFFF	Hibernation Module
0x400F.D000	0x400F.DFFF	Flash Memory Control
0x400F.E000	0x400F.EFFF	System Control
0x400F.F000	0x400F.FFFF	µDMA
0x4200.0000	0x43FF.FFFF	Bit-banded alias of 0x4000.0000 through 0x400F.FFFF
Private Peripheral Bus		
0xE000.0000	0xE000.0FFF	Instrumentation Trace Macrocell (ITM)
0xE000.1000	0xE000.1FFF	Data Watchpoint and Trace (DWT)
0xE000.2000	0xE000.2FFF	Flash Patch and Breakpoint (FPB)
0xE000.E000	0xE000.EFFF	Cortex-M4F System Peripherals (SysTick, NVIC, MPU, FPU and SCB)
0xE004.0000	0xE004.0FFF	Trace Port Interface Unit (TPIU)
0xE004.1000	0xE004.1FFF	Embedded Trace Macrocell (ETM)

For the interfaces to external serial peripherals, they can be further categorized into another two subgroups; synchronous and asynchronous communication mode.

The System Peripherals

The **System Peripherals** are related controls to the system peripherals, and the most popular system peripherals involved in this MCU are

- System Timer (SysTick)
- Two Watchdog Timers
- Six 16/32-bit and Six 32/64 Timers
- Instrumentation Trace Macrocell (ITM)
- Data Watchpoint and Trace (DWT)
- Flash Patch and Breakpoint (FPB)
- Trace Port Interface Unit (TPIU)

The On-Chip Peripherals

The **On-Chip Peripherals** are controls or components integrated on the TM4C123GH6PM chip, which include

- Two 12-bit Analog to Digital Converters (ADCs)
- Three Analog Comparators
- One Voltage Regulator
- One Temperature Sensor
- Two PWM Modules (PWM0 and PWM1) with total of 16 PWM Outputs
- Two QEI Modules (QEI0 and QEI1) providing control of two motors at the same time

Interfaces to External Parallel Peripherals

The TM4C123GH6PM MCU provides a General Purpose Input and Output (GPIO) module. This module contains six GPIO blocks, each block is related to an individual GPIO Port. Each GPIO Port is programmable or configurable to provide multiple functions to enable the port to handle different tasks. For example, each port can work either as input or output port, or either as parallel or serial port. Furthermore, each bit or pin on each port can be programmed separately to perform desired functions, such as an input–bit, an output bit, a parallel bit or a serial bit. Also, each bit or pin on each GPIO Port can be configured as an interrupt source to create an associated interrupt request to the Cortex®-M4 core. These interrupt requests can be configured to be triggered by either a rising or falling edge, or by voltage levels.

Six GPIO blocks are ranged from Port A through Port F with 8-pin for each port. Two kinds of accessing modes can be used to access each GPIO Port in the TM4C123GH6PM MCU: via the Advanced Peripheral Bus (APB) or via the Advanced High-Performance Bus (AHB). The latter provides better performances than the former.

Interfaces to External Serial Peripherals

The TM4C123GH6PM MCU supports both asynchronous and synchronous serial communications with

- Two CAN 2.0 A/B controllers
- One USB 2.0 OTG/Host/Device
- Eight UARTs with IrDA, 9-bit and ISO 7816 support.
- Four I^2C modules with four transmission speeds including high-speed mode
- Four Synchronous Serial Interface modules (SSI)

In fact, all interfaces to external peripherals, including to the external parallel or serial interfaces, are physically executed via the GPIO ports in the TM4C123GH6PM MCU. There is no extra interface to perform either parallel or serial peripheral jobs separately.

Because the GPIO play such a vital role in the interfaces to all peripheral devices, now let us take a closer look at these ports and related interfaces.

3.2.1.2 TM4C123GH6PM Microcontroller GPIO Module

As we mentioned, the GPIO module is composed of six physical GPIO blocks, each block is corresponding to an individual GPIO port (Port A ~ Port F). Each GPIO port can be configured to perform a special function, such as input or output with interrupt property. Also even each pin on the selected GPIO port can be configured to work as an input or output pin. However, not all 48 pins (6 GPIO Ports with each port having 8 pins) are configurable or programmable since some pins are not available. Therefore totally the GPIO module supports up to 43 programmable input/output pins, depending on the peripherals being used.

One very important point is that each GPIO module performs its works based on an independent clock source, which is a timing base to make each GPIO works properly. In order to make GPIO to work, the related registers in the GPIO module must be initialized or programmed. The prerequisite for programming any GPIO module is that each GPIO module clock must be enabled before the registers can be programmed. There must be a delay of three system clocks after the GPIO module clock is enabled before any GPIO registers are accessed.

Before we can continue to discuss the GPIO programming process, let us first take a look at the system clock and GPIO module clock.

The System Clock

As we know, the clock is a timing base and it provides an operational timing standard or criterion to enable computers to perform their jobs step by step based on each clock cycle. Similarly, in order to enable microcontrollers to execute their instructions in a definite sequence, a clock source is definitely needed. Without clock source, no any microcontroller or computer can run their instructions properly.

In TM4C123GH6PM MCU, four different clock sources are provided (Fig. 3.4)

- Precision Internal Oscillator (PIOSC): 16 MHz.
- Main Oscillator (MOSC): It can use an external clock source or an external crystal.
- Low-Frequency Internal Oscillator (LFIOSC): An on-chip internal 30 kHz Oscillator used for Deep-Sleep power-saving modes.
- Hibernate RTC Oscillator (RTCOSC) Clock Source: It can be configured to be the 32.768 kHz external oscillator source from the Hibernation (HIB) module or the HIB Low Frequency clock source (HIB LFIOSC), which is intended to provide the system with a real-time clock source.

The ARM® Cortex®-M4 Core or processor can be driven by

(1) Any clock source shown above. The CPU can also use a 4 MHz clock that is the internal 16 MHz oscillator divided by 4 (Fig. 3.4).
(2) Phase Locked Loop (PLL) clock generator.

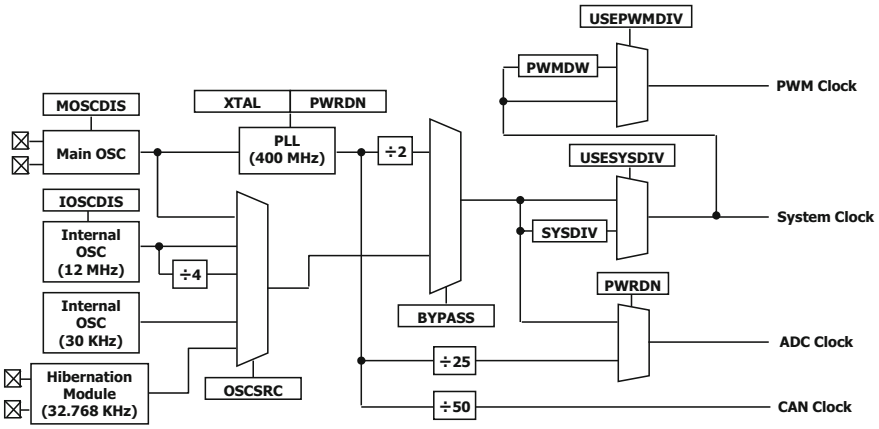


Fig. 3.4 Four clock sources and system clock

Refer to Fig. 3.4, two multiplexers (MUXs) are used to select different clock sources and two ways can be used to create a system clock to be used by the CPU.

One way is to use the Phase Locked Loop (PLL) clock generator that needs a clock source as the input source to create this system clock. It can be found from Fig. 3.4 that two clock sources, Main OSC and Precision Internal OSC (16 MHz), can work as a clock source for the PLL and this source can be selected via the MUX in the top path.

Another way is to directly use any one of four clock sources and this can be selected via a MUX in the bottom path. An easy way is to use Precision Internal OSC (16 MHz) divided by 4 to get a 4 MHz system clock.

Now let us have a closer look at Fig. 3.4 to get more details about the system clock generation process.

- In the top path, the system clock is generated by a PLL that can be driven by any crystals or oscillators running between 5 and 25 MHz. The output frequency of the PLL is always 400 MHz and it is independent on the input clock sources. In Fig. 3.4, a 16 MHz crystal on the Main Oscillator (MOSC) is selected to drive the PLL (dark line). After a default divided by 2 divider, the system clock should be 200 MHz.
- In the bottom path, the system clock can be generated by any clock source we discussed above. The selected clock source can avoid the SYSDIV and USESYSDIV dividing operations via BYPASS for both MUXs and can be directly sent out as the system clock.

When using the PLL, the output frequency of 400 MHz is pre-divided by 2 (becomes 200 MHz) before the other user's divisor is applied. The users can modify this 200 MHz system clock by adding different dividing factors in the **SYSCTL_SYSDIV_X** (SYSDIV) in their program to use a lower frequency system clock. The

X is an integer for the desired factor. For example, if a **SYSCTL_SYSDIV_5** is used, the system clock would be 400 MHz/2/5 = 400 MHz/10 = 40 MHz.

Two registers, **Run-Mode Clock Configuration (RCC)** register and **Run-Mode Clock Configuration 2 (RCC2)** register, provide controls for the system clock. The RCC2 register is used to provide additional control parameters that offer additional encodings over the RCC register. When used, the RCC2 register field values are used by the logic over the corresponding field in the RCC register. In particular, RCC2 provides a larger assortment of clock configuration options. These registers control the following clock functionality

- Source of clocks in sleep and deep-sleep modes
- System clock derived from PLL or other clock source
- Enable or disable the oscillators and PLL
- Clock divisors
- Crystal input selection

A point to be noted is that during the clock configuration process, always write the RCC register prior to writing the RCC2 register

In the RCC register, the **SYSDIV** field specifies which divisor is used to generate the system clock from either the PLL output or the oscillator source. When using the PLL, the VCO frequency of 400 MHz is pre-divided by 2 before the divisor is applied.

The General Configuration Procedures for GPIO Peripherals

Because the GPIO module provides various interface supports to meet the needs of multiple different peripheral functions, including the parallels and serials, inputs and outputs, synchronous and asynchronous, and even bit-band input/output operations, the configuration of this module should be very complicated compared with other simple peripheral interfaces. The more functions the device provided, the more complicated the device configurations, which is also true to GPIO.

Regularly, to enable GPIO in the Cortex®-M4 system to work properly, the following initialization processes are needed:

(1) Enable the system clock to be connected to the selected GPIO peripheral, exactly to connect the clock signal source to the corresponding peripheral device I/O pins. Typically the clock signal is disconnected to any peripheral after the system is reset by default. You need to enable this clock connection before you can program the desired peripheral. You also need to enable the clock to the peripheral bus system.
(2) Configure the operational mode of the selected peripheral I/O pins, such as input/output, parallel/serial, or special function. As we mentioned, the GPIO in the Cortex®-M4 system provides various modes with multiplexed I/O pins. For example, one I/O pin on Port A (PA0) provides either parallel or serial, either

input or output functions. Therefore you need program-related configuration registers to setup and configure each pin to meet your requirement.

(3) Configure the interrupt control registers on NVIC to enable the interrupt and define the priority level for the desired interrupt if the interrupt function is used for the peripheral.

(4) Include and use the related peripheral driver library in your program to make your program development process easier. Most vendors provide device driver library code to facilitate the developers coding jobs.

For the GPIO module used in the Cortex®-M4 MCU, the similar configuration procedures should be used to initialize all ports and pins before they can be used.

Now let us have a clear picture about the architecture and organization of the GPIO used in the TM4C123GH6PM MCU.

Tiva™ TM4C123GH6PM GPIO Architecture

Figure 3.5 shows an analog/digital functional block diagram of the GPIO module used in the TM4C123GH6PM MCU. It can be found that the GPIO module is controlled by seven (7) controls with several control registers involved in each control.

Six GPIO blocks, each can be mapped to a GPIO Port, provide six GPIO Ports, from Port A to Port F, with 8 bits mapping to 8 I/O pins for each port. The 8-pin mode is used to try to compatible with the old I/O interface configurations.

Although each control register is 32-bit, only the lowest 8-bit is used to provide a control byte function. Depends on two different I/O buses accessing modes, APB or AHB, six GPIO Ports are available and can be accessed by these two different accessing buses. Because of these two different accessing buses, these six GPIO Ports have different memory map addresses. Refer to Table 3.1 to get detailed address ranges for these six GPIO Ports.

All GPIO Ports can be configured by programming their related registers, and each port has several registers used to initialize, configure, and control the functions of each port and each pin.

Each register has a unique address, even each pin or bit has an aliased address (like bit-band), in the GPIO memory map.

Refer to Fig. 3.5, seven controls in all GPIO blocks are

(1) **Port Control**: Used to select the operational mode for the selected GPIO port, either GPIO mode or Peripheral mode, via a multiplexing selector MUX.

(2) **Commit Control**: Used to enable or disable other four registers' operational bits. In other words, the GPIOCR register is to control the transferring ability of each bit on other four registers. The GPIOLOCK register is used to lock (disable) or unlock (enable) the GPIOCR register.

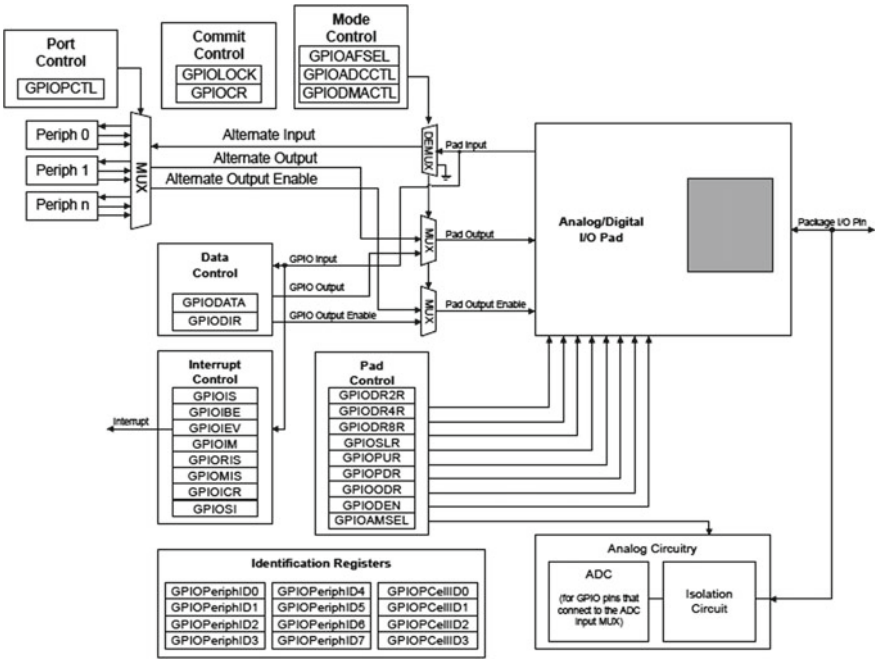


Fig. 3.5 Function block diagram of Analog/Digital GPIO control. *Courtesy* Texas Instruments

(3) **Mode Control**: The GPIOAFSEL register works together with GPIOPCTL to determine the selected GPIO port mode. Another two registers are used to setup either the ADC mode or DMA mode for the selected GPIO port.
(4) **Data Control**: Used to control the data modification ability and data transfer direction.
(5) **Interrupt Control**: Determine all interrupt properties for the selected GPIO port or bits, such as interrupt triggering method, edges or levels, interrupt enabled or disabled, interrupt masking, interrupt status, and interrupt clear function.
(6) **Pad Control**: Enable software to configure the GPIO pads based on the application requirements. The pad control includes nine (9) registers. These registers control drive strength, open-drain configuration, pull-up, and pull-down resistors, slew-rate control and digital input enable for each GPIO.
(7) **Identification Registers**: These registers configured at reset enable software to detect and identify the module as a GPIO block. The identification registers include twelve (12) related registers.

Although there are over 30 control registers available for each port, basically the following control registers are most popular and widely utilized for programming of GPIO Ports

A. **The Port Control Register (GPIOPCTL)**

This register is used to work together with the **GPIOAFSEL** register to select the specific peripheral signal for each GPIO pin when using the alternate function mode. Most bits in the **GPIOAFSEL** register are 0 on reset; therefore most GPIO pins are initialized to GPIOs mode by default. Unlike other control registers, the **GPIOPCTL** uses all 32-bit to define peripherals for all eight pins. This 32-bit is divided into eight segments, each segment is 4-bit and contains a Port Mux Control (**PMC**) code, from PMC0 to PMC7, and each 4-bit PMC code selects peripherals for each pin. Refer to Table 3.2 to get detailed information about the PMC values in the GPIOPCTL register and related peripheral functions.

Bit Value If a **GPIOAFSEL** bit = 0: The corresponding GPIO pin works in GPIO mode
If a **GPIOAFSEL** bit = 1: The corresponding GPIO pin is controlled by certain peripherals. The **GPIOPCTL** register selects one peripheral function for each GPIO.

B. **The Data Control Registers**

Two registers, **Data Control Register (GPIODATA)** and **Data Direction Register (GPIODIR)** are included in this group. These registers are used to configure the operational modes of the GPIOs, which include

- **Data Control Register (GPIODATA)**: The GPIO ports allow for the modification of individual bits in the GPIODATA register by using bits [9:2] of the address bus as a mask. In this manner, software drivers can modify individual GPIO pins in a single instruction without affecting the state of the other pins.
- **Data Direction Register (GPIODIR)**: Configure each port or each individual pin as an input or output. If a bit = 0, the associated bit in the GPIODATA is an input bit. If a bit = 1, the associated bit in the GPIODATA is an output bit.

C. **The Mode Control Registers**

Three registers are involved in this group: **GPIO Alternate Function Select Register (GPIOAFSEL)**, **GPIO ADC Control Register (GPIOADCCTL)**, and **GPIO DMA Control Register (GPIODMACTL).** These registers provide a protection layer against accidental programming of critical hardware peripherals, which include

- **GPIO Alternate Function Select Register (GPIOAFSEL)**: This register is the mode control selection register. If a bit = 0, the corresponding pin is used as a GPIO and is controlled by the GPIO registers (software). If a bit = 1, the corresponding pin is controlled by an associated peripheral (hardware). This register can work together with the GPIOPCTL register to enable users to select one of several peripheral functions for each GPIO when a bit in this register is set to 1. The reset value for this register is 0x0000.0000 for ports.

Table 3.2 GPIO pins and alternate functions

I/O	Pin	Analog function	Digital functions (GPIOPCTL PMCx bit field encoding)									
			1	2	3	4	5	6	7	8	9	14
PA0	17	–	U0RX	–	–	–	–	–	–	CAN1RX	–	–
PA1	18	–	U0TX	–	–	–	–	–	–	CAN1TX	–	–
PA2	19	–	–	SSI0CLK	–	–	–	–	–	–	–	–
PA3	20	–	–	SSI0FSS	–	–	–	–	–	–	–	–
PA4	21	–	–	SSI0RX	–	–	–	–	–	–	–	–
PA5	22	–	–	SSI0TX	–	–	–	–	–	–	–	–
PA6	23	–	–	–	I2C1SCL	–	M1PWM2	–	–	–	–	–
PA7	24	–	–	–	I2C1SDC	–	M1PWM3	–	–	–	–	–
PB0	45	USB0ID	U1RX	–	–	–	–	–	T2CCP0	–	–	–
PB1	46	USB0VBUS	U1TX	–	–	–	–	–	T2CCP1	–	–	–
PB2	47	–	–	–	I2C0SCL	–	–	–	T3CCP0	–	–	–
PB3	48	–	–	–	I2C0SDC	–	–	–	T3CCP1	–	–	–
PB4	58	AIN10	–	SSI2CLK	–	M0PWM2	–	–	T1CCP0	CAN0RX	–	–
PB5	57	AIN11	–	SSI2FSS	–	M0PWM3	–	–	T1CCP1	CAN0TX	–	–
PB6	1	–	–	SSI2RX	–	M0PWM0	–	–	T0CCP0	–	–	–
PB7	4	–	–	SSI2TX	–	M0PWM1	–	–	T0CCP1	–	–	–
PC0	52	–	TCK SWCLK	–	–	–	–	–	T4CCP0	–	–	–
PC1	51	–	TMS SWDIO	–	–	–	–	–	T4CCP1	–	–	–
PC2	50	–	TDI	–	–	–	–	–	T5CCP0	–	–	–
PC3	49	–	TDO SWO	–	–	–	–	–	T5CCP1	–	–	–
PC4	16	C1-	U4RX	U1RX	–	M0PWM6	–	IDX1	WT0CCP0	U1RTS	–	–

(continued)

Table 3.2 (continued)

I/O	Pin	Analog function	Digital functions (GPIOPCTL PMCx bit field encoding)									
			1	2	3	4	5	6	7	8	9	14
PC5	15	C1+	U4TX	U1TX	–	M0PWM7	–	PHA1	WT0CCP1	U1CTS	–	–
PC6	14	C0+	U3RX	–	–	–	–	PHB1	WT1CCP0	USB0EPEN	–	–
PC7	13	C0-	U3TX	–	–	–	–	–	WT1CCP1	USB0PFLT	–	–
PD0	61	AIN7	SSI3CLK	SSI1CLK	I2C3SCL	M0PWM6	M1PWM0	–	WT2CCP0	–	–	–
PD1	62	AIN6	SSI3FSS	SSI1FSS	I2C3SDC	M0PWM7	M1PWM1	–	WT2CCP1	–	–	–
PD2	63	AIN5	SSI3RX	SSI1RX	–	M0FAULT0	–	–	WT3CCP0	USB0EPEN	–	–
PD3	64	AIN4	SSI3TX	SSI1TX	–	–	–	IDX0	WT3CCP1	USB0PFLT	–	–
PD4	43	USB0DM	U6RX	–	–	–	–	–	WT4CCP0	–	–	–
PD5	44	USB0DP	U6TX	–	–	–	–	–	WT4CCP1	–	–	–
PD6	53	–	U2RX	–	–	M0FAULT0	–	PHA0	WT5CCP0	–	–	–
PD7	10	–	U2TX	–	–	–	–	PHB0	WT5CCP1	NMI	–	–
PE0	9	AIN3	U7RX	–	–	–	–	–	–	–	–	–
PE1	8	AIN2	U7TX	–	–	–	–	–	–	–	–	–
PE2	7	AIN1	–	–	–	–	–	–	–	–	–	–
PE3	6	AIN0	–	–	–	–	–	–	–	–	–	–
PE4	59	AIN9	U5RX	–	I2C2SCL	M0PWM4	M1PWM2	–	–	CAN0RX	–	–
PE5	60	AIN8	U5TX	–	I2C2SDC	M0PWM5	M1PWM3	–	–	CAN0TX	–	–
PF0	28	–	U1RTS	SSI1RX	CAN0RX	–	M1PWM4	PHA0	T0CCP0	NMI	C0O	–
PF1	29	–	U1CTS	SSI1TX	–	–	M1PWM5	PHB0	T0CCP1	–	C1O	TRD1
PF2	30	–	–	SSI1CLK	–	M0FAULT0	M1PWM6	–	T1CCP0	–	–	TRD0
PF3	31	–	–	SSI1FSS	CAN0TX	–	M1PWM7	–	T1CCP1	–	–	TRCLK
PF4	5	–	–	–	–	–	M1FAULT0	IDX0	T2CCP0	USB0EPEN	–	–

- **GPIO ADC Control Register (GPIOADCCTL)**: Configure a GPIO pin as a source for the ADC input. If the GPIOADCCTL Register in Port B is cleared, PB4 can still be used as an external trigger for the ADC.

  Bit Value 0: The corresponding pin is not used to trigger the ADC
  1: The corresponding pin is used to trigger the ADC.

- **GPIO DMA Control Register (GPIODMACTL)**: Configure a GPIO pin as a source for the μDMA trigger.

  Bit Value 0: The corresponding pin is not used to trigger the μDMA
  1: The corresponding pin is used to trigger the μDMA.

D. **The Commit Control Registers**

These two registers, GPIO Lock Register (GPIOLOCK) and GPIO Commit Register (GPIOCR), are used to provide commit controls to each bit for other four registers.

- **GPIO Lock Register (GPIOLOCK)**: Control writes access to the GPIOCR register. If the GPIOLOCK is unlocked (writing **0x4C4F434B** to GPIOLOCK), the contents on GPIOCR can be modified by writing. Otherwise, if this register is locked (writing any other number to GPIOLOCK), any writing to the GPIOCR is ignored and its contents cannot be changed.
- **GPIO Commit Register (GPIOCR)**: Control (enable or disable) each bit on the other four registers: GPIOAFSEL, GPIOPUR, GPIOPDR, and GPIODEN. The value of the GPIOCR register determines which bits of those four registers are committed when a write to these registers is performed.
  - If a bit in the **GPIOCR** = 0, the data being written to the corresponding bit in those registers cannot be committed or disabled and retains its previous value.
  - If a bit in the **GPIOCR** = 1, the data being written to the corresponding bit of those registers is committed or enabled to the register and reflects the new value.

The contents of the GPIOCR register can only be modified if the status in the GPIOLOCK register is unlocked.

E. **The Interrupt Control Registers**

This group contains seven (7) related registers and used to provide interrupt controllability for each GPIO Port. The control function for each register is

- **GPIO Interrupt Sense Register (GPIOIS)**: Determine the triggering mode of an interrupt source. All bits are cleared by a reset.

  Bit Value 0: The corresponding pin in the selected GPIO port will detect edges
  1: The corresponding pin in the selected GPIO port will detect levels.

- **GPIO Interrupt Both Edges Register (GPIOIBE)**: Allows both edges to cause interrupts. When a bit in the GPIOIS = 0, if the corresponding bit in the GPIOIBE = 1, the corresponding pin in the selected port detects both rising and falling edges, regardless of the corresponding bit in the GPIOIEV register. If the corresponding bit in the GPIOIBE = 0, the corresponding pin in the selected port will be controlled by the GPIOIEV register. All bits are cleared by a reset.
- **GPIO Interrupt Event Register (GPIOIEV)**: If a bit = 1, the corresponding pin in the selected port detects rising edge (when the corresponding bit in the GPIOIS = 0) or high levels (when the corresponding bit in the GPIOIS = 1). If a bit = 0, the corresponding pin in the selected port detects falling edge (when the corresponding bit in the GPIOIS = 0) or low levels (when the corresponding bit in the GPIOIS = 1). All bits are cleared by a reset.
- **GPIO Interrupt Mask Register (GPIOIM)**: Mask (disable) or unmask (enable) an interrupt generated by the corresponding pin to be transferred to the interrupt controller NVIC. All bits are cleared by a reset.

  Bit Value 0: Disable an interrupt generated by the corresponding pin to be sent to the interrupt controller NVIC
  1: Enable an interrupt generated by the corresponding pin to be sent to the interrupt controller NVIC.

- **GPIO Raw Interrupt Status Register (GPIORIS)**: Indicate the raw interrupt status for a specified bit. When an interrupt condition occurs on a GPIO pin, the corresponding bit in this register is set to 1. If the corresponding bit in the GPIOIM = 1, the interrupt is sent to the interrupt controller. Bits read as zero indicate that corresponding input pins have not initiated an interrupt. For edge-detect interrupts, this bit is cleared by writing a 1 to the corresponding bit in the GPIOICR register. For a GPIO level-detect interrupt, the bit is cleared when the level is deasserted.
- **GPIO Masked Interrupt Status Register (GPIOMIS)**: Indicate the state of the interrupt after masking. If a bit in this register = 1, the corresponding interrupt has triggered an interrupt to the interrupt controller. If a bit in this register = 0, either no interrupt has been generated or the interrupt is masked.
- **GPIO Interrupt Clear Register (GPIOICR)**: For edge-detect interrupts, writing a 1 to the IC bit in the GPIOICR register clears the corresponding bit in the GPIORIS and GPIOMIS registers. If the interrupt is a level-detect, the IC bit in this register has no effect. In addition, writing a 0 to any of the bits in the GPIOICR register has no effect.

F. **The Pad Control Registers**

Nine (9) control registers are involved in the Pad Control group. These registers control drive strength, open-drain configuration, pull-up and pull-down resistors, slew-rate control and digital input enable for each GPIO. The function of each register is

- **GPIO 2-mA Drive Select Register (GPIODR2R)**:
- **GPIO 4-mA Drive Select Register (GPIODR4R)**:
- **GPIO 8-mA Drive Select Register (GPIODR8R)**:

These three registers are used to select the drive strength (from 2, 4 to 8 mA) for selected pin on the GPIO port. The GPIODR2R is for 2-mA, GPIODR4R is for 4-mA and GPIODR8R is for 8-mA drive current.

Each GPIO pin in the port can be configured individually without affecting the other pads. As we mentioned, although all control registers are 32-bit, only the lowest 8-bit (1-byte) is used to provide the control function. When setting the DRV2 byte, which is the lowest 8-bit in the GPIODR2R register, the corresponding DRV4 byte in the GPIODR4R register and DRV8 byte in the GPIODR8R register are automatically cleared by hardware. The same thing would have happened to GPIODR4R and GPIODR8R, which means that only one drive strength can be selected for all of these three registers at a time. If a drive strength is selected in one register, the other two registers will be reset to 0 to disable other drive strength selections.

By default, all GPIO pins have 2-mA drive.

- **GPIO Open-Drain Select Register (GPIOODR)**: This register is used to configure the open-drain mode for each bit in the selected GPIO port. The lowest 8-bit, with each bit to each pin, or called **Open-Drain Enable (ODE)** byte in this register can be used to configure whether a bit or a pin for the GPIO port works in the open-drain mode or not. The open-drain mode enables the selected bit to output in a high-impendence status to drive more loads. When the open-drain mode is set, the corresponding bit should also be set in the GPIO Digital Enable Register (GPIODEN).

  Bit Value 0: Disable the corresponding pin to work in the open-drain mode
  1: Enable the corresponding pin to work in the open-drain mode.

- **GPIO Pull-Up Select Register (GPIOPUR)**:
- **GPIO Pull-Down Select Register (GPIOPDR)**:

These two registers are used to select either Pull-Up or Pull-Down resistor on the output of a pin. Figure 3.6 shows an illustration for pull-up and pull-down connections. These modes only work for output pins and have no effect if a pin works as an input one.

The resistor R1 is called a pull-up resistor and R2 is a pull-down resistor. When a bit in the GPIOPUR is set to 1 (a Pull-Up is selected), the SW1 is closed but the SW2 is open to connect R1 to the voltage source V_{DD} to enable the output of the corresponding pin is HIGH when no load is applied. Otherwise, if a bit in the GPIOPDR is set to 1 (a Pull-Down is selected), the SW2 is closed but the SW1 is open to connect R2 to the output to enable the output of the corresponding pin is LOW when no load is applied.

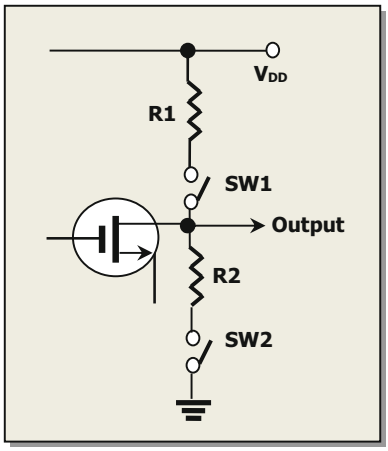


Fig. 3.6 Pull-up and pull-down mode

The real control bits are lowest 8-bit for two registers, **Pull-Up Enable (PUE)** byte and **Pull-Down Enable (PDE)** byte. Setting a bit to 1 in the GPIOPUR register clears the corresponding bit in the GPIOPDR register, and vice versa. *Both registers are reset to 0 after the reset*, which means that both resistors are not connected to any output pin after a reset operation. This means that after a system reset, most GPIO pins are in high-impendence (HZ) or a tri-state status.

- **GPIO Slew-Rate Control Select Register (GPIOSLR)**: The **GPIOSLR** register is the slew-rate control register. Slew-rate control is only available when using the 8-mA drive strength option setup in the GPIODR8R register. The real control bits are lowest 8-bit on 32-bit GPIOSLR register, which is called **SLR** byte.

  Bit Value 0: Disable the slew-rate control for the corresponding pin
  1: Enable the slew-rate control for the corresponding pin.

- GPIO **Digital Enable Register (GPIODEN)**: The GPIODEN register is the digital enable register. By default, most GPIO pins are configured to work as the GPIO mode with tri-state or undriven status. In the tri-state, the digital functions are disabled, which means that they do not output a logic value on the pin and they do not allow the pin to receive any voltage signal into the GPIO receiver. The actual control bits are lowest 8-bit or called **DEN** byte. However, some special consideration pins may be programmed to a non-GPIO function or may have special commit controls after a reset. All lowest 8-bit or DEN byte are cleared to 0 after a system reset, which means that the digital functions for all GPIO pins are disabled.

  Bit Value 0: The digital functions for the corresponding pin are disabled
  1: Enable a corresponding pin to work as a digital input or output.

- **GPIO Analog Mode Select Register (GPIOAMSEL)**: This register is used to enable or disable the analog input isolation status for the selected GPIO bits/pins. This register is only valid for ports and pins that can be used as ADC AINx inputs. Since the GPIOs may be driven by a 5 V source and affect analog operation, analog circuitry requires isolation from the pins when they are not used in their analog function. If any pin is to be used as an ADC input, the related bit in GPIOAMSEL must be set to 1 to disable the analog isolation circuit and enable the analog input function. Otherwise, if a bit in the GPIOAMSEL is 0, the analog isolation circuit is enabled and the analog input function is disabled for the corresponding pin.

  Bit Value 0: Enable the analog isolation circuit and disable the analog input function when a pin is not used for ADC
  1: Disable the analog isolation circuit and enable the analog input function when a pin is used for ADC.

Since the **GPIO Alternate Function Select Register (GPIOAFSEL)** needs to work with the **GPIO Port Control Register** (**GPIOPCTL**) together to determine the operational mode and related peripheral functions for the selected GPIO pins, we need to take a closer look at these two registers with more details about their functions.

Table 3.2 provides a clear picture about the working principle of these three components. Figure 3.7 shows the relationship between these components.

If a bit in the lowest 8-bit in the GPIOAFSEL is 0, this means that the GPIO mode has been selected for the corresponding pin, and the pin will be controlled by the GPIO registers to perform the default GPIO functions. This mode has nothing to do with the GPIOPCTL register.

However, if a bit in the lowest 8-bit in the GPIOAFSEL is set to 1, such as bit 0 and bit 6 shown in Fig. 3.7, the control for these two pins will be determined by the PMCx encoding bits value in Table 3.2. Since the GPIOPCTL is a 32-bit register and it has been divided into eight PMCx nibbles, each nibble has four (4) bits and

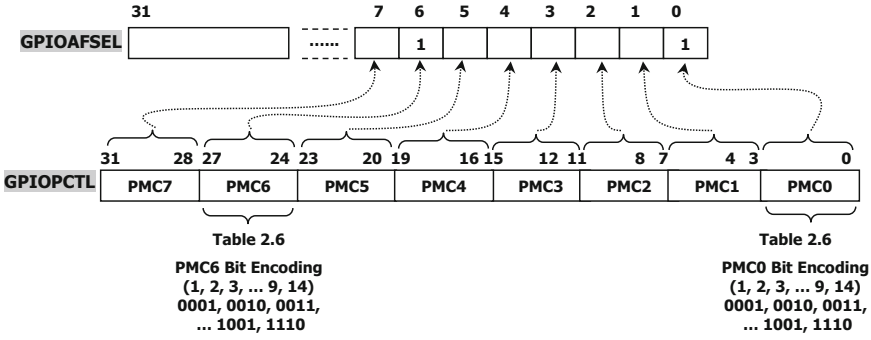


Fig. 3.7 The illustration for GPIOAFSEL and GPIOPCTL registers

used to control one pin selected by the bit in the GPIOAFSEL register. These nibbles have a one-to-one mapping relationship between the PMCx in the GPIOPCTL and bits in the GPIOAFSEL register. For example, PMC0 is for bit 0, PMC1 is for bit 1, and PMC7 is for bit 7 in the GPIOAFSEL register. Since bit 0 in the GPIOAFSEL is set to 1, if the nibble PMC0 or bits $3 \sim 0$ in the GPIOPCTL are 1 (0x0001), pin PA0 in GPIO Port A works as UART0 Receiver (U0RX). Similarly if bit 6 in the GPIOAFSEL is set to 1 and the PMC6 nibble is 5 (0x0101), then pin 6 in the Port A (PA6) works as a PWM generator (M1PWM2) to output a sequence of PWM signal.

G. **The Identification Registers**

There are eight (8) peripheral identification registers and four (4) primeCell identification registers used for GPIO Ports. The eight peripheral identification registers and four primeCell identification registers are

- **GPIO Peripheral Identification 0 (GPIOPeriphID0)~GPIO Peripheral Identification 7 (GPIOPeriphID7).**
- **GPIO PrimeCell Identification 0 (GPIOPCellID0)~GPIO PrimeCell Identification 3 (GPIOPCellID3).**

The main purpose of using eight peripheral identification registers is to identify the used peripherals by software. These eight registers can be divided into two groups:

- The GPIOPeriphID0, GPIOPeriphID1, GPIOPeriphID2, and GPIOPeriphID3 registers can conceptually be treated as one 32-bit register; each register contains eight bits of the 32-bit register.
- The GPIOPeriphID4, GPIOPeriphID5, GPIOPeriphID6, and GPIOPeriphID7 registers can conceptually be treated as one 32-bit register; each register contains eight bits of the 32-bit register

All registers uses their lowest 8-bit, from PID0 to PID7, to reserve the related peripheral identification code used to match the desired peripheral device. Four PrimeCell identification registers are used as standard cross-peripheral identification systems. All these registers are four 8-bit wide registers that can conceptually be treated as one 32-bit register.

The Initialization and Configuration of TM4C123GH6PM GPIO Ports

To access a GPIO Port or a pin, most time the full address of a GPIO port is divided into a combination of a Base address and an Offset. The Base address is the starting address of a port and the Offset is a distance or an interval of the related pin from the starting address.

The base addresses for six GPIO Ports or blocks are (APB and AHB apertures)

GPIO Port A (APB): 0x4000.4000	GPIO Port A (AHB): 0x4005.8000
GPIO Port B (APB): 0x4000.5000	GPIO Port B (AHB): 0x4005.9000
GPIO Port C (APB): 0x4000.6000	GPIO Port C (AHB): 0x4005.A000
GPIO Port D (APB): 0x4000.7000	GPIO Port D (AHB): 0x4005.B000
GPIO Port E (APB): 0x4002.4000	GPIO Port E (AHB): 0x4005.C000
GPIO Port F (APB): 0x4002.5000	GPIO Port F (AHB): 0x4005.D000

Each GPIO Port (Ports A ∼F) contains a set of all GPIO Registers discussed above and each register can be accessed by using an offset address combining (plus) with a base address.

For example, the Port A in the APB bus aperture has a base address 0x4000.4000. Its corresponding registers have different associate offset addresses. Table 3.3 shows a relationship between the base address, offset address, and full address for some Port A registers.

In Table 3.3, the symbolic definition of each related register in the last column is used for programming purpose. Users can use these definitions in their program to access each register.

To initialize and configure a specific GPIO Port, the following additional system control registers should be used:

Table 3.3 Some popular GPIO Port A registers in APB bus aperture

GPIO register	Base address	Offset	Full address	SW symbolic definition
GPIO Port A Data Register	0x4000.4000	0x000	0x4000.4000	GPIO_PORTA_DATA_R
GPIO Port A Direction Register	0x4000.4000	0x400	0x4000.4400	GPIO_PORTA_DIR_R
GPIO Port A AFSEL Register	0x4000.4000	0x420	0x4000.4420	GPIO_PORTA_AFSEL_R
GPIO Port A IS Register	0x4000.4000	0x404	0x4000.4404	GPIO_PORTA_IS_R
GPIO Port A ODR Register	0x4000.4000	0x50C	0x4000.450C	GPIO_PORTA_ODR_R
GPIO Port A PUR Register	0x4000.4000	0x510	0x4000.4510	GPIO_PORTA_PUR_R
GPIO Port A PDR Register	0x4000.4000	0x514	0x4000.4514	GPIO_PORTA_PDR_R
GPIO Port A DEN Register	0x4000.4000	0x51C	0x4000.451C	GPIO_PORTA_DEN_R
GPIO Port A LOCK Register	0x4000.4000	0x520	0x4000.4520	GPIO_PORTA_LOCK_R
GPIO Port A CR Register	0x4000.4000	0x524	0x4000.4524	GPIO_PORTA_CR_R

- **GPIO High-Performance Bus Control (GPIOHBCTL)** Register. As we mentioned, GPIO can be accessed by two kinds of bus apertures, APB or AHB. These two bus apertures are mutually exclusive, which means that at any time, only one bus aperture can be used to access the desired GPIO port. The bus aperture used for each GPIO Port is under the control of the associate bit in the **GPIO High-Performance Bus Control (GPIOHBCTL)** Register, which is a 32-bit register. In the lowest six (6) bits, bits $5 \sim 0$, of this register, each bit is used to control one GPIO Port, Ports $A \sim F$, with the following accessed bus aperture:

  Bit Value 0: The Advanced Peripheral Bus (APB) is used for the mapped GPIO Port
  1: The Advanced High-Performance Bus (AHB) is used for the GPIO Port.

After a system reset, all these six bits are reset to 0, which means that all ports use APB bus aperture after a system reset. Optionally you can configure and setup the associate bits in the **GPIOHBCTL** register to select the accessing bus aperture for the specific GPIO port. This step is optional since all ports use the APB bus aperture after a system reset.

- **General-Purpose Input/Output Run Mode Clock Gating Control (RCGCGPIO)** Register. This register is used to control the connection between the system clock and the desired GPIO Port. All GPIO Ports must have a clock as a timing base to work properly. Similarly to GPIOHBCTL register, each bit of the lowest six (6) bits in this register is mapped to one of six GPIO Ports, from A to F.

  Bit Value 0: The mapped GPIO Port is disabled and no clock is provided
  1: The mapped GPIO Port is enabled and a clock is provided.

After a system reset, all these six bits are reset to 0, which means that all ports are disabled with no clock is connected after a system reset.

Now let's start the initialization and configuration process for a specific GPIO Port. Perform the following operations to initialize and configure a specific GPIO Port:

(1) Setup a connection between the system clock and the specific GPIO **Port to enable the clock to drive the port by setting the appropriate bits in the General-Purpose Input/Output Run Mode Clock Gating Control (RCGCGPIO)** Register.
(2) Setup the direction for each pin on the GPIO port by programming the **GPIODIR** register.

Bit Value 0: The pin works as an Input pin
1: The pin works as an Output pin.

(3) Optionally you can configure the **GPIOAFSEL** register to program each bit as a GPIO Mode or Alternate Mode. This step is optional since most ports work in the GPIO Mode after a system reset. If an alternate pin is chosen for a bit, then the PMCx field must be programmed in the **GPIOPCTL** register for the specific peripheral function. There are also two registers, **GPIOADCCTL** and **GPIODMACTL**, which can be used to program a GPIO pin as an ADC or μDMA trigger, respectively.
(4) Enable GPIO pins as digital I/Os by setting the appropriate **DEN** bit in the **GPIODEN** register. Since all GPIO pins are tri-state status after a reset, this step is necessary. To enable GPIO pins to their analog function, set the **GPIOAMSEL** bit in the **GPIOAMSEL** register.
(5) Optionally, you can setup the drive strength for each pin through the **GPIODR2R**, **GPIODR4R**, or **GPIODR8R** registers. This step is optional since the default drive strength is 2-mA.
(6) Optionally you can configure each pad in the port to have pull-up, pull-down, or open-drain function through the **GPIOPUR**, **GPIOPDR**, or **GPIOODR** register. Slew rate may also be configured, if needed, through the **GPIOSLR** register.
(7) Optionally, you can configure the **GPIOIS**, **GPIOIBE**, **GPIOEV**, and **GPIOIM** registers to setup the type, event, and mask of the interrupts for each port if interrupts are used for the port. This step is optional and unnecessary until interrupts are utilized for a specific GPIO port.
(8) Optionally, you can lock the configurations of the NMI and JTAG/SWD pins on the GPIO port pins by setting the **LOCK** bits in the **GPIOLOCK** register.

It can be found from the initialization and configuration process for a specific GPIO Port shown above that only three steps, steps 1, 2, and 4, are necessary to be done to configure a GPIO Port by the user, and all other steps are optional.

After a system reset, all GPIO pins are configured to be undriven or tri-state with the following values in four registers: **GPIOAFSEL** = 0, **GPIODEN** = 0, **GPIOPDR** = 0 and **GPIOPUR** = 0.

3.2.2 Introduction to Tiva™ C Series LaunchPad™ TM4C123GXL EVB

The Tiva™ C Series TM4C123G LaunchPad™ Evaluation Board (EVB) EK-TM4C123GXL is a low-cost and multiple-function evaluation platform specially designed based on ARM® Cortex®-M4F microcontrollers. Two TM4C123GH6PM microcontrollers are included in this EVB to enable the board to provide various control functions, such as TM4C123GH6PM USB 2.0 device interface, on-board ICDI interface, hibernate and motion control PWM modules.

The TM4C123GXL EVB is specially designed for applying the Tiva™ for C Series Software, including the Tiva™ C Series TivaWare™ libraries, such as

Peripheral Driver Library, USB library, Sensor Hub library, Graphical library, and Tiva™ Boot Loader.

Figure 3.8 shows an illustration photo for the TM4C123GXL evaluation board. The following components and their functions are involved in this board:

- Two TM4C123GH6PM microcontrollers, **MCU-1** and **MCU-2**, are included in this EVB. The former is used as a program loading/debugging controller, and the latter works as a real microcontroller for this board.
- Two USB connectors are provided to support the program development. The **ICDI/Debug USB Connector** is used for programming/debugging purpose, and the **USB Micro-A/-B Connector** can be used as an interface to enable this EVB to work as a USB Device.
- The **Power Select Switch** is used to select the power for the board. When the board works as an evaluation board, the switch should be in the Debug position to enable the program to be downloaded and debugged in the board. Otherwise, the switch should be in the Device position when it works as a USB Device.
- Two user buttons, **SW1** and **SW2**, are provided to support the users' multi-application functions. Both user buttons can be used in the preloaded application program to adjust the light spectrum of the RGB LED as well as going into and returning from the hibernation mode. The user button **SW2** can be used to wake the processor up from the hibernation mode since it is

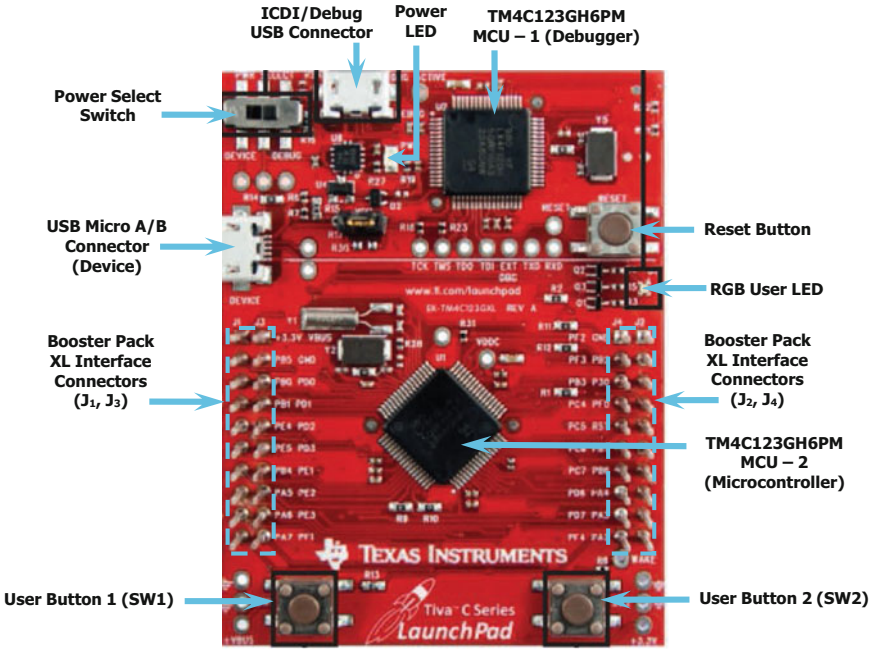


Fig. 3.8 The TM4C123GXL evaluation board. *Courtesy* Texas Instruments

connected to the $\overline{\text{WAKE}}$ pin in the MCU. The user buttons can also be used for other purposes in the user's custom application.
- One **RGB User LED** is provided in the board to enable users to use this LED to develop and build different application programs.
- The **Reset Button** is used to perform a reset for the processor and the entire system.
- There are two double rows of stackable headers that connected to most GPIO pins of the TM4C123GH6PM MCU located at both sides of the board. These rows are labeled as connectors J1, J2, J3, and J4, respectively. All of these headers are connected to related GPIO pins in the MCU and can be connected to other peripheral devices. Tables 3.4, 3.5, 3.6 and 3.7 show these connections. Among them, all five pins in the Port F, PF0 ~ PF4, have been connected to five on-board devices, which is shown in Table 3.8.

In addition to components listed above, the TM4C123GXL EVB contains the following components with the related functions:

(1) **Power Supply**: The EVB can be powered by (1) On-Board ICDI USB cable (**Debug**-default), and (2) USB Device cable (**Device**).
(2) **Clocking**: The EVB used the main internal clock circuit driven by a 16 MHz crystal (Y2) by default. This clock can be modified by software to program a PLL clock generator to get higher clock frequency.
(3) **In-Circuit Debug Interface (ICDI)**: The EVB provides an on-board In-Circuit Debug Interface (ICDI) to enable users to download their program to the flash ROM space in the TM4C123GH6PM MCU and perform the debugging functions for that program. This ICDI only supports the JTAG protocol.
(4) **Virtual COM Ports**: When plugged into a PC, the device enumerates as a debugger and a virtual COM port. Table 3.9 shows the connections of the COM port and the pins on the MCU.
(5) **Hibernate Mode**: The EVB provides an external 32.768-kHz crystal (**Y1**) as the clock source for the TM4C123GH6PM Hibernation module clock source to enable to MCU to work in the hibernate mode to save power consumptions. The user button 2 (**SW2**) can be used as a wake signal to wake up the processor to return to the normal run mode.

The Tiva™ C Series LaunchPad™ provides an easy and inexpensive way to develop applications with the TM4C123GH6PM microcontroller. Tiva™ C Series BoosterPacks and MSP430 BoosterPacks expand the available peripherals and potential applications of the Tiva™ C Series LaunchPad™. BoosterPacks can be used with the Tiva™ C Series LaunchPad™ or you can simply use the on-board TM4C123GH6PM microcontroller as its processor.

Figure 3.9 shows a functional block diagram for this TM4C123GXL evaluation board.

Two USB connectors, USB Debug and USB Device, enable the TM4C123GXL evaluation board to connect to either a host computer to download the user's

Table 3.4 Pin distributions and functions in J1 connector

J1 pin	GPIO	Analog function	On-board function	Digital functions (GPIOPCTL PMCx bit field encoding)									
				1	2	3	4	5	6	7	8	9	14
1	*Power Supply 3.3 V*												
2	PB5	AIN11	–	–	SSI2Fss	–	M0PWM3	–	–	T1CCP1	CAN0Tx	–	–
3	PB0	USB0ID	–	U1Rx	–	–	–	–	–	T2CCP0	–	–	–
4	PB1	USB0VBUS	–	U1Tx	–	–	–	–	–	T2CCP1	–	–	–
5	PE4	AIN9	–	U5Rx	–	I2C2SCL	M0PWM4	M1PWM2	–	–	CAN0Rx	–	–
6	PE5	AIN8	–	U5Tx	–	I2C2SDA	M0PWM5	M1PWM3	–	–	CAN0Tx	–	–
7	PB4	AIN10	–	–	SSI2Clk	–	M0PWM2	–	–	T1CCP0	CAN0Rx	–	–
8	PA5	–	–	–	SSI0Tx	–	–	–	–	–	–	–	–
9	PA6	–	–	–	–	I2C1SCL	–	M1PWM2	–	–	–	–	–
10	PA7	–	–	–	–	I2C1SDA	–	M1PWM3	–	–	–	–	–

Table 3.5 Pin distributions and functions in J2 connector

J2 pin	GPIO	Analog function	On-board function	Digital functions (GPIOPCTL PMCx bit field encoding)									
				1	2	3	4	5	6	7	8	9	14
1	*GROUND*												
2	PB2	–	–	–	–	I2C0SCL	–	–	–	T3CCP0	–	–	–
3	PE0	AIN3	–	U7Rx	–	–	–	–	–	–	–	–	–
4	PF0	–	USR_SW2/WAKE (R1)	U1RTS	SSI1Rx	CAN0Rx	–	M1PWM4	PhA0	T0CCP0	NMI	C0o	–
5	*RESET*												
6	PB7	–	–	–	SSI2Tx	–	M0PWM1	–	–	T0CCP1	–	–	–
	PD1	AIN6	Compatible MSP430	SSI3Fss	SSI1Fss	I2C3SDA	M0PWM7	M1PWM1	–	WT2CCP1	–	–	–
7	PB6	–	–	–	SSI2Rx	–	M0PWM0	–	–	T0CCP0	–	–	–
	PD0	AIN7	Compatible MSP430	SSI3Clk	SSI1Clk	I2C3SCL	M0PWM6	M1PWM0	–	WT2CCP0	–	–	–
8	PA4	–	–	–	SSI0Rx	–	–	–	–	–	–	–	–
9	PA3	–	–	–	SSI0Fss	–	–	–	–	–	–	–	–
10	PA2	–	–	–	SSI0Clk	–	–	–	–	–	–	–	–

Table 3.6 Pin distributions and functions in J3 connector

J3 pin	GPIO	Analog function	On-board function	Digital Functions (GPIOPCTL PMCx bit field encoding)									
				1	2	3	4	5	6	7	8	9	14
1	*Power Supply 5.0 V*												
2	*GROUND*												
3	PD0	AIN7	–	SSI3Clk	SSI1Clk	I2C3SCL	M0PWM6	M1PWM0	–	WT2CCP0	–	–	–
	PB6	–	Compatible MSP430 -R9	–	SSI2Rx	–	M0PWM0	–		T0CCP0	–	–	–
4	PD1	AIN6	–	SSI3Fss	SSI1Fss	I2C3SDA	M0PWM7	M1PWM1	–	WT2CCP1	–	–	–
	PB7	–	Compatible MSP430-R10	–	SSI2Tx	–	M0PWM1	–	–	T0CCP1	–	–	–
5	PD2	AIN5	–	SSI3Rx	SSI1Rx	–	M0FAULT0	–	–	WT3CCP0	USB0EPEN	–	–
6	PD3	AIN4	–	SSI3Tx	SSI1Tx	–	–	–	–	WT3CCP1	USB0PFLT	–	
7	PE1	AIN2	–	U7Tx	–	–	–	–	–	–	–	–	–
8	PE2	AIN1		–	–	–	–	–		–	–	–	–
9	PE3	AIN0	–	–	–	–	–	–	–	–	–	–	–
10	PF1	–	–	U1CTS	SSI1Tx	–	–	M1PWM5	–	T0CCP1	–	C1o	TRD1

Table 3.7 Pin distributions and functions in J4 connector

J4 pin	GPIO	Analog function	On-board function	Digital functions (GPIOPCTL PMCx bit field encoding)									
				1	2	3	4	5	6	7	8	9	15
1	PF2	–	Blue LED (R11)	–	SSI1Clk	–	M0FAULT0	M1PWM6	–	T1CCP0	–		TRD0
2	PF3	–	Green LED (R12)	–	SSI1Fss	CAN0Tx	–	M1PWM7	–	T1CCP1	–	–	TRCLK
3	PB3	–	–	–	–	I2C0SDA	–	–	–	T3CCP1	–	–	–
4	PC4	C1−	–	U4Rx	U1Rx	–	M0PWM6	–	IDX1	WT0CCP0	U1RTS	–	–
5	PC5	C1+	–	U4Tx	U1Tx	–	M0PWM7	–	PhA1	WT0CCP1	U1CTS	–	–
6	PC6	C0+	–	U3Rx	–	–	–	–	PhB1	WT1CCP0	USB0EPEN	–	–
7	PC7	C0−	–	U3Tx	–	–	–	–	–	WT1CCP1	USB0PFLT	–	–
8	PD6	–	–	U2Rx	–	–	–	–	PhA0	WT5CCP0	–	–	–
9	PD7	–	–	U2Tx	–	–	–	–	PhB0	WT5CCP1	NMI	–	–
10	PF4	–	USR_SW1 (R13)	–	–	–	–	M1FAULT0	IDX0	T2CCP0	USB0EPEN	–	–

Table 3.8 GPIO Port F pins and on-board devices

GPIO pin	Pin function	On-board devices
PF4	GPIO	User Button 1 (SW1)
PF0	GPIO	User Button 2 (SW2)
PF1	GPIO	RGB LED—Red
PF2	GPIO	RGB LED—Blue
PF3	GPIO	RGB LED—Green

Table 3.9 Virtual COM port signals

GPIO pin	Pin function
PA0	U0Rx
PA1	U0Tx

program into the flash memory in this EVB and debug it or some other controllers to allow this EVB to work as a USB device.

Two user buttons, **SW1** and **SW2**, are used to control the RGB LED intensity and wake the processor up from the hibernation mode via the GPIO Port F (PF4 and PF0).

One of the most important components in this EVB is the 40-pin BoosterPack XL expansion headers (J1 ~ J4). These headers allow users to connect other desired peripherals with this EVB to perform multiple control functions for their actual applications. These headers are also compatible with MSP430 MCU.

3.2.3 Introduction to EduBASE ARM® Trainer

The EduBASE ARM® Trainer is specially designed for Tiva™ for C Series LaunchPad™ microcontroller evaluation board EK-TM4C123GXL. Multiple peripheral devices and components are provided by this trainer.

Figure 3.10 shows the photo of this trainer. The main peripherals and components installed on this board include

- 16 × 2 LCD display module with LED backlight
- 4-digit, 7-segment display module for learning multiplexing technique
- 4 × 4 keypad
- Four data LEDs
- A 4-position DIP switch
- Four pushbutton switches
- Speaker
- Light sensor for home automation applications
- Potentiometer for analog input
- X-Y-Z accelerometer module interface header

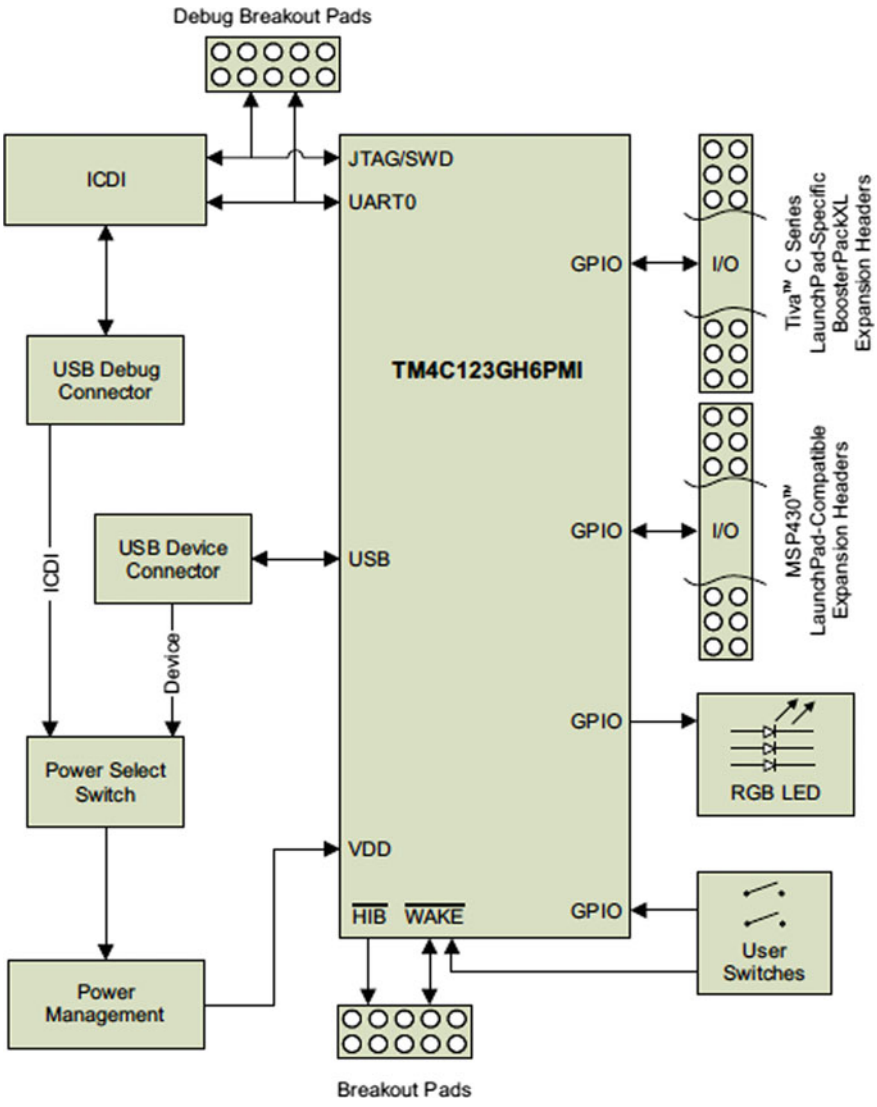


Fig. 3.9 The functional block diagram of the LaunchPad board. *Courtesy* Texas Instruments

- Two analog sensor inputs
- Four servo or relay outputs
- SPI based dual 12-bit DAC for generating analog waveforms
- I^2C based Real Time Clock with a capacitor backup
- High-efficiency dual H-Bridge for controlling two DC motors or one stepper motor

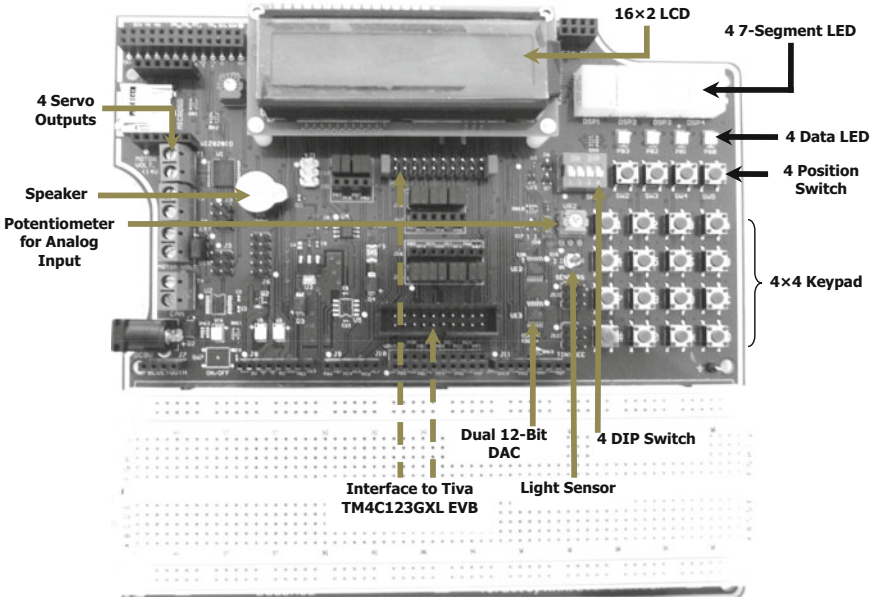


Fig. 3.10 The EduBASE ARM® Trainer

An interface is provided to allow the TM4C123GXL EVB to be inserted into this Trainer. Two dashed lines in Fig. 3.10 show these two connectors. A complete connection between the TM4C123GXL EVB and the EduBASE ARM® Trainer is shown in Fig. 3.1.

3.2.4 *Introduction to Other Related Peripherals and Interfaces*

Some other important and related peripherals and interfaces, such as DC Motors, Pulse Width Modulation (PWM) module and Quadrature Encoder Interface (QEI), will be used for class projects and lab projects through this book. In this section we will concentrate on these components.

The motor we used is a DC motor, **Mitsumi 448 PPR Motor** with an optical rotary encoder including a 448 lines/pulses speed disk. This motor can be purchased from some sites such as ebay for about $5.00. The reason we select this kind of motor is because an optical rotary encoder is attached with the motor axis to provide a position/speed feedback to enable us to design a closed-loop control system easily.

The details about this motor will be discussed in the later chapter with closed-loop control system together to provide a complete control strategy for

designers. Similar to DC motor section, we will provide more detailed discussion about the QEI Module in the PID control system design section.

Since the PWM Module in TM4C123GH6PM MCU system is relatively complicated, therefore we leave that part in Chap. 4.

3.3 Introduction to the Software Package

The software package is composed of the following components:

- Integrated Development Environment Keil® ARM®-MDK µVersion®5.24a (IDE).
- TivaWare™ SW-EK-TM4C123GXL Software Driver Package.
- Stellaris In-Circuit Debug Interface (ICDI).

First let us have a global picture about these components and their relationships.

3.3.1 Overview ARM Microcontroller Development Kits

Basically the whole development kits can be divided into two layers; (1) The Keil® MDK-ARM® Suite that provides a graphic user interface (GUI) with all general required development tools, and (2) The TivaWare™ for C Series LaunchPad™ firmware that provides specified software and libraries for Tiva™ C Series TM4C123G evaluation board.

It can be found from Fig. 3.11 that two dash-line are pointed to the Debug Adaptor. This means that both Keil® MDK and TivaWare™ provide a related device driver for the debug adaptor that is connected between the software tools and the evaluation board. The ULINK2 and the Tiva™ In-Circuit Debug Interface (ICDI). The former is a device driver for the Debug Adaptor developed by the Keil® MDK, and the latter is a device driver built by the Texas Instruments. This debug adaptor is used to perform some necessary communications between the software development tools and the microcontroller hardware, in this case, TM4C123G EVB. These communications include the debugging user's programs

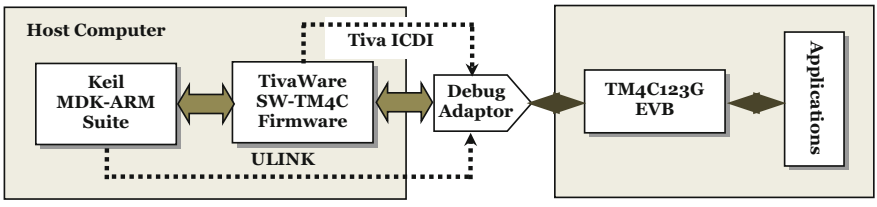


Fig. 3.11 Configuration of Tiva TM4C123G based EVB

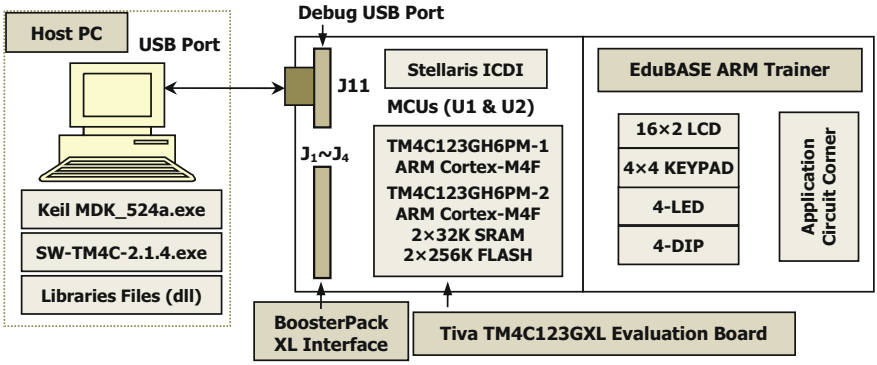


Fig. 3.12 The setup and connection of entire Tiva TM4C123G based development system

and downloading the compiled programs to the EVB, either to RAM or Flash memory.

The development kits are composed of a Tiva™ TM4C123GXL EVB and an EduBASE ARM® Trainer that contained some other useful peripheral devices. The hardware setup and connection for the whole evaluation system is shown in Fig. 3.12.

It can be found from Fig. 3.12 that the entire Tiva™ TM4C123G based development system, EduBASE ARM® Trainer, is composed of two pieces of important hardware components: the host computer (PC) that works as a control unit and the EduBASE ARM® Trainer Evaluation Board. The components and functions of these two pieces of hardware are:

- The host computer works as an interface to enable users to create, assembly, debug and test the user's program in the EduBASE ARM® Trainer using the Keil® IDE and TivaWare™ firmware installed in the host computer. All these functions are performed by accessing various libraries and tools provided by the TivaWare™ firmware and Keil® IDE installed in the host computer.
- The EduBASE ARM® Trainer provides all hardware and software interfacing abilities to facilitate the above operations performed in the host computer.

The commands and data communications between the host computer and the EduBASE ARM® Trainer are made through the USB ports, USB Port in the host computer and the Debug USB Port in the Tiva™ TM4C123GXL EVB, via a USB cable.

For the Tiva™ TM4C123GXL microcontroller evaluation board, more than 15 development platforms are available and they are provided by the different vendors. However, the following platforms and tools are relatively popular:

- Keil® ARM® RealView® Microcontroller Development Kit (MDK)
- Texas Instruments' Code Composer Studio™ IDE (CCS)
- ARM® DS-5 Development Studio 5

- IAR Embedded Workbench for ARM®
- Mentor Graphics Sourcery CodeBench
- GNU Compiler Collection (GCC)

Among those tools and platforms, one of the popular choices is the Keil® Microcontroller Development Kit for ARM®, or MDK-ARM®. This MDK contains all required components and tools to develop application programs for ARM®-related microcontrollers.

The MDK-ARM® is a complete software development environment for Cortex™-M, Cortex-R4, ARM7™ and ARM9™ processor-based devices. MDK-ARM® is specifically designed for microcontroller applications, it is easy to learn and use, and powerful enough for the most demanding embedded applications.

MDK-ARM® is available in four editions: MDK-Lite, MDK-Cortex-M, MDK-Standard, and MDK-Professional. All editions provide a complete C/C++ development environment and MDK-Professional includes extensive middleware libraries. Since we are using ARM® Cortex®-M4 MCU, we will concentrate our discussion on MDK-Cortex-M development system.

As we discussed at the beginning of this chapter, the hardware configuration of the entire Tiva™ TM4C123G development system is composed of the following components:

- Tiva™ TM4C123GH6PM MCUs
- Tiva™ C Series TM4C123GXL EVB
- EduBASE ARM® Trainer
- Host Computer

This configuration can be presented in Fig. 3.13.

Based on the discussion above, the entire development system for ARM® microcontroller can be composed of the following components or tools

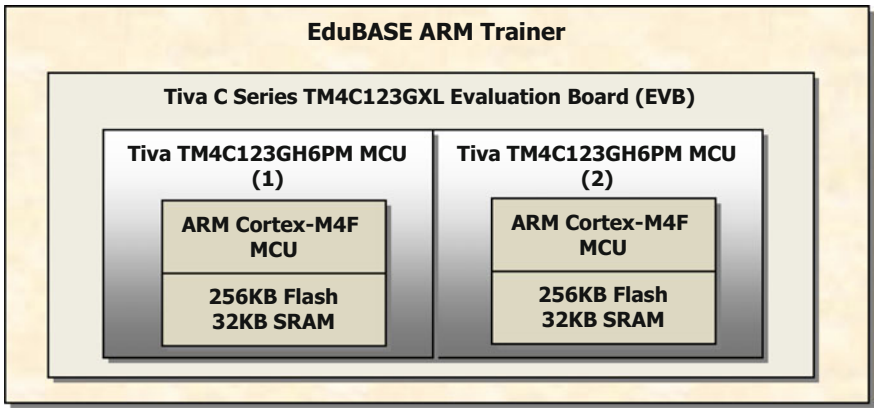


Fig. 3.13 The configuration of the EduBASE ARM Trainer hardware

1. Development Kits or Suites
2. Debug Adaptor and Drivers
3. Specified MCU-related Firmware
4. Program Examples
5. Development or Evaluation Boards

Since the development evaluation board has been discussed in the last chapter, now let's discuss top four components in the following section one by one. However, before we can continue our discussions for these components, we need first to download and install them in the host computer.

3.3.2 Download and Install Development Suite and Specified Firmware

The first three components, development kits, specified firmware and debug adaptor drivers, are three different components and can be downloaded and installed separately. The program examples are MCU-related and they can be installed with the specified firmware together.

Refer to Appendix A to download and install Keil® MDK-ARM® 5.24a. This installation process not only installs the MDK Core, but also installs some Software Packs. An icon of installed Keil® MDK, **Keil µVersion5**, will be added into your desktop when the installation is complete. The default installation location of this development suite on your host PC is *C:/Keil_v5*.

Refer to Appendix B to download and install Tiva™ C Series specified firmware *TivaWare SW-EK-TM4C123GXL* Package. The default installation location of this firmware in your host computer is *C:/ti/TivaWare_C_Series-2.1.4.178*.

Refer to Appendix C to download and install Stellaris® In-Circuit Debug Interface (ICDI) and Virtual COM Port. The installation location for this ICDI device driver is in the folder *Stellaris® In-Circuit Debug Interface*, which is under the **Device Manager** in the **Control Panel** in your host computer.

Now let's discuss these four components in the following section one by one. Since the MDK Core contains the Debugger, we will discuss these two components together.

3.3.3 Introduction to Integrated Development Environment —Keil® MDK

The Keil® MDK is the most comprehensive software development environment for ARM®-based microcontrollers. MDK Version 5 is now split into the MDK Core and Software Packs which makes new device support and middleware updates independent from the toolchain.

The entire Keil® MDK development system can be divided into the following key components:

- The MDK Core
  - µVersion® IDE with Source Editor and GUI
  - Pack Installer
  - ARM® C/C++ Compiler
  - µVersion® Debugger with Trace Function
- Software Packs
  - Device Drivers for Serial Peripheral Interface (SPI), USB, and Ethernet
  - The Cortex Microcontroller Software Interface Standard (CMSIS) support, including the CMSIS-CORE, CMSIS DSP and CMSIS-RTOS
  - MDK Middleware support
  - Example programs

MDK Core

The MDK Core contains all development tools including µVersion IDE, Compiler, and Debugger. By using the MDK Core, you can create, build, and debug an embedded application for Cortex-M processor-based microcontroller devices. The new Pack Installer adds and updates Software Packs for devices, CMSIS, and middleware. The purpose of the new added Pack Installer is to manage Software Packs that can be added any time to the MDK Core. This makes new device support and middleware updates independent from the toolchain. Software Packs that add support for a complete microcontroller family are called Device Family Packs.

An illustration block diagram of the MDK Core and its components is shown in Fig. 3.14.

Software Packs

Software Packs contain device support, CMSIS libraries, middleware, board support, code templates, and example projects. Among all components included in the Software Packs, two components, CMSIS and Middleware, needed to be explained in more details.



Fig. 3.14 The components included in the MDK Core (Reproduced with permission from Arm® Limited. Copyright© Arm Limited)

The CMSIS provides a ground-up software framework for embedded applications that run on Cortex-M based microcontrollers. The CMSIS enables consistent and simple software interfaces to the processor and the peripherals, simplifying software reuse, reducing the learning curve for microcontroller developers.

The CMSIS application software components include

- **CMSIS-CORE**: Defines the API for the Cortex-M processor core and peripherals and includes a consistent system startup code. The software components ***CMSIS:CORE*** and ***Device:Startup*** are all you need to create and run applications on the native processor that uses exceptions, interrupts, and device peripherals.
- **CMSIS-RTOS**: Provides standard real-time operating systems and enables, therefore, software templates, middleware, libraries, and other components that can work across supported RTOS systems.
- **CMSIS DSP**: Is a library collection for digital signal processing (DSP) with over 60 Functions for various data types: fix-point (fractional q7, q15, q31) and single precision floating-point (32-bit).

The MDK-Professional Middleware offers a wide range of communication peripherals to meet many embedded design requirements, and it is essential to make efficient use of these complex on-chip peripherals. The MDK Professional Middleware provides a Software Pack that includes royalty-free middleware with components for TCP/IP networking, USB Host and USB Device communication, file system for data storage, and a Graphical User Interface.

A complete block diagram including all Software Packs components is shown in Fig. 3.15.

Because the MDK is a powerful development suite with great amount of components, in this chapter, we will concentrate our discussions on the MDK Core, especially on the MDK-Cortex-M family. Two components, µVersion IDE and Debugger, are main topics to be discussed in this Chapter.

First let's have a closer look at the Keil® MDK-ARM® works for the Tiva™ C Series LaunchPad™ evaluation board.

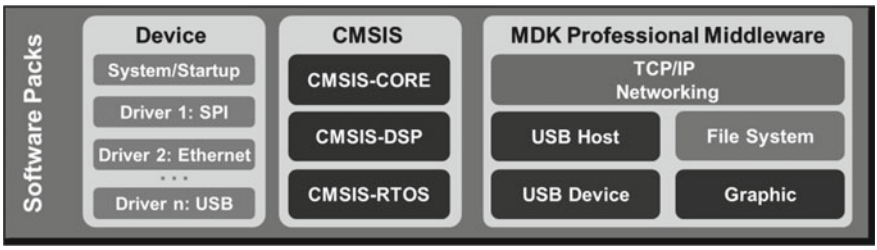


Fig. 3.15 Components in the Software Packs (Reproduced with permission from Arm® Limited. Copyright© Arm Limited)

3.3.3.1 The Keil® MDK-ARM® for the MDK-Cortex-M Family

Similar to Keil® MDK-ARM®, the MDK-Cortex-M family contains the following components:

(1) **µVision5 Integrated Development Environment (IDE)**: Provides a GUI with all general required development tools, such as debugger and simulation environment.
(2) **ARM Compilation Tools**: These tools include C/C++ Compiler, ARM® Assembler, Linker and other utilities.
(3) **Debugger**: Provides debug functions for ARM® microcontroller programs.
(4) **Simulator**: Provides simulation environment to enable users to build and run program without any real hardware.
(5) **Keil RTX Real-Time Operating System Kernel**: Provides a real operating system kernel.
(6) **TCP/IP Networking Suite**: Offers multiple protocols and various applications.
(7) **USB Device and USB Host stacks** are provided with standard driver classes.
(8) **ULINK*pro*** enables on-the-fly analysis of running applications and records every executed Cortex-M instruction.
(9) **Complete Code Coverage** information about your program's execution.
(10) **Execution Profiler and Performance Analyzer** enable program optimization.
(11) **CMSIS Cortex Microcontoller Software Interface Standard** compliant.
(12) **Reference startup codes** for about 1000 microcontrollers.
(13) **Flash Programming Algorithms**.
(14) **Program examples**.

A complete configuration of the MDK-ARM® for Cortex®-M family is shown in Fig. 3.16.

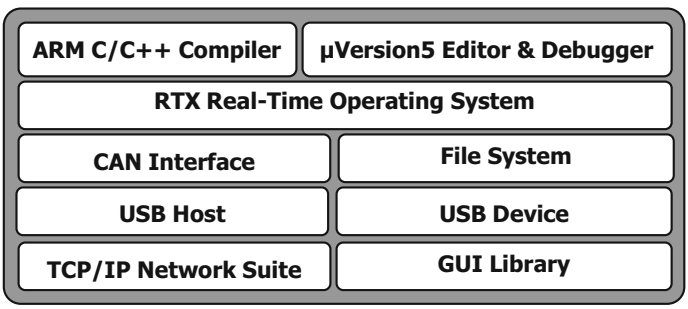


Fig. 3.16 The complete structure of the MDK-Cortex-M development system

To better understand the program development process with Keil® MDK, first let's have a detailed discussion about the general development flow of a user project by using the MDK-Cortex®-M development system.

3.3.3.2 General Development Flow with MDK-ARM®

Figure 3.17 shows a general development process of a user project in the Keil® MDK.

Generally, a user project in Keil® MDK can be developed in the following steps:

- A new project is created using the Keil® MDK.
- The user's source files with source codes, either C or ARM Assembly codes, are added into the project.
- Depends on your source codes, the **armcc** (ARM® Compiler) or the **armasm** (ARM® Assembler) is called and executed to translate the user's source codes to the object codes and stored in the host computer. The startup codes with Project Settings will be involved into this compiling or assembling process.
- The object code files will be linked with all other system library files or MCU-related library files and converted to the executable files, or image file in the ARM® terminology, and downloaded into flash memory or RAM in the EVB.
- Finally the executable codes can be sent to the debugger to perform the debugging or executing operations. In fact, the compiler, linker and loader are integrated into one unit, the Builder in the ARM® μVersion® IDE.
- Alternatively, the executable file can also be converted to the binary file or text file for the users' reference.

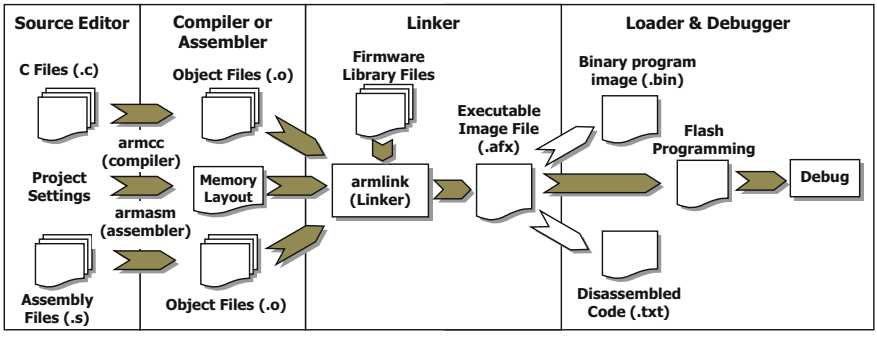


Fig. 3.17 Development structure of a user project with MDK

To successfully build a basic user project with Keil® MDK, one needs to use the following two key components:

1. Keil® MDK Core
2. CMSIS Core

The MDK Core provides all development tools including µVersion IDE, Compiler, and Debugger. The CMSIS-CORE defines the API for the Cortex-M processor core and peripherals and includes a consistent system startup code. The software components ***CMSIS:CORE*** and ***Device:Startup*** are all you need to create, build and run applications on the native processor that uses exceptions, interrupts, and device peripherals.

All user source codes can be written in C, however, the startup codes that are provided by MCU vendors and generally included in the Keil® MDK installation process are ARM® assembly codes. The users also need to use some library files provided by the MCU vendor, in our case, the firmware TivaWare™ SW-EK-TM4C123GXL Package provided by Texas Instruments. This situation is shown in Fig. 3.18.

As shown in Fig. 3.18, to build and develop an ARM® application project, a native Cortex-M Core with CMSIS Core should be used. In fact, one CMSIS Core component ***CMSIS:CORE***, which should be used together with the software component ***Device:Startup***, is used to build a successful project. These components provide the following central files:

- The **startup_<device>.s** file with reset handler and exception vectors.
- The **system_<device>.c** configuration file for basic device setup (clock and memory bus).
- The **system_<device>.h** include file for user code accessing to the microcontroller device.

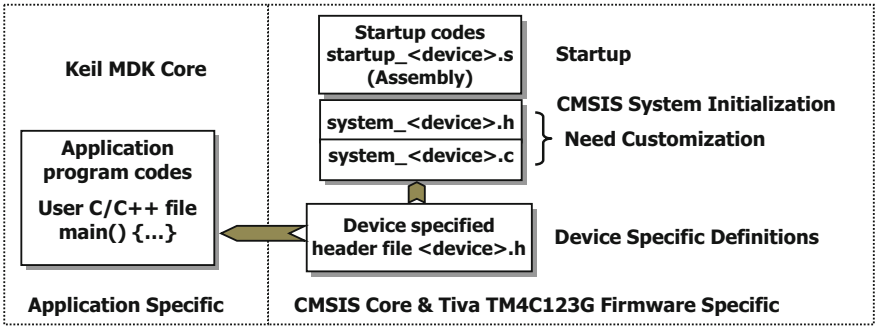


Fig. 3.18 The program development with MDK Core and CMSIS Core

The device specified **<device>.h** header file is included in C source files and defines:

- Peripheral Access with standardized register layout.
- Access to Interrupts and Exceptions and the Nested Interrupt Vector Controller (NVIC).
- Intrinsic Functions to generate special instructions, for example to activate sleep mode.
- Systick Timer (SYSTICK) functions to configure and start a periodic timer interrupt.
- Debug Access for printf-style I/O and ITM communication via on-chip CoreSight.

One point to be noted is that in an actual application file, the <*device*> is the name of the microcontroller device used in the real user project. For example, in our case, this device name should be <*tm4c123gh6 pm*>. Also not all these four files can be found at the development stage, and some files, **system_<device>.h** and device specified **<device>.h**, cannot be found until the program has been built successfully. Let us start with an example project to familiar this IDE.

3.3.3.3 Warming up Keil® MDK-Cortex-M Kit with Example Projects

Before open the Keil® MDK-ARM® µVersion 5.24a Suite, make sure that the following two important components have been setup with your host computer:

- The Tiva™ TM4C123G based EVB has been connected to your host PC with the USB cable. Refer to Fig. 3.1 to complete this hardware setup if you have not.
- All development tools have been downloaded and installed as we did in Sect. 3.3.2.

Perform the following operations to launch Keil® MDK-ARM® µVersion® 5.24a IDE, load and run an example project **Hello World**.

1. Double click on the icon **Keil µVersion5**, which is located on your PC's desktop, to launch and open this kit, which is shown in Fig. 3.19.
2. From the **Project** menu, select **Open Project** menu to open the **Select Project File** wizard.
3. Use this wizard to browse to the **Hello World** project, which should be located in the **ti** folder when we installed this TivaWare™ package. The actual location of this project is: **C:/ti/TivaWare_C_Series-2.1.4.178/examples/boards/ek-tm4c123gxl/hello**. Browse to that folder and select the project file **hello.uvproj**, and click on the **Open** button to load and open this project (Fig. 3.20).
4. The opened project Hello World is shown in Fig. 3.21.

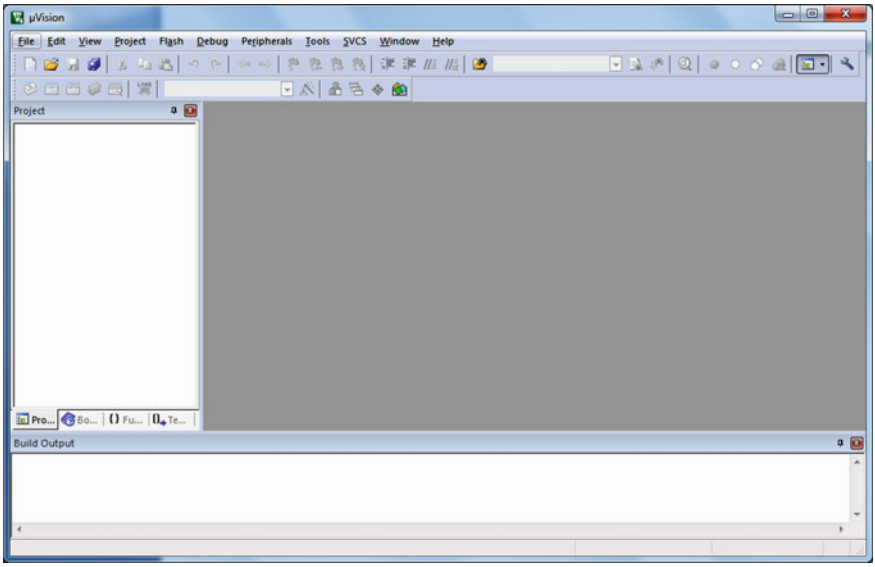


Fig. 3.19 The opened Keil® MDK-ARM µVersion® 5.24a suite (Reproduced with permission from Arm® Limited. Copyright© Arm Limited)

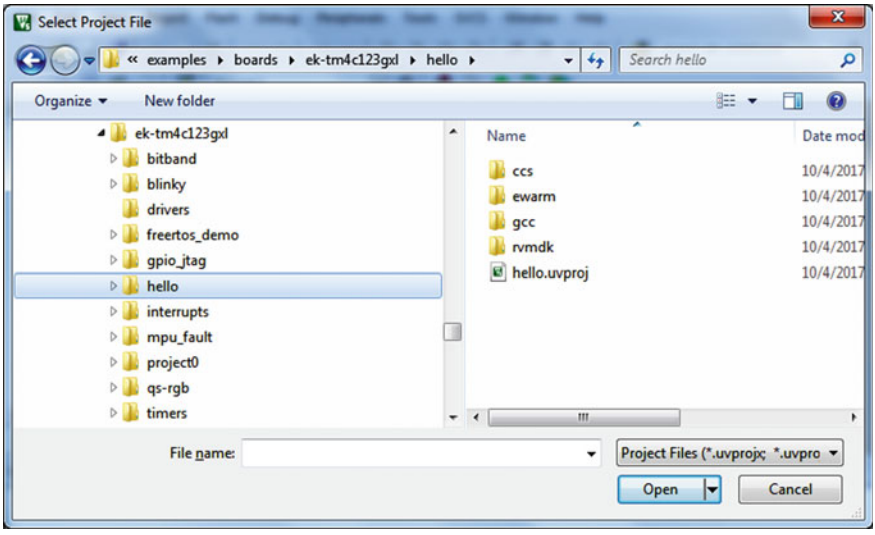


Fig. 3.20 The Hello World project (Reproduced with permission from Arm® Limited. Copyright© Arm Limited)

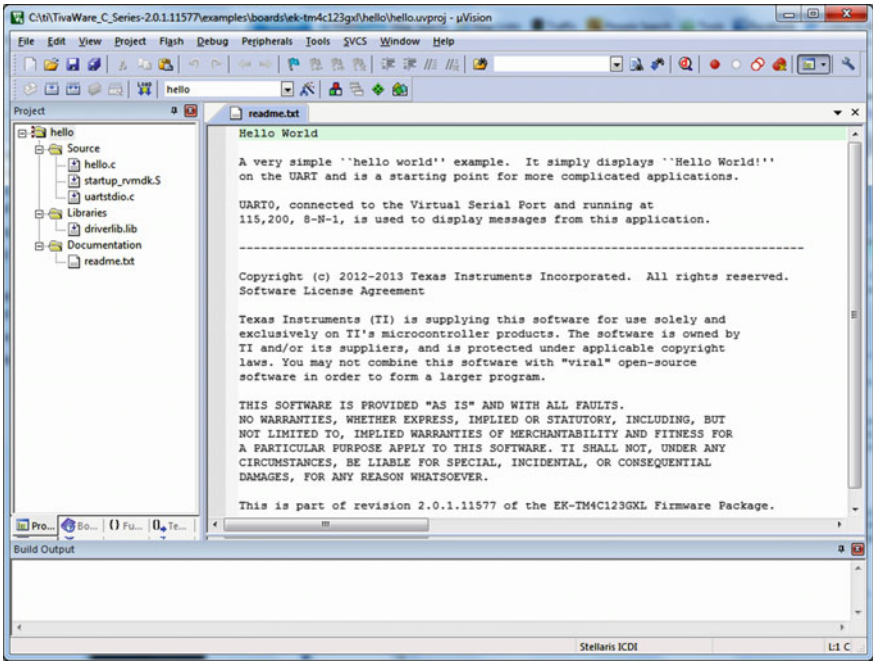


Fig. 3.21 The opened Hello World project (Reproduced with permission from Arm® Limited. Copyright© Arm Limited)

5. You can view any file in the Project Workspace Pane in the left by double-clicking on the desired file. For example, to view the source file, **hello.c**, just double click on this file. The opened source file is shown in Fig. 3.22.
6. You can build the project Hello World by clicking on the **Project|Rebuild all target files**, as shown in Fig. 3.23. All of the source files are compiled and linked. The activity can be seen in the Build window at the bottom of the µVision IDE. The process completes with an application named **hello.axf** built with no errors and no warnings, which can be shown in Fig. 3.24.
7. Now let us load the **Hello** Program into the flash memory in the Tiva™ C Series LaunchPad™ EVB, TM4C123G. You can debug with either the on-board ICDI or you can use the Keil® ULINK™ debug probe. Because we did install the Keil® MDK Debugger driver, ULINK2, when we installed the Keil® MDK. However, we did not have this debug adaptor or the debug hardware. Therefore we have to use the Tiva C Series on-board ICDI since the TI has a built-in debug adaptor installed in the TM4C123G EVB.
8. To begin this loading process, select the **Download** menu item from the **Flash** menu, or click the **Download** button (icon). The process takes a few seconds. A progress bar will show at the bottom of the IDE window as the device is programmed. When it is finished, the Build window will show that the device was erased, programmed, and verified OK. The Hello application is now

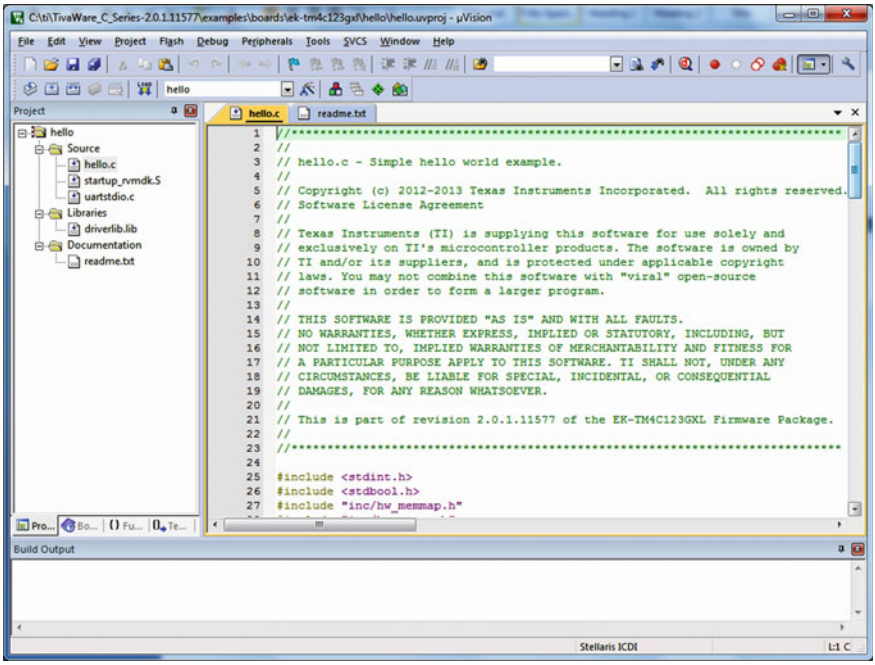


Fig. 3.22 The opened source file hello.c (Reproduced with permission from Arm® Limited. Copyright© Arm Limited)

programmed into the flash memory of the Stellaris® microcontroller on the Evaluation Board.

9. Now let us debug and run the Hello World project. Select the **Start/Stop Debug Session** from the **Debug** menu or click the **Debug** button. The IDE switches to debugging mode.
   A warning message is displayed to indicate that the board using the evaluation mode with a space limitation of 32 KB. Click on the **OK** button to continue. The processor registers show in a window on the left, the debugger command window is visible at the bottom, and the main window shows the source code being debugged. The debugger automatically stops at main, as shown in Fig. 3.25.
10. To run this project, select the **Run** from the **Debug** menu, or click on the **Run** button (icon). The application runs and the text **Hello World!** should be displayed on an UART-related device. Since we do not connect to any UART with our PC, the only visible running result is that the RGB User LED (D1) on the TM4C123G EVB is flashing with the blue color periodically. This flashing function is related to the codes written in the **hello.c** file, exactly the codes between lines 145 and 166 in a **while()** loop.

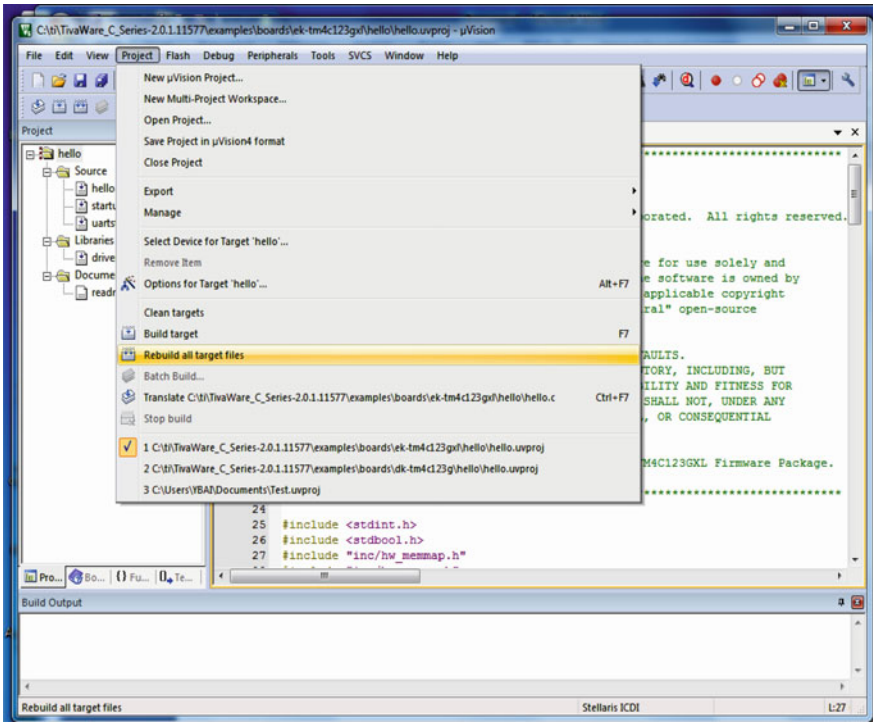


Fig. 3.23 Rebuild the hello project (Reproduced with permission from Arm® Limited. Copyright© Arm Limited)



Fig. 3.24 The building result of the hello project (Reproduced with permission from Arm® Limited. Copyright© Arm Limited)

Now you can stop the project by clicking on the **Stop** menu item from the **Debug** menu. You may go to **Flash|Erase** menu item to erase your download program from the Flash memory in the EVB. Otherwise, your program will continue to run forever. Select the **Close Project** item from the **Project** menu to close this Hello World project.

After finish this example project, you should have some basic idea and feel about the Keil® MDK µVersion IDE. Next, let us have a closer look at all functions of this IDE.

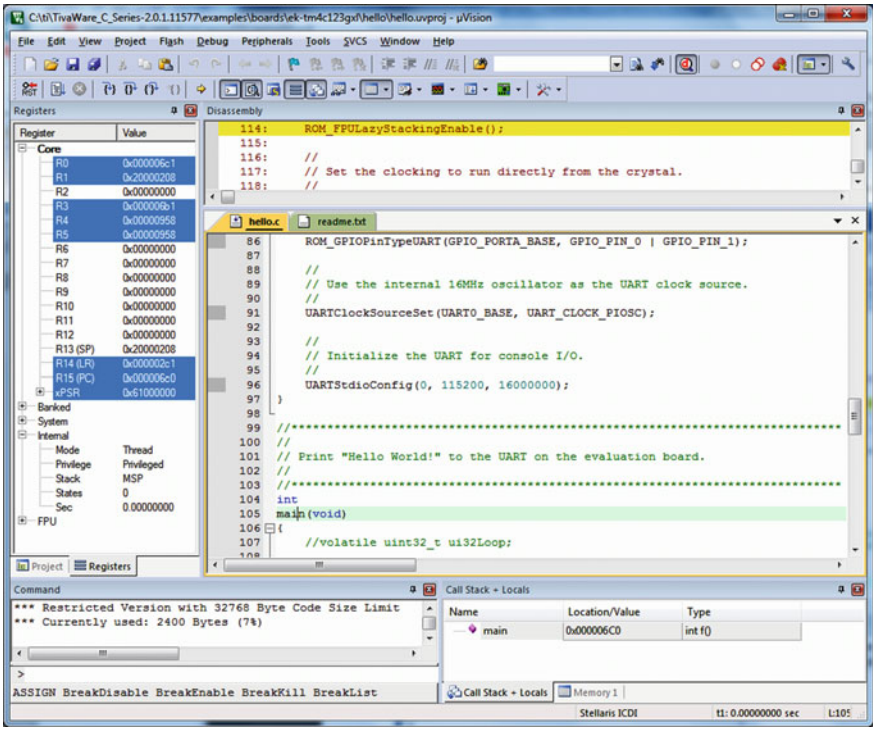


Fig. 3.25 The debugging result of the hello world project (Reproduced with permission from Arm® Limited. Copyright© Arm Limited)

3.3.3.4 The Functions of the Keil® MDK-ARM® µVersion®5 GUI

In this section, we will have an overview about this MDK-ARM® µVision®5 GUI.

The µVision5 is an updated window-based software development platform that combines a robust and modern editor with a project manager and make facility tool. It integrates all the tools needed to develop embedded applications including a C/C ++ compiler, macro-assembler, linker/locator, and a HEX file generator. The µVision5 helps expedite the development process of embedded applications by providing the following:

- Full-featured source code editor.
- Device Database® for configuring the development tool.
- Project Manager for creating and maintaining your projects.
- Integrated Make Utility functionality for assembling, compiling, and linking your embedded applications.
- Dialogs for all development environment settings.
- True integrated source-level and assembler level Debugger with high-speed CPU and peripheral Simulator.

- Advanced GDI interface for software debugging on target hardware and for connecting to a Keil® ULINK™ Debug Adapter.
- Flash programming utility for downloading the application program into Flash ROM.
- Links to manuals, online help, device datasheets, and user guides.

The **µVision5** IDE and **Debugger** are the central part of the Keil® development toolchain and have numerous features that help the programmer to develop embedded applications quickly and successfully. The Keil® tools are easy to use, and are guaranteed to help you achieve your design goals in a timely manner.

The µVision5 offers a **Build Mode** for creating applications and a **Debug Mode** for debugging applications. Applications can be debugged with the integrated µVision5 **Simulator** or directly on hardware, for example with adapters of the Keil® ULINK™ USB-JTAG family. Developers can also use other AGDI adapters or external third-party tools for analyzing applications.

The µVision5 GUI provides Menus for selecting commands and Toolbars with command buttons. The Status Bar, at the bottom of the window, displays information and messages about the current µVision5 command. Windows can be relocated and even docked to another physical screen. The window layout is saved for each project automatically and restored the next time the project is used. You can restore the default layout using the menu **Window|Reset View to Defaults**.

3.3.4 Introduction to TivaWare™ for C Series Software Suite

Texas Instruments' Tiva™ C Series software provides supports to most popular MCUs, such as ARM® Cortex™-M4 core, with scalable memory and package options, unparalleled connectivity peripherals, and advanced analog integration.

The TivaWare™ for C Series suite contains and integrates all user-required source-code functions and objects libraries, which include

- Peripheral Driver Library (DriverLib)
- Graphic Library
- USB Library
- Sensor Hub Library
- Open Source RTOS
- Open Source Stacks
- Utilities
- Boot Loader and In-System Programming Support
- Example Codes
- Third-Party Examples

Figure 3.26 shows an illustration block diagram for these source-code functions and libraries included in the TivaWare™ for C Series suite. TivaWare™ for C

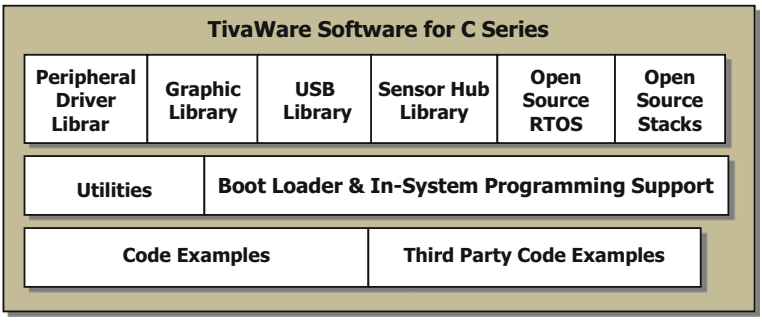


Fig. 3.26 Tools included in the TivaWare for C Series suite

Series libraries offer users the flexibility of working with sample applications or the freedom to create their own projects.

Different source-code functions and libraries are provided by this suite, which include

- The **Peripheral Driver Library** offers an extensive set of functions for controlling the peripherals found on various TM4C devices.
- The **Graphic Library** includes a set of graphics primitives and a widget set for creating graphical user interfaces on Tiva™ C Series-based microcontroller boards that have a graphical display.
- The **USB Library** provides a TivaWare™ royalty-free USB stack to enable efficient USB host, device, and on-the-go operations.
- The **Sensor Hub Library** offers an advanced sensor fusion algorithm and a broad range of sensor support.
- The **Utilities** provide all required developing tools and user-friendly functions to make the user-program development easier and simpler.
- The **Boot Loader and In-System Programming** support users to build the startup codes, install them at the beginning of the flash ROM in the EVB, and run them when the user-program starts.
- The **Open Source RTOS** is a Free RTOS that is a popular real-time operating system for most popular embedded devices. This RTOS accelerates development schedules by eliminating the need to create basic system software functions from scratch. It also scales from a real-time multitasking kernel, the TI-RTOS Kernel to a complete RTOS solution including additional middleware components and device drivers. By providing essential system software components pretested and pre-integrated, the RTOS enables developers to focus on differentiating their application.
- The **Open Source Stacks** offers different stacks for most popular host and peripheral devices.
- The **Code Examples** offer some useful coding guides to help users to start and speed up their coding developments.

- The **Third-Party Code Examples** provide some codes developed by different vendors.

Among these software selections, we are more interested in the TivaWare C Series LaunchPad Evaluation Software Package, which is closely related to the Peripheral Driver Library. Let us have a closer look at this package.

3.3.4.1 TivaWare™ C Series LaunchPad™ Evaluation Software Package

The SW-EK-TM4C123GXL package contains the TivaWare™ for C Series release for the Tiva™ C Series TM4C123G LaunchPad. This package includes the latest version of the TivaWare™ for C Series Peripheral Driver Library, USB Library, Sensor Hub Library, and Graphics Library. The package also contains some Utilities supports and Boot Loader as well as In-System Programming functions. It also includes several complete example applications for the Tiva™ C Series LaunchPad. Refer to Appendix B to download and install this package on your host computer system if you have not done it.

The four separate **EK-TM4C123GXL** packages, each contain the **SW-EK-TM4C123GXL** software as well as installation files for each respective IDE (CCS, Keil®/ARM®-MDK, IRA, and GNU) and the Microsoft® Windows® drivers for the debug interface. These packages also contain several documents to help you get started with using the Tiva™ C Series LaunchPad. Each of the four files is single package with everything you need to program and debug the Tiva™ C Series LaunchPad using the respective toolset.

Use one of these packages if you need to install a different IDE on your host computer

- **EK-TM4C123GXL-CCS**: TivaWare™ for C Series and Code Composer Studio™ (CCS) for the Tiva™ C Series TM4C123G LaunchPad™.
- **EK-TM4C123GXL-IAR**: TivaWare for C Series and IAR Embedded Workbench for the Tiva C Series TM4C123G LaunchPad.
- **EK-TM4C123GXL-KEIL**: TivaWare™ for C Series and Keil® ARM®-MDK for the Tiva C Series TM4C123G LaunchPad™.
- **EK-TM4C123GXL-CB**: TivaWare™ for C Series and Mentor Sourcery CodeBench for the Tiva C Series TM4C123G LaunchPad™.
- **SW-EK-TM4C123GXL**: TivaWare™ for the Tiva™ C Series TM4C123G LaunchPad Evaluation Board Software.

One point to be noted is that no matter which IDE you selected and have installed in your host computer, you must download and install the software package **SW-EK-TM4C123GXL** if you used a Tiva™ C Series TM4C123G LaunchPad Evaluation Board since you need this package to provide all required supporting code functions and object libraries to help you develop your desired projects in your selected IDE. As we mentioned, this package contains four

different versions of software **SW-EK-TM4C123GXL**, each one works for one related IDE.

For our discussions and implementations in this book, we used Keil® ARM®-MDK as the IDE (Keil® μVersion5 IDE) and Tiva™ for C Series TM4C123G evaluation board, therefore we need to download and install **EK-TM4C123GXL-KEIL** and **SW-EK-TM4C123GXL** package on our host computer. In fact, the package **EK-TM4C123GXL-KEIL** only includes all libraries and functions supporting to Keil® ARM-MDK IDE. Instead of installing that package, we can download and install the Keil® ARM-MDK μVersion5 IDE, **MDK.524a.exe**, and a TivaWare™ general-purpose software package, **SW-TM4C-2.1.4.178.exe**, which contains libraries and functions supporting to all four IDEs. Refer to Appendices A and B to download and install these software if you have not done so.

The software package provided with the SW-EK-TM4C123GXL provides access to all of the peripheral devices supplied in the design. The TivaWare™ for C Series Peripheral Driver Library is used to operate the on-chip peripherals.

The software includes a set of example applications that use the TivaWare™ Peripheral Driver Library. These applications demonstrate the capabilities of the TM4C123GH6PM microcontroller, as well as provide a starting point for the development of the applications for use on the TM4C123GXL development board.

After the TivaWare™ C Series LaunchPad Evaluation Software Package **SW-EK-TM4C123GXL** has been installed in your host computer, all related code functions and object libraries files are located at the following folders by default

- Hardware/device-related specified header files: **C:/ti/TivaWare_C_Series-<version>/inc**
- Peripheral driver library files: **C:/ti/TivaWare_C_Series-<version>/driverlib/rvmdk**
- Graphic library files: **C:/ti/TivaWare_C_Series-<version>/grlib/rvmdk**
- Sensor hub library files: **C:/ti/TivaWare_C_Series-<version>/sensorlib/rvmdk**
- USB library files: **C:/ti/TivaWare_C_Series-<version>/usblib/rvmdk**
- All Utilities function files: **C:/ti/TivaWare_C_Series-<version>/utils**
- Compiler-specific output directories: C:/ti/TivaWare_C_Series-<version>/driverlib/rvmdk
- Loader-related codes, include source code and header files: **C:/ti/TivaWare _ C_ Series-<version>/boot_loader**
- All example program files: **C:/ti/TivaWare_C_Series-<version>/examples/boards/ek-tm4c123gxl**
- All related documents or user guide files: **C:/ti/TivaWare_C_Series-<version>/docs**
- All related tools files: **C:/ti/TivaWare_C_Series-<version>/tools**

The **<version>** tab used in these folders is the actual version of the software package installed in your host computer. In our case, it is **2.1.4.178**, which means that the **<version>** should be replaced by **2.1.4.178.**

There are some additional Utilities provided by TivaWare™ for C Series software, which include the LMFlash Programmer, the UniFlash, the FTDI Drivers, IQMath Library and CMSIS Support. Since the CMSIS Support is major utility software widely used to support CMSIS package, we will take more attention to this topic.

3.3.4.2 TivaWare™ for C Series CMSIS Support

Texas Instruments, with the C Series of ARM® Cortex®-M microcontrollers (MCUs), supports the ARM® Cortex® Microcontroller Software Interface Standard (CMSIS), a standardized hardware abstraction layer for the Cortex-M processor series. The CMSIS enables consistent and simple software interfaces to the processor core and simple basic MCU peripherals for silicon vendors and middleware providers, simplifying software reuse, reducing the learning curve for new microcontroller developers and reducing the time to market for new devices.

The CMSIS-DSP library includes source code and example applications, and saves time by including common DSP algorithms such as complex arithmetic, vector operations, and filters, and control functions. The ARM® Cortex®-M4 core uses the DSP SIMD instruction set and floating-point hardware that enhances the Tiva™ C Series microcontroller algorithm capabilities for digital signal control applications.

A standardized software interface allows developers to make the switch from a competitive MCU to TI's Tiva™ C Series microcontrollers and more easily migrate the existing software to any C Series microcontroller.

Many microcontroller-based applications can benefit from the use of an efficient digital signal processing (DSP) library. To that end, ARM® has developed a set of functions called the CMSIS-DSP library that is compatible with all Cortex®-M3 and M4 processors and that is specifically designed to use ARM® assembly instructions to quickly and easily handle various complex DSP functions. Currently, ARM® supplies example projects for use in their Keil® µVision IDE that are meant to show how to build their CMSIS-DSP libraries and run them on an M3 or M4.

The basic idea of using this TivaWare™ for C Series support to CMSIS is to develop a set of interface functions to access the CMSIS-DSP Library files developed by the ARM® in the Keil® ARM-MDK environment from the TivaWare™ for C Series evaluation boards.

A complete CMSIS-DSP Library is underdeveloped by the TI and some application examples can be found in the CCS IDE for TivaWare™ for C Series evaluation boards

An illustration block diagram that contains complete components, including hardware and software, for our project development system is shown in Fig. 3.27.

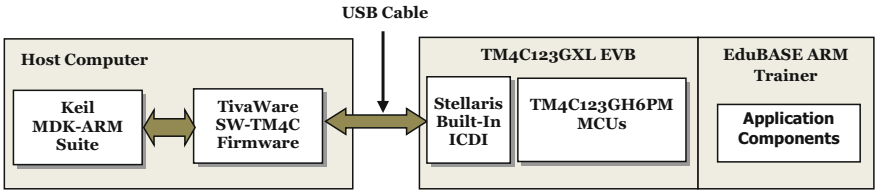


Fig. 3.27 A complete development system for our project

3.3.5 Tiva™ C Series LaunchPad™ Debug Adaptor and Debug Adaptor Driver

Since we use the Tiva™ C Series LaunchPad evaluation board TM4C123G with this book, therefore we need to emphasize the specified debugger used in this system.

The Tiva™ C Series LaunchPad evaluation board comes with an on-board In-Circuit Debug Interface (ICDI). This interface is mainly composed of a TM4C123GH6PM microcontroller unit (MCU), and it can be considered as a debug controller with a debug adapter. Therefore, we do not need to use any other debug adapter since it has been built on this evaluation board.

As you know, there are two pieces of MCU units, TM4C123G6PM, on the Tiva™ C Series LaunchPad evaluation board TM4C123G. The one located at the top of this EVB works as a debug controller/adapter. This debugger, combined with other related debug elements in this EVB, is called an ICDI interface.

The ICDI allows for the programming and debug of the TM4C123GH6PM using the LM Flash Programmer and/or any of the supported tool chains. Note that the ICDI supports only JTAG debugging. An external debug interface can be connected for Serial Wire Debug (SWD) and SWO (trace).

To access the on-board ICDI, a debug adapter driver is needed. A good choice is the Stellaris® ICDI Driver. Depending on the different operating systems the user used, different debug drivers are provided by Stellaris®. Refer to Appendix C to download and install this debug driver on your host computer if you have not done it.

One point to be noted is that the Stellaris® ICDI driver can only work for the JTAG interface.

An illustration block diagram of using the Keil® ULINK2 and Stellaris® ICDI debug interfaces and adapters for the different MCU boards are shown in Fig. 3.28.

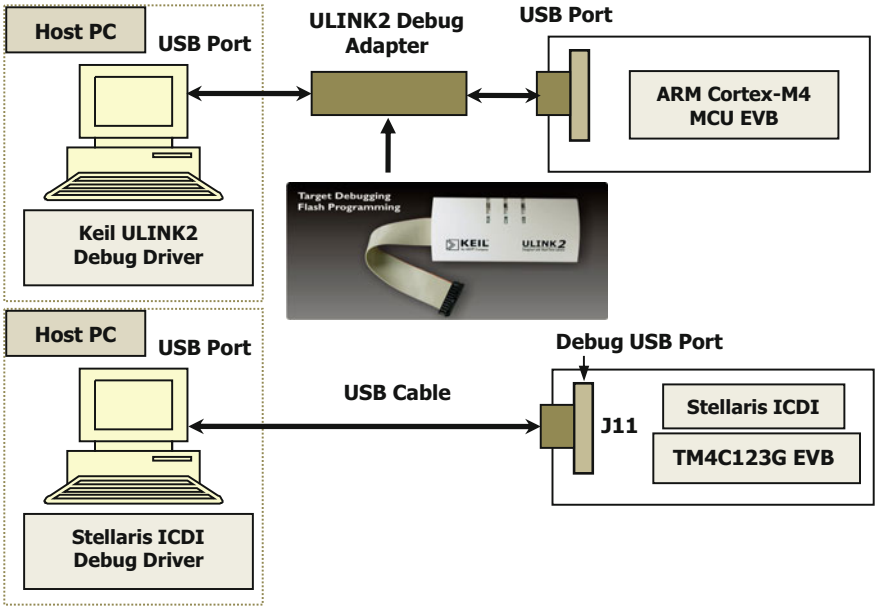


Fig. 3.28 The debug system connections for ULINK2 and Stellaris®

3.4 Development Steps to Build a Basic Project with Keil® MDK µVersion5 IDE

In Sect. 3.3.3.3, we have discussed a simple example project **Hello World** to illustrate how to build a project with Keil ARM-MDK µVersion5. In this section, we will provide more detailed introduction and discussion to show users how to build an example project **MyProject** with Keil ARM-MDK IDE and TivaWare™ C Series LaunchPad™ Evaluation Software Package.

Make sure the TM4C123GXL EVB is connected to your host computer with a USB cable, as shown in Fig. 3.1. Following steps below to build and run this project

(1) Create a new folder **MyProject** on your **C:/** drive with Windows Explorer.
(2) Open the Keil ARM-MDK µVersion5 IDE by double clicking the Keil µVersion5 icon on your desktop.
(3) Click on the **Project|New µVersion Project** item and select the folder **MyProject** created at step 1, and enter the project name, **MyProject** (Fig. 3.29). The file extension is *.**uvproj** for MDK µVersion4, or *.**uvprojx** for µVersion5. Click on the **Save** button to save this new project. It is good practice to use a separate folder for each project.
(4) The **Select Device for Target 'Target1'** wizard is displayed after this new project is saved, as shown in Fig. 3.30. Expand the folders, **Texas**

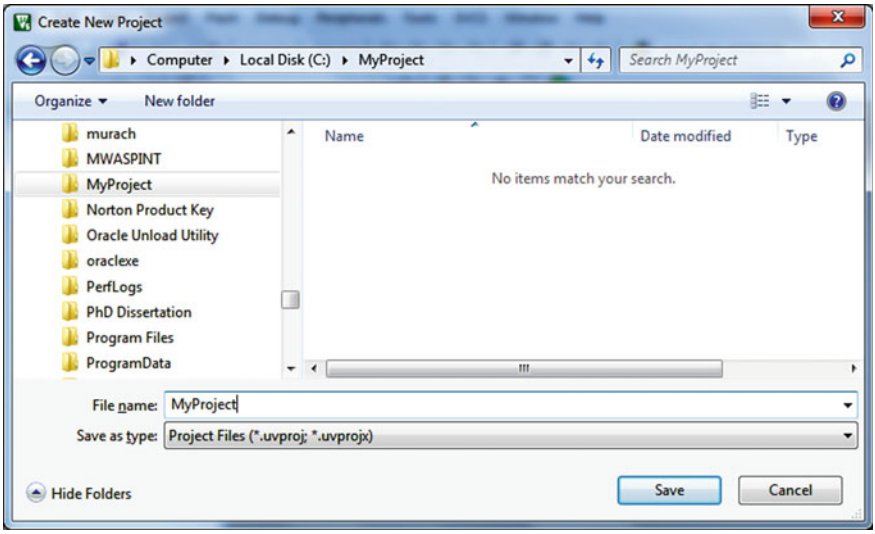


Fig. 3.29 Create a new µVersion5 project MyProject

Instruments—Tiva C Series—TM4C123x Series, and select our device, **TM4C123GH6PM**, from the list, as shown in Fig. 3.30. This selection defines essential tool settings such as compiler controls, the memory layout for the linker, and the Flash programming algorithms. Click on the **OK** button to continue.

(5) Immediately the **Manage Run-Time Environment (RTE)** wizard is shown, as shown in Fig. 3.31, after a new project is created. The **RTE** wizard allows you to manage Software Components of this new project. Software Components can be added, deleted, disabled, or updated during the software development process at any time. You can also open this wizard by going to the menu **Project|Manage|Run-Time Environment**.

(6) Now let us expand the following nodes since we need to use some components and need to setup some of them in this **Manage Run-Time Environment** wizard:

- CMSIS
- Device

For a basic user application, we only need the MDK Core and CMSIS Core. Select and setup the following components by checking the related checkbox in the **Sel** column

- **CMSIS:CORE**
- **Device:Startup**

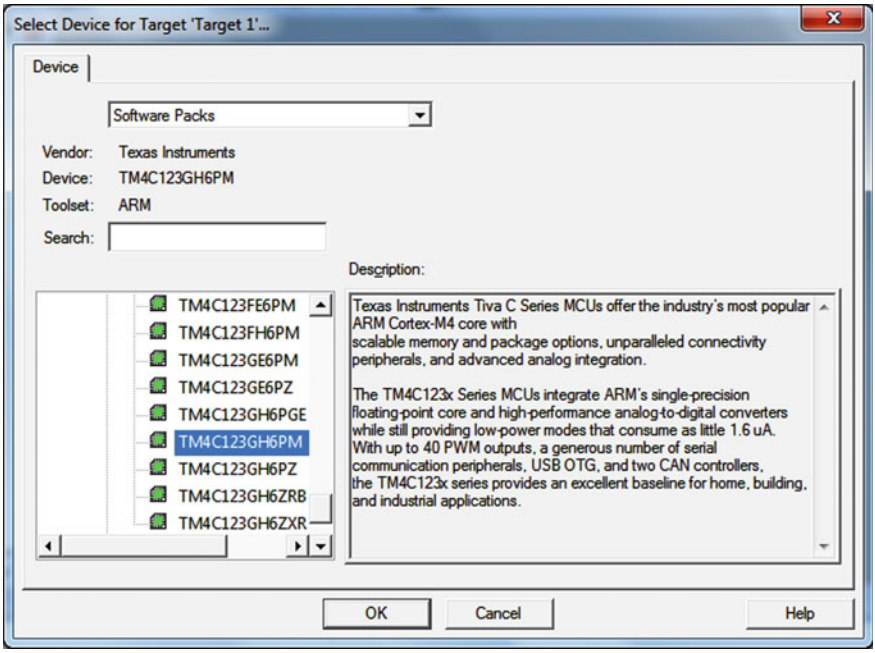


Fig. 3.30 The Select Device for Target wizard (Reproduced with permission from Arm® Limited. Copyright© Arm Limited)

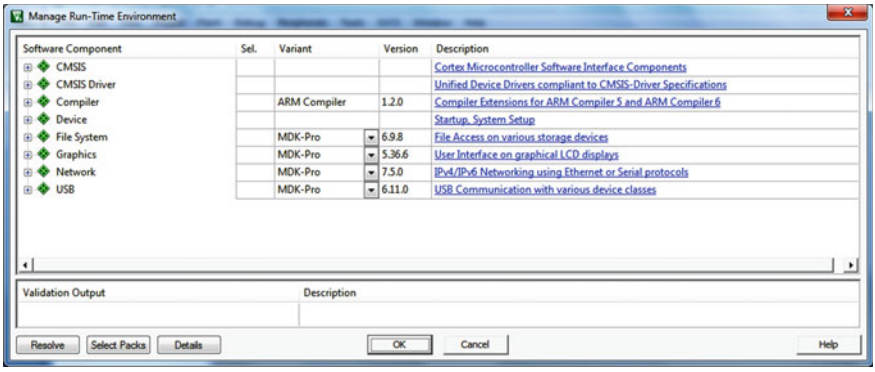


Fig. 3.31 The Manage Run-Time Environment wizard (Reproduced with permission from Arm® Limited. Copyright© Arm Limited)

(7) Your finished setup and selection for this **Manage Run-Time Environment** wizard should match one that is shown in Fig. 3.32.
One point to be noted is that different colors may be displayed for the different selected components when you do this setup and selection. Also the detailed information will be displayed in the Validation Output window in the bottom

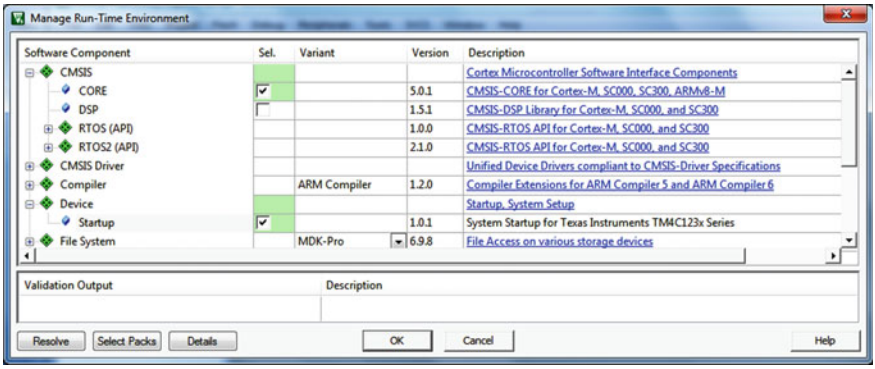


Fig. 3.32 The selected components in the Manage RTE wizard (Reproduced with permission from Arm® Limited. Copyright© Arm Limited)

of this wizard to indicate some missed or required components you need to get to complete these setups and selections.

- **Green color**: The Software Component has been resolved or a Software Component allowing multiple instances has been resolved. Nothing has been displayed in the Validation Output window.
- **Yellow color**: This Software Component is unresolved. Other components are required for correct operation and are listed in the Validation Output window.
- **Red color**: The Software Component conflicts with other components or is not installed on the computer. Detailed information is listed in the Validation Output window.

(8) Click on the **OK** button to close this wizard when the setup and selection process is done. In case if you encountered either yellow or red colors, you may need to use the Pack Installer to install those missed components.

(9) Now our new project **MyProject** has been created and necessary environments have been set using the Manage RTE. The MDK GUI looks like one shown in Fig. 3.33.

(10) In MDK µVersion5, the projects are organized in a format called **Target** → **Source Group** → **Source Files**. All projects can be collected together to form a group and put into one group. The default group is **Source Group 1.** You can add additional source files into a source group by double clicking on the selected source group and then use the file browser to add the source file. Different source groups can also be collected together to form a Target, the default target is **Target 1**. You can also add any additional source groups into the Target by right clicking on **Target 1**, and select **Add Group**.

(11) Refer to Fig. 3.33, you can find that the default group, **Source Group 1**, has been added into the **Target 1** under the **Project** pane in the left. To add our project source file, **MyProject.c**, into this default source group, perform the following operations.

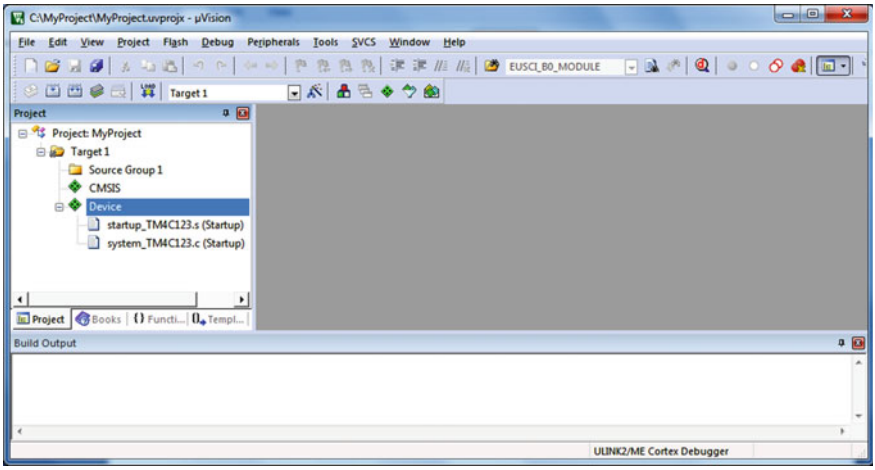


Fig. 3.33 The new project wizard (Reproduced with permission from Arm® Limited. Copyright© Arm Limited)

(12) Right click on the **Source Group 1** and select **Add New Item to Group "Source Group 1"** to open the Add New Item to Group wizard.

(13) Select **C File (.c)** as the template and enter **MyProject** into the **Name** box. Click on the **Add** button to add this source file into our new project.

(14) Add codes shown in Fig. 3.34 into this source file.

(15) Now let us build our project by going to **Project|Rebuild all target files** menu item. The building process begins and the detailed building steps are shown in the Build Output window in the bottom, as shown in Fig. 3.35.

(16) Now let us load our project into the flash memory in the Tiva™ C Series LaunchPad EVB, TM4C123G. This download process needs a debug adaptor and related driver. Before you can do this, you need to double check the debug driver used for the debugger provided by the MDK. Remember a related debug driver called ICDI has also been installed in our host computer in Sect. 3.3.2. Therefore, we can use this debug driver to save our cost. Perform the following operations to finish this debug driver checking:

- Go to **Project|Options for Target "Target 1"** item to open this wizard.
- Click on the **Debug** tab to open its settings.
- On the upper right of this wizard, click on the dropdown arrow for the **Use** combobox, and select **Stellaris ICDI** from the list.

Your finished wizard should match one that is shown in Fig. 3.36. Click on the **OK** button to close this wizard.

(17) Select the **Download** item from the **Flash** menu, or click the **Download** button (icon). The process takes a few seconds. A progress bar will show at the bottom of the IDE window as the device is programmed. When it is finished,

```
//*****************************************************************
// Blink the on-board LED in TM4C123G.
//*****************************************************************
#include <stdint.h>

#define SYSCTL_RCGC2_R        (*((volatile uint32_t *)0x400FE108))
#define SYSCTL_RCGC2_GPIOF     0x00000020  // Port F Clock Gating Control

//*****************************************************************
// GPIO registers (PORTF)
//*****************************************************************

#define GPIO_PORTF_DIR_R      (*((volatile uint32_t *)0x40025400))
#define GPIO_PORTF_DEN_R      (*((volatile uint32_t *)0x4002551C))
#define GPIO_PORTF_DATA_R     (*((volatile uint32_t *)0x400253FC))

Int main(void)
{
  volatile uint32_t ui32Loop;
  // Enable the GPIO port that is used for the on-board LED.
  SYSCTL_RCGC2_R = SYSCTL_RCGC2_GPIOF;
  // Do a dummy read to insert a few cycles after enabling the peripheral.
  ui32Loop = SYSCTL_RCGC2_R;
  // Enable the GPIO pin for the LED (PF3).  Set the direction as output, and
  // enable the GPIO pin for digital function.
  GPIO_PORTF_DIR_R = 0x08;
  GPIO_PORTF_DEN_R = 0x08;
   while(1)                                  // Loop forever.
  {
    GPIO_PORTF_DATA_R |= 0x08;          // Turn on the LED.
    for(ui32Loop = 0; ui32Loop < 200000; ui32Loop++)    // Delay for a bit.
    {
    }
    GPIO_PORTF_DATA_R &= ~(0x08);      // Turn off the LED.
    for(ui32Loop = 0; ui32Loop < 200000; ui32Loop++)    // Delay for a bit.
    {
    }
  }
}
```

Fig. 3.34 The source codes in the MyProject.c file

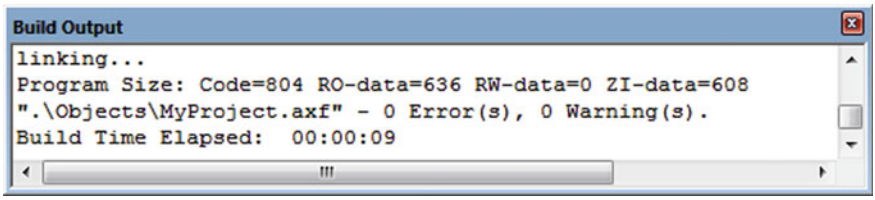


Fig. 3.35 The project building result (Reproduced with permission from Arm® Limited. Copyright© Arm Limited)

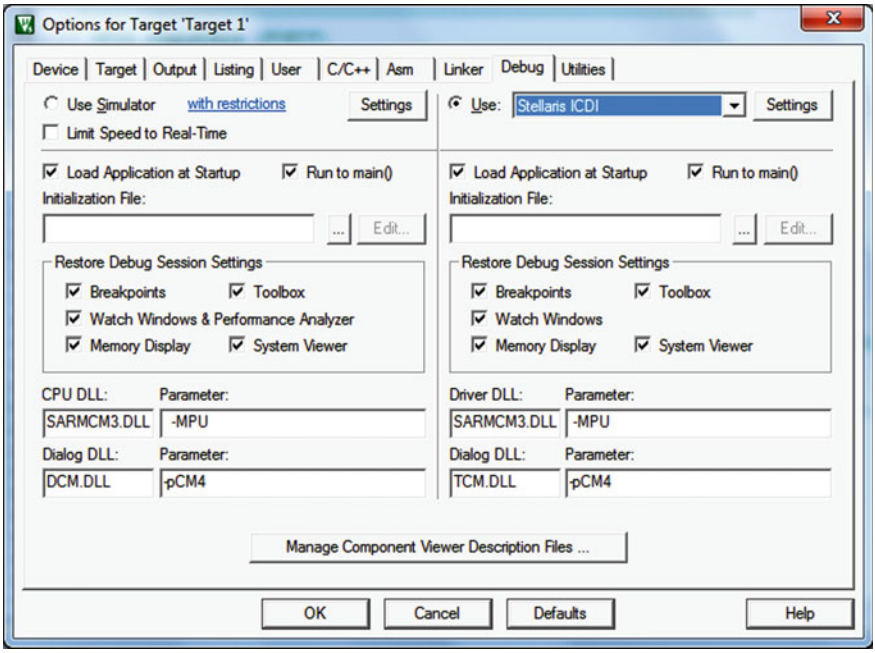


Fig. 3.36 The finished debugger checking wizard (Reproduced with permission from Arm® Limited. Copyright© Arm Limited)

the Build window will show that the device was erased, programmed, and Verified OK. Our project application, **MyProject.axf**, which is an image file, is now programmed and downloaded into the flash memory on the Evaluation Board.

(18) We can now start to debug our project by going to **Debug|Start/Stop Debug Session** menu item. Click on the **OK** button for the popup memory limitation message for the evaluation version to begin this debug process.

(19) To run our project, go to **Debug|Run** menu item to run it. The green color LED in the TM4C123G EVB will be flashed periodically. Click on the **Stop** item from the **Debug** menu to stop our project.

(20) Click on the **Erase** item from the **Flash** menu to erase our project from the flash memory in the TM4C123G EVB.

Based on these operational steps, one can build a microcontroller project in a similar way.

3.5 Summary

In order to successfully build and develop a microcontroller control programming project, one needs a complete set of development kits, which include both hardware and software package. A complete and detailed introduction about the Tiva C Series LaunchPad Evaluation Board development kits are provided in this chapter.

The entire and complete development tools and kits can be categorized into two parts; the ***Hardware Package*** and the ***Software Package***.

The ***hardware package*** includes

- TM4C123GH6PM Microcontroller Unit (MCU).
- Tiva™ C Series LaunchPad™ TM4C123GXL Evaluation Board (EVB).
- EduBASE ARM® Trainer (contains most popular peripherals and interfaces).
- Some other related peripherals and interfaces, such as DC Motors, Pulse Width Modulation (PWM) module, and Quadrature Encoder Interface (QEI).

The ***software package*** includes

- Integrated Development Environment Keil® ARM®-MDK µVersion®5.24a (IDE).
- TivaWare™ SW-EK-TM4C123GXL Software Driver Package.
- Stellaris In-Circuit Debug Interface (ICDI).

Starting from Sect. 3.2.1, the TM4C123GH6PM MCU is introduced in details since it is a core for the TM4C123GXL EVB. The main components involved with this core include the TM4C123GH6PM on-chip memory map and GPIO Modules. Following this introduction, two important hardware components, TM4C123GXL EVB and EduBASE ARM Trainer, are discussed in detailed. The software package is discussed starting from Sect. 3.3, which includes Keil ARM-MDK IDE, TivaWare for C Series software suite, and Stellaris® ICDI Debugger Driver.

At the end of chapter, a detailed description about all steps used to build a microcontroller project under the Keil ARM-MDK IDE and TivaWare software package is provided to help users to build their projects later.

Homework

3.1 Provide a brief description about components to be used to develop a basic microcontroller project.

3.2 Provide a brief description about components used to build a microcontroller project with Keil® MDK µVersion IDE and running in the TivaWare™ for C Series Evaluation Board EK-TM4C123GXL.

3.3 Explain the debug components used in Keil® µVersion5 debugger and TivaWare™ built-in ICDI in the TM4C123G Evaluation Board.

3.4 Explain the relationship between the general-purpose development kit (Keil MDK) and specified TivaWare™ for C Series LaunchPad™ software package.

3.5 Explain the development steps used to build a basic microcontroller project with Keil® MDK μVersion5 IDE.

3.6 Provide a brief description about the GPIO Module in the TM4C123GH6PM MCU system and answer the following questions:

(a) How many GPIO blocks are existed in the TM4C123GH6PM MCU system? What are they? How many bits for each block or port?

(b) What are the initialization steps for those GPIO ports?

Chapter 4
System Mathematical Models and Model Identifications



The objective for a control system is to try to get the desired output and make it as closely as possible to the input reference set value. In order to realize this objective, a mathematical model is necessary to enable designers to obtain the detailed and accurate knowledge about the control target or plant, design, and build an optimal controller to get the desired control performances.

In this chapter, we will discuss most popular mathematical models for most often used systems in our world. Different methods are introduced to set up, analyze, and identify the system models to enable users to successfully design desired control systems based on those models.

4.1 Introduction

A so-called mathematical model is to use mathematical equations to define the input–output relationship for a control system, as completely and accurately as possible. However, in our real world, most or maybe all of systems belong to nonlinear and time-varying systems. Therefore, in order to obtain a desired model, we must make a compromise between the simplicity of the ideal model and the accuracy of results of the real model. The following factors must be considered when we derive any model:

Simplicity Versus Accuracy: To derive a reasonably simplified mathematical model, we frequently find it necessary to ignore certain inherent physical properties of the system. In particular, if a linear lumped-parameter mathematical model is desired, it is always necessary to ignore certain nonlinearities and distributed parameters that may be present in the physical system. If the effects that these ignored properties have on the response is small, good agreement will be obtained between the results of the analysis of a mathematical model and the results of the experimental study of the physical systems.

© Springer Nature Switzerland AG 2019

Y. Bai and Z. S. Roth, *Classical and Modern Controls with Microcontrollers*,
Advances in Industrial Control, https://doi.org/10.1007/978-3-030-01382-0_4

Linear Versus Nonlinear Systems: We have provided detailed discussions about these two kinds of control systems in Chap. 2. The principle of superposition states that the response produced by the simultaneous application of two different forcing functions is the sum of the two individual responses. Hence, for the linear system, the response to several inputs can be calculated by treating one input at a time and adding the results finally to get the terminal outputs. It is this principle that allows one to build up complicated solutions to the linear differential equation from simple solutions. For an experimental investigation of a dynamic system, if cause and effect are proportional, it implied that the principle of superposition holds, then the system can be considered linear.

If a system can be described by linear differential equations, the Laplace transform can be used to get transfer functions.

However, for a nonlinear system, one cannot use the superposition principle to treat different functions in the system. Special analysis methods must be adopted to handle those systems.

Linear Time-Invariant Systems and Linear Time-Varying Systems: As we mentioned in Chap. 2, a linear differential equation can be used if all coefficients are constants or their functions are only independent on variable. Dynamic systems that are composed of linear time-invariant lumped-parameter components may be described by linear time-invariant differential equations, that is, constant-coefficient differential equations. Systems that are represented by differential equations whose coefficients are functions of time are called linear time-varying systems. An example of a time-varying control system is a spacecraft control system.

Basically, the steps to build a mathematical model for a system include the following:

(1) Define the system and its components.
(2) Approximate any nonlinearity to linearity and formulate the mathematical model.
(3) Derive the related differential equations to describe the model.
(4) Solve the equations to get the desired outputs.
(5) Check and examine the solutions.

In the following sections, we will review the most popular and basic principles of modeling for the most common systems, including mechanical, electrical, and electromechanical systems.

4.2 Mathematical Models for Most Popular Systems

Basically, any LTI system can be described by using differential equations that can be obtained by utilizing physical laws governing a particular system, such as Newton's laws for mechanical systems and Kirchhoff's voltage and current laws for electrical systems.

Let us start with mechanical systems.

4.2.1 Models for Mechanical Rotational Systems

Consider a mechanical rotational system shown in Fig. 4.1. The system consists of a load inertia J and a viscous–friction damper f, where

- J = The moment of inertia of the load (kg · m^2)
- f = Viscous–Friction coefficient N(rad/s)
- ω = Angular velocity (rad/s)
- T = Torque applied to the system (N · m)

According to Newton's law, for mechanical rotational systems, it states

$$\sum T = Ja \tag{4.1}$$

where a = angular acceleration (rad/s^2)

Applying Newton's law to this system, we obtain the following differential equation:

$$J\frac{\mathrm{d}\omega}{\mathrm{d}t} + f\omega = T \tag{4.2}$$

Since this is a first-order linear differential equation, after applying the Laplace transform, we have the following transfer function:

$$G(s) = \frac{\Omega(s)}{T(s)} = \frac{1}{Js + f} \tag{4.3}$$

where $\Omega(s) = L\{\omega(t)\}$ and $T(s) = L\{T(\mathrm{t})\}$. The only pole is −f/$J$.

4.2.2 Models for Electrical Circuit Systems

A resistor–inductor–capacitor (RLC) circuit is shown in Fig. 4.2.

After applying the Kirchhoff's voltage law to this RLC circuit system, we obtain the following differential equations:

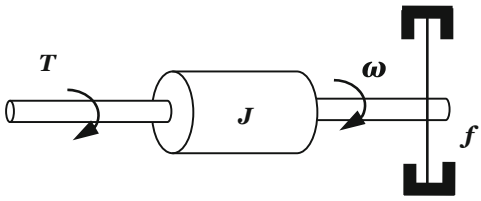


Fig. 4.1 A mechanical rotational system

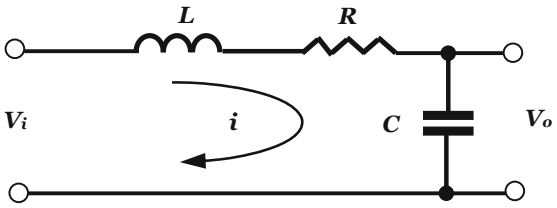


Fig. 4.2 An RLC electrical circuit system

$$L\frac{\mathrm{d}i}{\mathrm{d}t} + Ri + \frac{1}{C}\int i\mathrm{d}t = V_i \tag{4.4}$$

$$V_O = \frac{1}{C}\int i\mathrm{d}t \tag{4.5}$$

These two differential equations make up the mathematical model for this RLC circuit. Since these are linear differential equations, after applying the Laplace transform and assuming zero initial conditions, we get the transfer function for this system, which is

$$LsI(s) + RI(s) + \frac{1}{C}\frac{1}{s}I(s) = V_i(s) \tag{4.6}$$

$$V_O(s) = \frac{1}{C}\frac{1}{s}I(s) \tag{4.7}$$

The transfer function is

$$G(s) = \frac{V_O(s)}{V_i(s)} = \frac{1}{LCs^2 + RCs + 1} \tag{4.8}$$

4.2.3 Models for Electromechanical Systems

A DC motor is a power actuator device used to transfer or convert direct current electrical energy into the rotational mechanical energy. A DC motor directly provides rotary motion and, coupled with wheels or drums and cables, can provide translational motion. The electric equivalent circuit of the armature and the free-body diagram of the rotor are shown in Fig. 4.3.

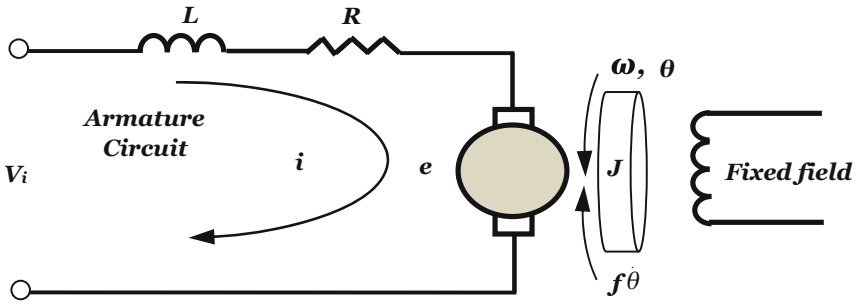


Fig. 4.3 A DC motor structure and its equivalent circuit

Most popular DC motors are field current controlled motors and if the field current is a fixed or constant value, it provides a constant magnetic field, and thus it is called an armature-controlled motor. In this case, the motor torque is only proportional to the armature current i by a constant factor K_m as shown in Eq. (4.9).

$$T = K_m i \tag{4.9}$$

The back electromotive force *emf* or e is proportional to the motor rotation speed or the angular velocity of the shaft by a constant factor K_e.

$$e = K_e \frac{\mathrm{d}\theta}{\mathrm{d}t} \tag{4.10}$$

Regularly, in the SI units, the motor torque and back *emf* constants should be equal, which means, $K_m = K_e$; therefore, we can use K to represent both the motor torque constant and the back *emf* constant.

Based on Fig. 4.3, we can derive the following governing equations based on Newton's second law and Kirchhoff's voltage law:

$$J\frac{\mathrm{d}^2\theta}{\mathrm{d}t^2} + f\frac{\mathrm{d}\theta}{\mathrm{d}t} = Ki \tag{4.11}$$

$$L\frac{\mathrm{d}i}{\mathrm{d}t} + Ri = V_i - K\frac{\mathrm{d}\theta}{\mathrm{d}t} \tag{4.12}$$

Applying the Laplace transform to the above two equations, we get the following functions for this system:

$$s(Js+f)\,\theta(s) = KI(s) \tag{4.13}$$

$$(Ls+R)I(s) = V_i(s) - Ks\theta(s) \tag{4.14}$$

Combining Eqs. (4.13) and (4.14) to eliminate the intermediate variable I(s), we get the transfer function for this DC motor system as

$$G(s) = \frac{\Omega(s)}{V_i(s)} = \frac{K}{(Js+f)(Ls+R)+K^2} \quad \left(\frac{\text{rad/s}}{\text{V}}\right) \tag{4.15}$$

where $\dot{\theta}(s) = s\theta(s) = \Omega(s)$ and it is the rotation speed of the motor (ω) measured by rad/s.

This is a second-order system with two poles and no zeros, and it is a typical transfer function used for most linear systems.

4.3 Relationship Between Poles and Step Response

Generally, a transfer function represented for a second-order system can be rewritten in a standard format as

$$G(s) = K\frac{\omega_n^2}{s^2 + 2\zeta\omega_n s + \omega_n^2} \tag{4.16}$$

With this transfer function, we have the following relationship among different parameters:

$$\sigma = \zeta\omega_n \tag{4.17}$$

$$\omega_d = \omega_n\sqrt{1-\zeta^2} \tag{4.18}$$

where the parameter $\boldsymbol{\xi}$ is called the **damping ratio**, $\boldsymbol{\omega_n}$ is called the **undamped natural frequency**, and ω_d is actual frequency. The poles of this transfer function are located at a radius ω_n in the s-plane and at an angle $\theta = \sin^{-1}\xi$, as shown in Fig. 4.4.

In rectangular coordinates, the poles are at $s = -\sigma - j\omega_d$, where $\sigma = \xi\omega_n$ and $\omega_d = \omega_n\sqrt{1-\xi^2}$. When $\xi = 0$, we have no damping and the damped natural frequency $\omega_d = \omega_n$, the undamped natural frequency. Therefore, the damping ratio reflects the level of damping as a fraction of the critical damping value where the poles become the real poles.

In order to get the relationship between a complex transfer function shown in (4.16) and the corresponding step response in the time domain, the (4.16) can be rewritten as ($K = 1$)

$$G(s) = \frac{\omega_n^2}{(s+\zeta\omega_n)^2 + \omega_n^2(1-\xi^2)} \tag{4.19}$$

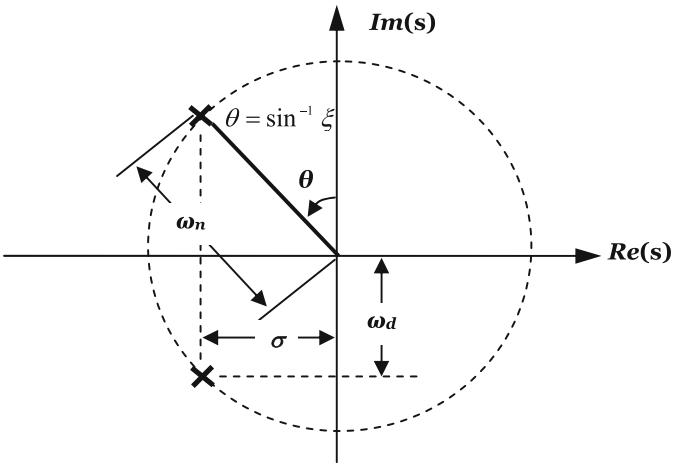


Fig. 4.4 The *s*-plane configuration for a pair of complex poles

The step input signal can be presented as a unit input $u = 1(t)$ and its Laplace transformation is $U(s) = 1/s$. Therefore, the step response represented in the frequency domain is $Y(s) = G(s)U(s)$.

$$Y(s) = G(s)\,U(s) = \frac{1}{s}\frac{\omega_n^2}{(s+\zeta\omega_n)^2 + \omega_n^2(1-\xi^2)} \tag{4.20}$$

Using the inverse Laplace transform, we can obtain the step response in the time domain

$$y(t) = 1 - e^{-\sigma t}(\cos\omega_d t + \frac{\sigma}{\omega_d}\sin\omega_d t) \tag{4.21}$$

Figure 4.5 shows a plot of step response $y(t)$ for some different damping ratio values from 0 to 2. The undamped natural frequency is time normalized to radian degrees. It can be found from Fig. 4.5 that the actual frequency ω_d will be reduced as the damping ratio is increased. When the damping ratio ξ is less than one, the step response is oscillatory, but for large damping ratio or when ξ is closer to one, the step response shows no oscillation at all. This makes sense since $\omega_d = \omega_n\sqrt{1-\xi^2}$.

Recall that we have discussed the effect of poles locations in *s*-plane on the step responses in Fig. 2.9 at Sect. 2.6 in Chap. 2. You can refer to that section to get more details about this effect. In fact, the negative real part of the pole, σ, determines the decay rate of an exponential envelope that multiplies the sinusoidal response. This means that a pole has a negative real part if σ is positive since complex poles always come in complex conjugate pairs for real polynomials, and any complex poles can be described in terms of their real and imaginary parts as

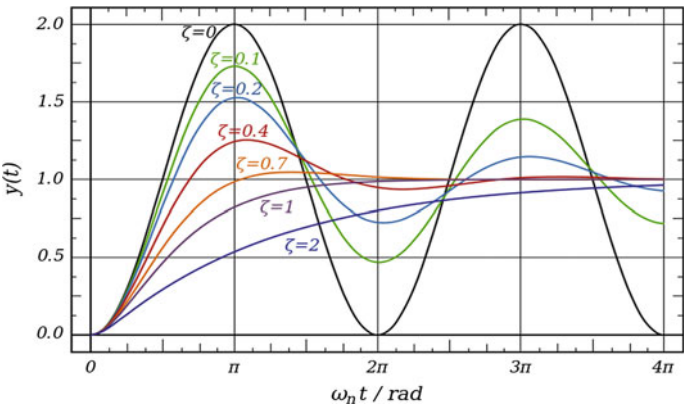


Fig. 4.5 Step response for a DC motor system (second-order system)

$$s = -\sigma \pm j\omega_d \tag{4.22}$$

Let us have a closer look at the step response and related specifications in the time domain.

4.3.1 Time-Domain Specifications for Step Responses

Even we have a clear picture about the relationship between the poles locations in the *s*-plane and the step response for a second-order system such as a DC motor; however, we still need to get a little more details about the step response in the time domain, such as rise time, overshoot and settling time, since these specifications are directly related to a closed-loop system design process. Figure 4.6 shows a graphical illustration for these specification parameters for a standard step response. The definitions of these parameters are shown below:

- Rise time t_r—The time period it takes the step response from its 10 to 90% of the final value.
- Settling time t_s—The time it takes the step response from the transients to decay.
- Peak time t_p—The time it takes the step response from 0 to its peak value.
- Overshoot M_p—The difference between the peak value and the final value.

In Fig. 4.6, the steady-state error is defined $\pm 1\%$ and this error could be a different value that depends on the actual control systems.

To speed up and simplify the control system designs, some approximation methods have been reported to set up relationships among these specifications. One of the most commonly used methods is especially useful for the second-order system without zeros, which is

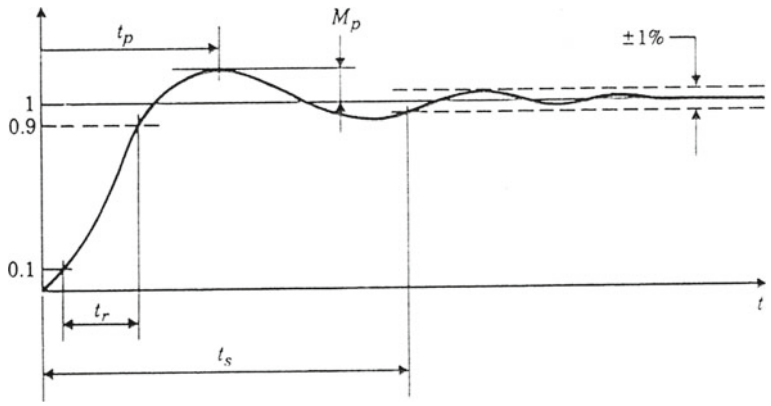


Fig. 4.6 Definition of rise time, settling time, and overshoot for a step response

$$t_r \approx \frac{1.8}{\omega_n} \tag{4.23}$$

$$t_s \approx \frac{4.6}{\xi\omega_n} = \frac{4.6}{\sigma} \tag{4.24}$$

$$M_p \approx e^{-\pi\xi/\sqrt{1-\xi^2}} \quad (0 \le \xi \le 1) \quad M_p \approx \left(1 - \frac{\xi}{0.6}\right) \quad (0 \le \xi \le 0.6) \tag{4.25}$$

For the overshoot M_p, two often used values are $M_p = 16\%$ for $\xi = 0.5$ and $M_p = 5\%$ with $\xi = 0.7$.

One point to be noted is that Eqs. (4.23)–(4.25) are only used for any second-order LTI system having no finite zeros and two complex conjugate poles with undamped natural frequency ω_n, damping ratio ξ, and negative real part σ. Also, these estimations are only used to obtain a set of rough or approximated values for rise time t_r, settling time t_s, and overshoot M_p for an LTI system. Thus, it is important to keep in mind that they are only qualitative guides and not accurate design formulas, and can only work as a starting point for the design and the final time response should always be double checked after the control design is completed by a numerical simulation to verify the final control objectives.

Figure 4.7 shows the step response for our DC motor open-loop system shown in Eq. (4.15). It can be found that this is an underdamped system with the rise time $t_r = 0.027$ s and settling time $t_s = 0.05$ s. The motor parameters are selected from a typical DC motor, which is shown in Table 4.1.

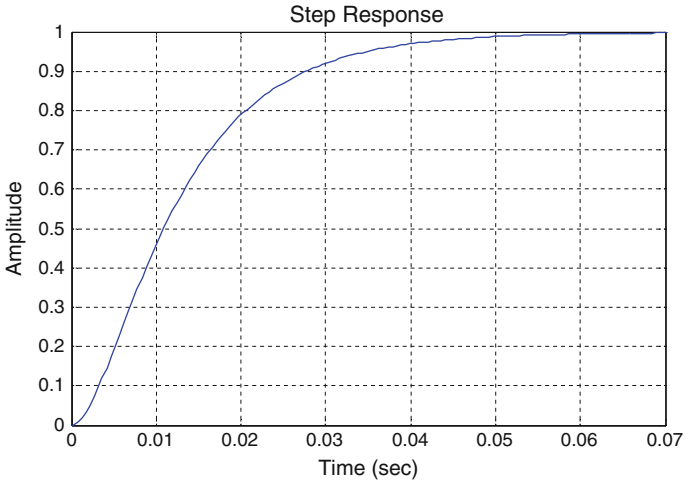


Fig. 4.7 The step response of a DC motor system (open loop)

Table 4.1 Errors versus system type for unity feedback system

Parameter	Unit	Value
R	Ω	0.19
L	H	0.0005
K	V_s	0.0323
J	kg m^2	$7.5e^{-5}$
f	Mm s	$2e^{-5}$

4.4 Some Important Properties Related to System Models

There are some other important properties used to describe a closed-loop control system, and some properties are related to step responses and very useful in design and analysis.

4.4.1 The Final Value Theorem

The final value theorem sets a relationship for the system output between the frequency domain and the time domain. For example, we have a Laplace transform of a system output $Y(s)$ that is related to an actual output in the time domain $y(t)$, and we like to know the final value $y(t)$ from the $Y(s)$.

Three possible results can be obtained; constant, undefined, or unbounded, as shown below:

(1) If $Y(s)$ has any pole in the right-hand side of s-plane or multiple poles on the imaginary axis, the $y(t)$ will grow and the limit is unbounded.
(2) If all poles are in the left-hand side of s-plane except for a simple pair on the imaginary axis, the $y(t)$ will contain a sinusoid that persists forever and the final value is not defined.
(3) If all poles are in the left-hand side of s-plane, the $y(t)$ will decay to zero except for the term corresponding to the pole at the origin ($s = 0$), and that term is a constant. Therefore, the final value is given by the residue of the pole at the origin.

Based on these results, we can conclude it as a theorem.
If all poles of the product $sY(s)$ are in the left-hand side of s-plane, then

$$\lim_{t\to\infty} y(t) = \lim_{s\to 0} sY(s) \tag{4.26}$$

The final value theorem is a very useful tool to determine the final value if its Laplace transform is given since we can get it from the frequency domain.

4.4.2 Steady-State Errors

As we have discussed in Chap. 2, a standard closed-loop control system can be described by a block diagram shown in Fig. 4.8.

Now let us consider a special case, where the feedback path $H(s) = 1$, then this closed-loop control system can be represented by

$$Y(s) = D(s)G(s)E(s) \tag{4.27}$$

$$E(s) = R(s) - H(s)Y(s) = R(s) - Y(s) \tag{4.28}$$

Putting (4.27) into (4.28), we have

$$E(s) = R(s) - Y(s) = R(s) - D(s)G(s)E(s) \tag{4.29}$$

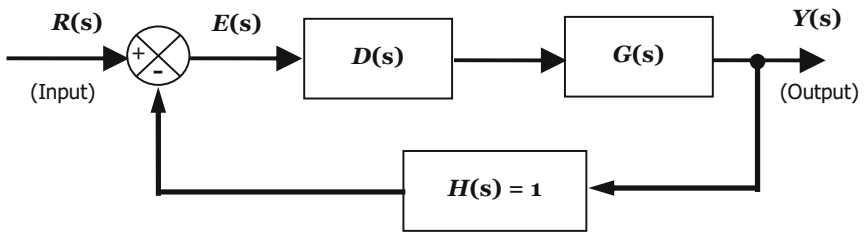


Fig. 4.8 Block diagram of a typical closed-loop control system

If we consider the $E(s)$ as an output, then output to input ratio is

$$\frac{E(s)}{R(s)} = \frac{1}{1 + D(s)G(s)} = S(s) \tag{4.30}$$

Equation (4.30) defined the ratio of the error output to the input as the **sensitivity** $S(s)$.

When the input $r(t)$ is a step signal ($R(s) = 1/s$) and the system is stable, the final value theorem tells us that the steady-state error e_{ss} is

$$e_{ss} = \frac{1}{1 + K_p} \tag{4.31}$$

where

$$K_p = \lim_{s \to 0} D(s)G(s) \tag{4.32}$$

The K_p is a constant and it can be called the position-error constant. If the product $D(s)G(s)$ has a denominator that does not have s as a factor, both K_p and e_{ss} are finite. This kind of system is called a **type 0** system.

If there is an integrator in either $D(s)$ or $G(s)$, the steady-state gain will be infinitive large and the error will be zero. Thus, we can define the velocity constant as

$$K_v = \lim_{s \to 0} sD(s)G(s) \tag{4.33}$$

and the acceleration constant as

$$K_a = \lim_{s \to 0} s^2 D(s)G(s) \tag{4.34}$$

For a system, if its K_v is finite, the system is called a **type 1** system; similarly, if K_a is finite, the system is called a **type 2** system. For the unity feedback control system shown in Fig. 4.8, it can categorize the error styles for command inputs consisting of step, ramp, and parabola. Table 4.2 summarized these results.

Table 4.2 Errors versus system type for unity feedback system

	Step	Ramp	Parabola
Type 0	$\frac{1}{1+K_p}$	∞	∞
Type 1	0	$\frac{1}{K_v}$	∞
Type 2	0	0	$\frac{1}{K_a}$

4.4.3 Stability

As we have discussed in Chap. 2, the open-loop gain for a closed-loop control system is defined as $D(s)G(s)$ and, in fact, this open-loop gain is determined by the poles of $D(s)G(s)$. For a unity feedback closed-loop control system (H(s) = 1), the transfer function can be written as

$$\frac{Y(s)}{R(s)} = \frac{D(s)G(s)}{1 + D(s)G(s)} = \tau(s) \tag{4.35}$$

The $\tau(s)$ is defined as the **complementary sensitivity**. For this unity feedback control system, the dynamic properties and stability are determined by the poles of the closed-loop transfer function, which is determined by the roots of

$$1 + D(s)G(s) = 0 \tag{4.36}$$

Equation (4.36) is very important and it is called the **characteristic equation**. The roots of this characteristic equation represent the types of motion that will be shown by the feedback system. In fact, the roots of Eq. (4.36) are poles of the system and these poles play a major role in the system's stability. Depending on the locations of these poles, three different types of responses may be possible. If a pole is given by

$$s = \sigma + j\omega \tag{4.37}$$

- Response 1: $\sigma < 0$ and $|\omega| < |\sigma|$, the system response is overdamped and stable.
- Response 2: $\sigma \leq 0$ and $|\omega| > |\sigma|$, the system response is underdamped with an overshoot, but the system is still stable.
- Response 3: $\sigma > 0$, the system response is diverged and the system is unstable.

When analyzing the stability of a closed-loop control system, different methods are available. For LTI systems, Bode plot and Nyquist stability criterion are most powerful methods to be used. But the Bode plot is relatively simple. For nonlinear systems, Lyapunov or the circle criterion methods can be adopted.

For a LTI system, make the open-loop gain unity and then check the phase margin. The system would be stable if the phase margin is less than $-180°$. Otherwise, the system is unstable. We will discuss more details about the stability in Chap. 5.

Now let us discuss some popular methods used to evaluate whether an LTI system is stable.

4.4.3.1 Bounded-Input-Bounded-Output Stability

For a LTI system, if all bounded inputs result in bounded outputs, the system is said to have bounded-input-bounded-output (BIBO) stability. In other words, if a system has its impulse response $h(t)$ for a unit input $u(t)$, the output $y(t)$ will be

$$y(t) = \int_{-\infty}^{\infty} h(\tau)u(t-\tau)\mathrm{d}\tau \tag{4.38}$$

If $u(t)$ is bounded, there will be a constant M to make $|u(t)| < M < \infty$, the output is also bounded by

$$|y(t)| = \left| \int h(\tau)u(t-\tau)\mathrm{d}\tau \right| \leq M \int_{-\infty}^{\infty} |h(\tau)|\mathrm{d}\tau \tag{4.39}$$

Thus, if the integration of the impulse response $h(t)$ is bounded, the output will be bounded by the above equation. Otherwise, the output is not bounded.

Based on this discussion, we can conclude that the system with impulse response $h(t)$ is BIBO stable if and only if $\int_{-\infty}^{\infty} |h(\tau)|\mathrm{d}\tau < \infty$.

A typical example for a non-BIBO stable system is a capacitor charging circuit with a current source. Based on the common sense, the final voltage across the capacitor will be bounded because of the capacitor's capacity. However, the integration of the impulse response is not bounded. Therefore, this capacitor charging circuit is not a BIBO stable system.

4.4.3.2 Routh's Stability Criterion

The **Routh–Hurwitz stability criterion** is a mathematical method that is a necessary and sufficient condition for the stability of an LTI control system. This method can be used to check whether an LTI system is stable or not based on the roots' polarity of the characteristic equation of that LTI system.

For example, we have an LTI system with the following characteristic equation of n-order

$$P(s) = a_n s^n + a_{n-1}s^{n-1} + a_{n-2}s^{n-2} + \cdots + a_1 s + a_n \tag{4.40}$$

This is exactly an n-order polynomial equation. By applying Routh's stability criterion, we can check whether this system is stable without exactly solving this characteristic equation. Let us start from a second-order system.

For a second-degree polynomial, $P(s) = s^2 + a_1 s + a_0$, if all the roots are in the left half plane (LHP), the system with characteristic equation $P(s) = 0$ is stable if and only if both coefficients satisfy $a_i > 0$.

For a third-order polynomial $P(s) = s^3 + a_2 s^2 + a_1 s + a_0$, the conditions are that a_2 and a_0 must be positive and $a_2 a_1 > a_0$.

In general, for higher order systems, the Routh stability criterion states that all first-column elements of the Routh array are to be of the same sign. Systems meeting the above criteria are said to be open-loop stable. Otherwise, they are unstable because there are sign changes in the first-column elements in the Routh array. In order to handle the higher order systems, let us have a closer look at the Routh array.

For an *n*th-degree polynomial shown in (4.40), a Routh array can be built with the specific format as shown in Table 4.3a. This table can be called a Routh array or a Routh table, and the table has $n + 1$ rows and $n - 2$ columns.

The first row, row n is composed of all even coefficients starting from a_n in a decrement order. The second row, row $n - 1$, is composed of odd coefficients starting from a_{n-1} also in a decrement order. Starting from the third row, row $n - 2$, and fourth row, row $n - 3$, all coefficients on both rows, b_i and c_i, are calculated based on the coefficients from the previous rows.

The used calculation equations are

$$b_i = \frac{a_{n-1}a_{n-2i} - a_n a_{n-2i-1}}{a_{n-1}} \tag{4.41}$$

$$c_i = \frac{b_1 a_{n-2i-1} - a_{n-1} b_{i+1}}{b_1} \tag{4.42}$$

Table 4.3 *a* The Routh array. *b* Another routh array

a				
row n	a_n	a_{n-2}	a_{n-4}	…
row $n - 1$	a_{n-1}	a_{n-3}	a_{n-5}	…
row $n - 2$	b_1	b_2	b_3	…
row $n - 3$	c_1	c_2	c_3	…
…	…	…	…	…
row 1	a_1			
row 0	a_0			

b				
row n	1	a_2	a_4	…
row $n - 1$	a_1	a_3	a_5	…
row $n - 2$	b_1	b_2	b_3	…
row $n - 3$	c_1	c_2	c_3	…
row $n - 4$	d_1	d_2	d_3	…
…	…	…	…	…
row 1	a_1			
row 0	a_0			

According to the Routh's stability criterion, only all coefficients in the first column on the Routh array or Routh table are important to check if a system is stable or not. The criterion is:

If all coefficients in the first column are positive, the system is stable. Otherwise, the system is unstable. The number of sign changes in the first column is equal to the number of roots in the right half plane (RHP). The number of sign changes including from positive to negative, and from negative change back to positive. For example, one coefficient in the first column changes from + to −, and then another coefficient changes back from − to +, this is considered as two (2) times change on the sign. Thus, there are two (2) roots in the RHP.

In order to simplify the calculations for b_i and c_i in (4.41) and (4.42), a modified version of Routh array or table is provided and is shown in Table 4.3b, in which the characteristic equation or polynomial in (4.40) can be expressed as

$$P(s) = s^n + a_1 s^{n-1} + a_2 s^{n-2} + \cdots + a_{n-1} s + a_n \tag{4.43}$$

Each coefficient and its related s^n item can be arranged as the following format:

row n	s^n	1	a_2	a_4	...
row $n-1$	s^{n-1}	a_1	a_3	a_5	...
row $n-2$	s^{n-2}	b_1	b_2	b_3	...
row $n-3$	s^{n-3}	c_1	c_2	c_3	...
row $n-4$	s^{n-4}	d_1	d_2	d_3	...
...					
row 1	s	*			
row 0	s^0	*			

where the coefficients for the first two rows, row n and row $n-1$, are filled by the user based on the polynomial shown in (4.43). That is, the coefficients on the first row are even number in the increment order and the coefficients on the second row are odd number also in the increment order. The following coefficients are calculated based on the coefficients of the two previous rows using determinants, which is

$$b_1 = -\frac{\begin{vmatrix} 1 & a_2 \\ a_1 & a_3 \end{vmatrix}}{a_1} = \frac{a_1 a_2 - a_3}{a_1}; \quad b_2 = -\frac{\begin{vmatrix} 1 & a_4 \\ a_1 & a_5 \end{vmatrix}}{a_1} = \frac{a_1 a_4 - a_5}{a_1}; \quad b_3 = -\frac{\begin{vmatrix} 1 & a_6 \\ a_1 & a_7 \end{vmatrix}}{a_1} = \frac{a_1 a_6 - a_7}{a_1}$$

$$c_1 = -\frac{\begin{vmatrix} a_1 & a_3 \\ b_1 & b_2 \end{vmatrix}}{b_1} = \frac{a_3 b_1 - a_1 b_2}{b_1}; \quad c_2 = -\frac{\begin{vmatrix} a_1 & a_5 \\ b_1 & b_3 \end{vmatrix}}{b_1} = \frac{a_5 b_1 - a_1 b_3}{b_1}; \quad c_3 = -\frac{\begin{vmatrix} a_1 & a_7 \\ b_1 & b_4 \end{vmatrix}}{b_1} = \frac{a_7 b_1 - a_1 b_4}{b_1}$$

$$d_1 = -\frac{\begin{vmatrix} b_1 & b_2 \\ c_1 & c_2 \end{vmatrix}}{c_1} = \frac{b_3 c_1 - b_1 c_2}{c_1}; \quad d_2 = -\frac{\begin{vmatrix} b_1 & b_3 \\ c_1 & c_3 \end{vmatrix}}{c_1} = \frac{b_3 c_1 - b_1 c_3}{c_1}; \quad d_3 = -\frac{\begin{vmatrix} b_1 & b_4 \\ c_1 & c_4 \end{vmatrix}}{c_1} = \frac{b_4 c_1 - b_1 c_4}{c_1};$$

Similarly, if more coefficients are included for a higher order polynomial, the additional coefficients can be calculated in a similar way such as e_1, e_2, e_3, and so on.

There are two possible situations on which the Routh's stability criterion cannot be directly applied: (1) the first coefficient in one of the rows is zero and (2) the entire row on the Routh table is zero.

For the first situation, we can replace the zero with a small positive constant $\varepsilon > 0$ and proceed as the normal situations. The stability criterion is applied by taking the limit as $\varepsilon \to 0$. For example, the characteristic equation or polynomial is

$$P(s) = s^5 + 3s^4 + 2s^3 + 6s^2 + 6s + 9$$

The related Routh array or table can be calculated as

s^5	1	2	5	
s^4	3	6	8	
s^3	0	3	0	
s^3	ε	3	0	$\Leftarrow$ New s^3, with ε replacing 0
s^2	$\frac{2\varepsilon-3}{\varepsilon}$	3	0	
s	$3-\frac{3\varepsilon^2}{2\varepsilon-3}$	0	0	
s^0	3	0		

For small $\varepsilon < 1$, there are two sign changes in the first column, from row s^2 to row s ($-$ to $+$) and from row s to s^0 ($+$ to $-$), respectively. This means that there are two roots in the RHP and the system is not stable.

For the second situation, the rest of the Routh array can still be obtained by the following modified steps:

(1) Use the previous $(i + 1)$ row as an auxiliary polynomial $P(s)$.
(2) Take the derivative of $P(s)$, and use the resulting coefficients as the elements of the ith row.
(3) Continue as normal to obtain the rest elements of the array.

For example, for the following polynomial:

$$P(s) = s^5 + 2s^4 + 24s^3 + 48s^2 - 25s - 50$$

s^5	1	24 (a_2)	-25 (a_4)
s^4	2 (a_1)	48 (a_3)	-50 (a_5)
$s^3(1)$	0	0	
$s^3(2)$	4 (b_1)	48 (b_2)	$\Leftarrow$ Derivative the row s^4 to fill coefficients to here
s^2	24	-50	$\Leftarrow$ Using determinants a_1, a_3, b_1, b_2 to calculate this row
s	14.08	0	
s^0	-50		

As all elements in row $s^3(1)$ are zero, an auxiliary polynomial $s^3(2)$ is formed using the elements of the previous row and its derivative is obtained as

$$P(s) = s^4 + 24s^2 - 25 \text{ and } \frac{\mathrm{d}P(s)}{\mathrm{d}s} = 4s^3 + 48s$$

The coefficients of $P'(s)$, 4 and 48, are then used as the elements of the row $s^3(2)$ and the process continues as normal. The coefficients 24 and −50 in the next line are calculated as

$$-\frac{\begin{vmatrix} a_1 & a_3 \\ b_1 & b_2 \end{vmatrix}}{b_1} = \frac{a_3 b_1 - a_1 b_2}{b_1} = \frac{48 \times 4 - 2 \times 48}{4} = 24;$$

$$-\frac{\begin{vmatrix} a_1 & a_5 \\ b_1 & b_3 \end{vmatrix}}{b_1} = \frac{a_5 b_1 - a_1 b_3}{b_1} = \frac{-50 \times 4 - 1 \times 0}{4} = -50$$

As there is only one sign change (from 14.08 to −50) in the first column, the system is unstable and there is one root on the RHP.

Next, let us have a closer look at the linear system approximations and their applications for most physical systems.

4.4.4 Linear Approximation of Physical Systems

As we mentioned, all physical systems in our real world belong to nonlinear systems. Therefore, in order to use modern computational methods to analyze and design control systems, we must find some ways to perform linear approximations to obtain linear systems.

Regularly, some conditions should be met to approximate nonlinear systems to linear systems. First of all, the system should work in a certain range to enable us to make it a linear system as closely as possible. Second, the input signal should be limited to a small range to allow us to select the desired working point around it.

Figure 4.9 shows a mass sitting on a nonlinear spring. The normal working point is the equilibrium position that occurs when the spring force balances the gravitational force Mg. So at that position $f_0 = Mg$. The relationship between the spring force f and the distance of spring moving y is nonlinear, $f = ky^2$, with the equilibrium point being $y_0 = \sqrt{Mg}$.

In order to make this nonlinear system to a linear one, we have to limit the working range of this spring to a small range around the equilibrium position, as shown in Fig. 4.9. Exactly only the straight line with the slope at the working point in the range Δy and Δf can be considered as a linear equation. The system will become a nonlinear one when the force and spring moving distance are beyond that range.

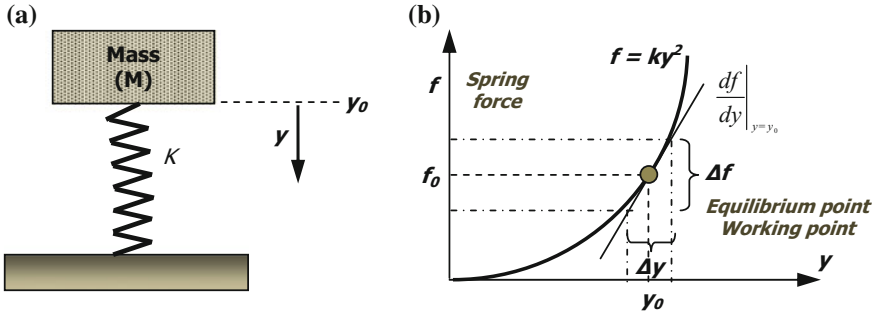


Fig. 4.9 A mass sitting on a nonlinear spring

Another example is a MOSFET amplifier circuit shown in Fig. 4.10.
The relationship between the output current i_D and the input voltage v_{GS} is

$$i_D = K_n(V_{GS} - V_{TN})^2 \tag{4.44}$$

This is a nonlinear system and needs to be approximated to a linear one. Two conditions must be met to make this as a linear system

(1) The MOSFET must be working in the saturation region and Eq. (4.44) will be true.
(2) The input voltage V_{GS} must be limited to a small range ΔV_{GS} around the DC bias point.

If both conditions are true, this nonlinear system can be approximated to a linear system. In fact, the straight line with a slope shown in Fig. 4.10b at the working point (Q-Point) in the range Δi_D and ΔV_{GS} is a linear equation.

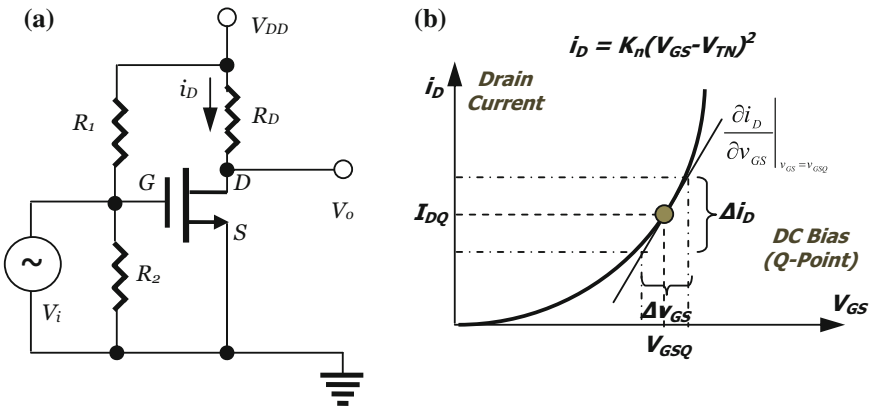


Fig. 4.10 A NMOSFET amplifier circuit

In order to meet the superposition principle to allow us to perform appropriate analysis for most real physical systems, we need to approximate those nonlinear systems to linear systems. A popular and powerful way to do this is to use **Taylor series** expansion around the operating point (equilibrium point for mass–spring system shown in Fig. 4.9b and *Q*-Point for MOSFET amplifier circuit shown in Fig. 4.10b).

For example, we have a nonlinear system represented by a nonlinear function

$$y(t) = f[x(t)]$$

where $f[x(t)]$ indicates that $y(t)$ is a nonlinear function of $x(t)$. The normal operating point is designated by x_0. By using the Taylor series expansion method around the operating point x_0, we get

$$y(t) = f[x(t)] = f(x_0) + \left.\frac{df}{dx}\right|_{x=x0} \frac{(x - x_0)}{1!} + \left.\frac{d^2f}{dx^2}\right|_{x=x0} \frac{(x - x_0)^2}{2!} + \cdots$$

The slope at the operating point x_0

$$\left.\frac{df}{dx}\right|_{x=x0}$$

is a good approximation to the curve over a small range of $(x - x_0)$, the deviation from the operating point. Thus, as a reasonable approximation, a nonlinear function around the operating point can be approximated to a linear function using the Taylor series expansion method. The approximation result can be expressed as

$$y(t) = f(x_0) + \left.\frac{df}{dx}\right|_{x=x0} (x - x_0) = y_0 + m(x - x_0)$$

or

$$\Delta y = m\Delta x$$

For example, for our MOSFET amplifier circuit, the output drain current and the input voltage has a nonlinear relationship as

$$i_D = K_n(V_{GS} - V_{TN})^2$$

The *Q*-point can be I_{DQ} and V_{GSQ}. The linear model around the *Q*-point for small deviation is

$$\Delta i_D = m\Delta v_{GS}$$

$$m = \left.\frac{\mathrm{d}i_D}{\mathrm{d}v_{GS}}\right|_{vGS=VGSQ} = 2K_n(V_{GSQ} - V_{TN})$$

For a multiple variable system, $y(t)$ is a function of multivariable,

$$y(t) = f(x_1, x_2, x_3, \ldots, x_n)$$

The slopes or deviations for multiple variables can be approximated using the partial deviations for each variable in Taylor series expansion method.

4.4.5 Some Important MATLAB Functions Used in System Models

MATLAB provides a set of functions used to assist users to analyze and design control systems, and most functions are located in the related toolboxes, such as Control Toolbox and System Identification Toolbox. In this section, we only concentrate on some popular functions used for system models and parameter-related analysis.

The functions covered in this section are

1. Transfer function-related functions
   - `tf()`
   - `step()`
   - `tfdata()`
   - `idtf()`
   - `zpk()`
   - `residue()`
2. Block diagram-related functions
   - `series()`
   - `parallel()`
   - `feedback()`
3. Root locus-related functions
   - `roots()`
   - `pole()`
   - `zero()`
   - `pzplot()`
   - `rlocus()`
4. Frequency response and state-space-related functions
   - `frd()`
   - `ss()`

4.4.5.1 Transfer Function-Related Functions

The `tf()` is a very useful function in control system analysis and design, and it can be used to build or convert a system to its transfer function format. The syntax of this function is

(1) `sys = tf(num, den)`,
(2) `sys = tf(num, den, Ts)`,
(3) `s = tf('s')`.

The first function **sys = tf(num, den)** creates a continuous-time transfer function `sys` with the numerator **num** and denominator **den**. For example, `sys = tf([1 2],[1 0 10])` specifies the transfer function $(s + 2)/(s^2 + 10)$. The second function **sys = tf(num, den, T_s)** creates a discrete-time transfer function with a sample time **T_S** (**T_S** = −1 if the sample time is undetermined). For example, `sys = tf(1,[1 2 5],0.1)` creates a discrete transfer function with a sample period as 0.1 s. The third function **s = tf('s')** specifies the transfer function `H(s) = s` (Laplace variable). You can then specify transfer functions directly as expressions in `S` or `Z` domain later, for example,

```
s = tf('s');
H = exp(-s)*(s + 1)/(s^2 + 3*s + 1);
```

The function `sys = tf(sys)` converts any dynamic system `sys` to the transfer function format. The resulting sys is always of class `tf`.

The `step()` function can be used to plot a step response for a control system, either open- or closed-loop system. The syntax of this function is

```
(1) [Y,T] = step(sys)
(2) step(sys)
```

The first function `[Y,T] = step(sys)` is used to compute the step response `Y` of the dynamic system `sys`. The time vector `T` is expressed in the time units of `sys` and the time step and final time are chosen automatically. For multi-input systems, independent step commands are applied to each input channel. If `sys` has `NY` outputs and `NU` inputs, `Y` is an array of size `[LENGTH(T) NY NU]`, where `Y(:,:,j)` contains the step response of the *j*-th input channel.

The second function `step(sys)` can be used to directly plot the step response for a control system `sys`.

For example, using the following MATLAB scripts, one can plot the step response of the transfer function `H = exp(-s)*(s + 1)/(s^2 + 3*s + 1)`.

```
s = tf('s');
H = exp(-s)*(s+1)/(s^2+3*s+1);
sys=tf(H);
step(sys);
grid;
```

The function `tfdata()` can be used to quickly access the transfer function data. Three syntaxes are popular which are given below:

```
(1) [NUM,DEN] = tfdata(sys)
(2) [NUM,DEN] = tfdata(sys, 'v')
(3) [NUM,DEN,TS] = tfdata(sys)
```

The first function `[NUM,DEN] = tfdata(SYS)` returns the dimensions of the numerator(s) and denominator(s) of the transfer function `sys`. For a transfer function with `NY` outputs and `NU` inputs, `NUM` and `DEN` are `NY-by-NU` cell arrays where the `(I,J)` entry specifies the transfer function from input `J` to output `I`. The sys is first converted to transfer function if necessary.

The second function `[NUM,DEN] = tfdata(sys, 'v')` returns the real values of the numerator(s) and the denominator(s) in a vector format. The third function also returns the sampling period T_s.

For example, when using the second function to get the real values of the numerator and denominator for the system *H* shown above, the following scripts can be developed to get the actual values for numerator and denominator (the results are shown on right):

```
s = tf('s');
H = exp(-s)*(s+1)/(s^2+3*s+1);
sys=tf(H);
step(sys);
grid;
[NUM,DEN] = tfdata(sys, 'v')
```

```
NUM = 0    1    1
DEN = 1    3    1
```

The function `sys = idtf(NUM,DEN)` has the similar functionality as `tf()` and it can be used to create a continuous-time transfer function `sys` with numerator `NUM` and denominator `DEN`.

A transfer function of an LTI system with no pure time delay is a ratio of polynomials in `s`. It can be written in terms of the numerator's and denominator's coefficients using the `tf()` function, or in terms of the numerator's factors (zeros) and denominator's factors (poles) using the `zpk()` function.

Both `tf()` and `zpk()` functions may be used for both a block construction and conversion from one representation to another. The only difference between these two functions is that the former is used to create a transfer function if the numerator and denominator are known, the second function `zpk()` is used to build a transfer function if the poles and zeros are known.

Three popular syntaxes of the function `zpk()` are

(1) `sys = zpk(Z,P,K)`

(2) `sys = zpk(Z,P,K,Ts)`

(3) `sys = zpk(M)`

The function `sys = zpk(Z, P, K)` creates a continuous-time zero-pole-gain model with zeros `Z`, poles `P`, and gain(s) `K`. The output `sys` is a `zpk` model object

storing the model data. In the SISO case, `Z` and `P` are the vectors of real- or complex-valued zeros and poles, and `K` is the real- or complex-valued scalar gain, such as

$$G(s) = K\frac{(s-z(1))(s-z(2))\ldots(s-z(m))}{(s-p(1))(s-p(2))\ldots(s-p(n))}$$

If this function returns a `[]`, it means that no zeros or poles are available to the system.

The function `sys = zpk(Z, P, K, Ts)` creates a discrete-time zero-pole-gain model with sample time Ts (in seconds). Set Ts = −1 or Ts = [] to leave the sample time unspecified. The input arguments `Z,P,K` are as in the continuous-time case.

The function `sys = zpk(M)` specifies a static gain M.

For example, we have a transfer function $H(s)$

$$H(s) = \frac{2s}{(s-1+j)(s-1-j)(s-2)}$$

One can use the function $H = zpk(0, [1-i\ \ 1+i\ \ 2],\ 2)$ to build it a `zpk` model object.

Using the following MATLAB scripts, one can convert the transfer function to a zpk model

```
h = tf([-10 20 0],[1 7 20 28 19 5]);
zpk(h)
```

The function returns the following result:

```
ans =
    -10 s (s-2)
  -------------------
  (s + 1)^3 (s^2 + 4 s + 5)
```

The function `residue()` can be used to help users to quickly perform partial fraction expansion operations to get coefficients for each item for a Laplace transform. The format of this function is `[R, P, K] = residue(B,A)`, which is used to find the residues, poles, and direct term of a partial fraction expansion of the ratio of two polynomials `B(s)/A(s)`.

The vectors `B` and `A` specify the coefficients of the numerator and denominator polynomials in descending powers of `s`. The residues are returned in the column vector `R`, the pole locations in column vector `P`, and the direct terms in row vector `K`. The number of the poles is equal to `n = length(A) − 1 = length (R) = length(P)`. The direct term coefficient vector `K` is empty if `length`

`(B) < length(A);` otherwise, `length(K) = length(B) − length (A) + 1`.

For example, for a partial fraction expansion equation, if there are no multiple roots, it is

$$\frac{B(s)}{A(s)} = \frac{R(1)}{s - P(1)} + \frac{R(2)}{s - P(2)} + \cdots + \frac{R(n)}{s - P(n)} + K(s)$$

If `P(j) = ⋯ = P(j + m − 1)` is a pole of multiplicity m, then the expansion includes terms of the form

$$\frac{R(j)}{s - P(j)} + \frac{R(j+1)}{(s - P(j))^2} + \cdots + \frac{R(j+m-1)}{(s - P(j))^m}$$

For example, we have a transfer function described by

$$\frac{C(s)}{R(s)} = \frac{10(s+2)}{s^2 + 8s + 15}$$

In order to get $c(t)$ in the time domain, we need to do partial fraction expansion operations to get coefficients for each item for a Laplace transform. Then, using the inverse Laplace transform we can get the output $c(t)$. Using the following MATLAB scripts, we can easily get this job done

```
B = [10 20];
A = [1 8 15];
[R, P, K]=residue(B, A)
```

```
R = 15
    -5
P = -5
    -3
K = []
```

The running results for this piece of codes are shown on the right, which means that

$$\frac{10(s+2)}{s^2 + 8s + 15} = \frac{R(1)}{s - P(1)} + \frac{R(2)}{s - P(2)} + K = \frac{15}{s+5} - \frac{5}{s+3} + 0$$

It looks like so easy to decompose or to do partial fraction expansion of this equation.

4.4.5.2 Block Diagram-Related Functions

Both `series()` and `parallel()` are popular functions used to connect multiple LTI systems in series or parallel format. In most situations, some complicated systems can be significantly simplified using these functions to make our analysis much easier.

For example, we have a control system with C(s) and G(s) with the following transfer functions:

$$G(s) = \frac{1}{200s^2} \text{ and } C(s) = \frac{s+1}{s+2}$$

If we want to cascade or connect these two transfer functions in the series format, we can use the `series()` function as

```
M1 = tf([1], [200 0 0]);
M2 = tf([1 1], [1 2]);
sys = series(M1, M2)
```

```
sys =
          s + 1
  -----------------
  200 s^3 + 400 s^2
```

The `sys` returns a series connection of these two transfer functions that are shown above. It can be found that all zeros and poles in the final cascaded system are identical with the two original systems. This means that there is no changing for zeros and poles if multiple systems are cascaded or series connected. The zeros and poles in the resulted system are a combination or a collection of zeros and poles of original systems. This operation is equivalent to a multiplying operation, such as `sys` = $M_1 \times M_2$.

Now let us have a look at the `parallel()` function. Develop the following MATLAB scripts and run this piece of codes to see what happened:

```
M1 = tf([1], [200 0 0]);
M2 = tf([1 1], [1 2]);
sys = parallel(M1, M2)
```

```
sys =

  200 s^3 + 200 s^2 + s + 2
  -------------------------
      200 s^3 + 400 s^2
```

The `sys` returns a parallel connection of these two transfer functions that are shown above. It can be found that the poles are with no changes for the resulted parallel system, but the zeros are totally changed. In fact, the `parallel()` function performs an addition (+) operation to add two systems together, such as `sys` = M_1 + M_2. This is the reason why we get this kind of zeros' combination.

The `feedback()` function is used to help users to build a closed-loop system with a feedback path, which can be either negative or positive feedback.

The syntax of this function is

```
sys = feedback(sys1,sys2)
```

By default, the feedback is a negative feedback. However, if a positive feedback is desired, one can use

```
sys = feedback(sys1,sys2, +1)
```

to indicate it.

For example, we have a closed-loop system shown in Fig. 4.11. Now, we want to develop a closed-loop control output transfer function $Y(s)/R(s)$, use the following MATLAB scripts:

```
G = tf([1 0.5], [1 2 3]);
H = zpk(-2, -10, 5);
Y = feedback(G, H)
```

The function returns the following closed-loop transfer function:

```
Y =
        (s + 10) (s + 0.5)
  --------------------------------
  (s + 14.75) (s^2 + 2.245 s + 2.372)
```

This is much easier than doing that by hands.

4.4.5.3 Root Locus-Related Functions

The function `roots(p)` computes the roots of the polynomial whose coefficients are the elements of the vector p. If `p` has $n + 1$ components, the polynomial is $p(1)x^n + \cdots + p(n)x + p(n + 1)$. If p (s) is a row vector containing the coefficients of p(s) in descending order, the `roots(p)` returns a column vector containing the roots of the polynomial.

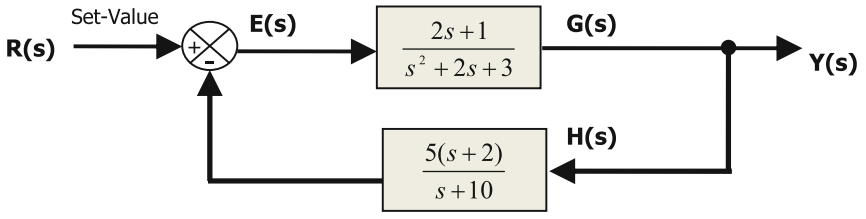


Fig. 4.11 A closed-loop system

For example, if we have a vector `p = [3 2 −2]`, which represents the polynomial $3x^2 + 2x - 2$. Using the MATLAB scripts

```
p=[3 2 -2];
roots(p)
```

```
ans =
   -1.2153
    0.5486
```

We can get two roots for this vector, which are shown on the above right.

The function `pole()` and `zero()` are used to compute poles and zeros for an LTI system. For example, if we have an LTI system with transfer function defined as

$$G(s) = \frac{s + 0.5}{s^2 + 2s + 3}$$

Using the following MATLAB scripts, we can get the poles and zeros for this system:

```
G = tf([1 0.5], [1 2 3]);
p = pole(G)
z = zero(G)
```

```
p =   -1.0000 + 1.4142i
      -1.0000 - 1.4142i
z =   -0.5000
```

The function `pzplot(sys)` computes the poles and zeros of a dynamic system model `sys` and plots them in the complex plane. The poles are plotted as ***x*** and the zeros are plotted as ***o***.

For example, when using this function to plot our LTI system $G(s)$ shown above, with the following scripts, the poles and zeros of this system can be plotted in the s-plane, which is shown in Fig. 4.12.

```
G = tf([1 0.5], [1 2 3]);
pzplot(G);
sgrid;
```

The function `rlocus(sys)` computes and plots the root locus of the single-input, single-output LTI model `sys`. The root locus plot is used to analyze the negative feedback loop and shows the trajectories of the closed-loop poles when the feedback gain `K` varies from 0 to Inf. This function can automatically generate a set of positive gain values that produce a smooth plot.

Alternatively, one can use the function `rlocus(sys, K)` with a user-specified vector `K` of gain values to compute and plot the root locus.

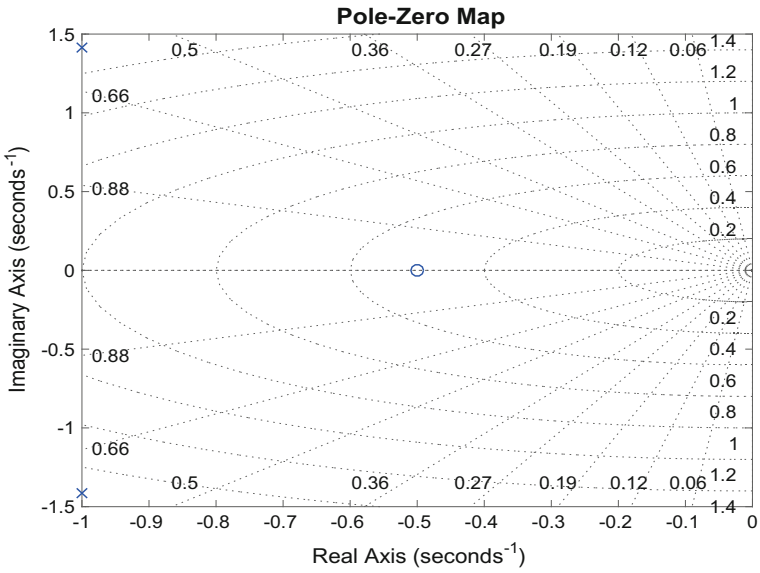


Fig. 4.12 The poles and zeros plotted by pzplot() function

For example, if we want to plot the root locus for our system $G(s)$ defined above, we can use the following scripts:

```
G = tf([1 0.5], [1 2 3]);
rlocus(G);
grid;
```

The resulted plot is shown in Fig. 4.13.

4.4.5.4 Frequency Response and State-Space-Related Functions

The function `sys = frd(response, frequency)` creates a frequency response data (FRD) model object `sys` from the frequency response data stored in the multidimensional array `response`. The vector `frequency` represents the underlying frequencies for the frequency response data. The function `sys = frd (response, frequency, Ts)` creates a discrete-time FRD model object `sys` with scalar sample time `Ts`. If we set `Ts` = −1, it creates a discrete-time FRD model object without specifying the sample time.

The function `sys = frd()` creates an empty FRD model object.

For example, if we want to use the function `sys = frd(response, frequency)` to create a FRD model, the following scripts can be used:

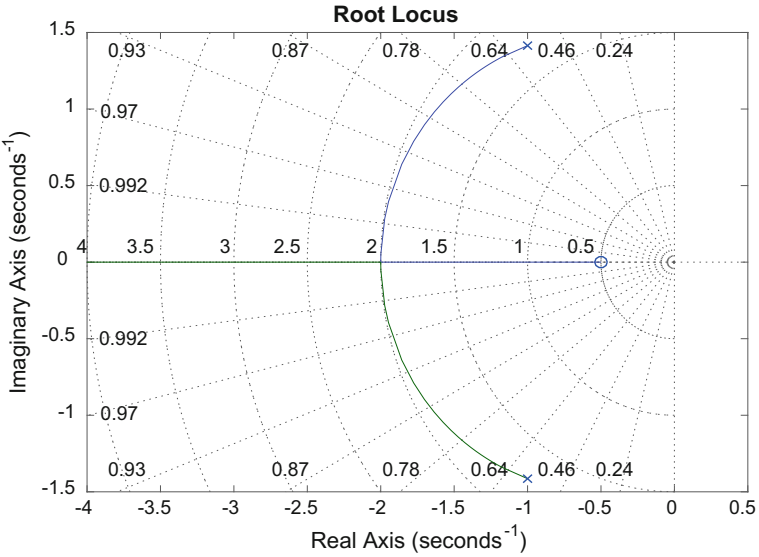


Fig. 4.13 The plot result of our system $G(s)$ with rlocus() function

```
freq = logspace(1,2);
resp = 0.05*(freq).*exp(i*2*freq);
sys = frd(resp,freq);
bode(sys);
grid;
```

The resulted FRD model is shown in Fig. 4.14.

The function `sys = ss()` can be used to create or convert a state-space model. Multiple syntaxes are available for this function such as

(1) `sys = ss(A,B,C,D)`,
(2) `sys = ss(A,B,C,D,Ts)`,
(3) `sys = ss(D)`,
(4) `sys = ss(A,B,C,D,ltisys)`, and
(5) `sys = ss(sys)`.

The function `sys = ss(A, B, C, D)` creates a state-space model object representing the continuous-time state-space model

$$\dot{x} = Ax + Bu$$
$$y = Cx + Du$$

For a model with `Nx` states, `Ny` outputs, and `Nu` inputs

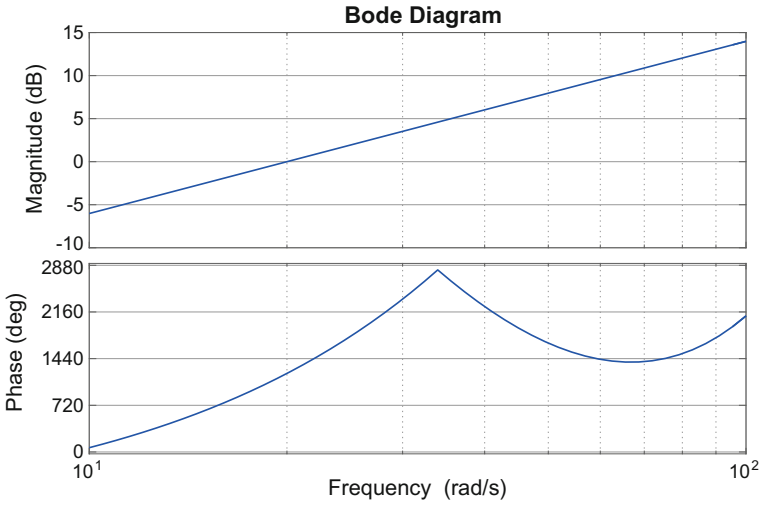


Fig. 4.14 Frequency response of a FRD model

- `A` is a `Nx-by-Nx` real- or complex-valued matrix.
- `B` is a `Nx-by-Nu` real- or complex-valued matrix.
- `C` is a `Ny-by-Nx` real- or complex-valued matrix.
- `D` is a `Ny-by-Nu` real- or complex-valued matrix.

If we set `D` = 0, set `D` to the scalar 0 (zero), regardless of the dimension. The function `sys = ss(A,B,C,D,Ts)` creates the discrete-time model

$$x[n+1] = Ax[n] + Bu[n]$$
$$y[n] = Cx[n] + Du[n]$$

with sample time `Ts` (in seconds). If we set `Ts` = −1 or `Ts` = [], it will leave the sample time unspecified. The function `sys = ss(D)` specifies a static gain matrix `D` and is equivalent to

```
sys = ss([], [], [], D)
```

The function `sys = ss(A,B,C,D,ltisys)` creates a state-space model with properties inherited from the model `ltisys` with the original sample time.

The function `sys = ss(sys)` converts a dynamic system model sys to state-space form. The output `sys` is an equivalent state-space model (`ss` model object). This operation is known as **state-space realization**.

For example, we have a system represented by the state-space model as

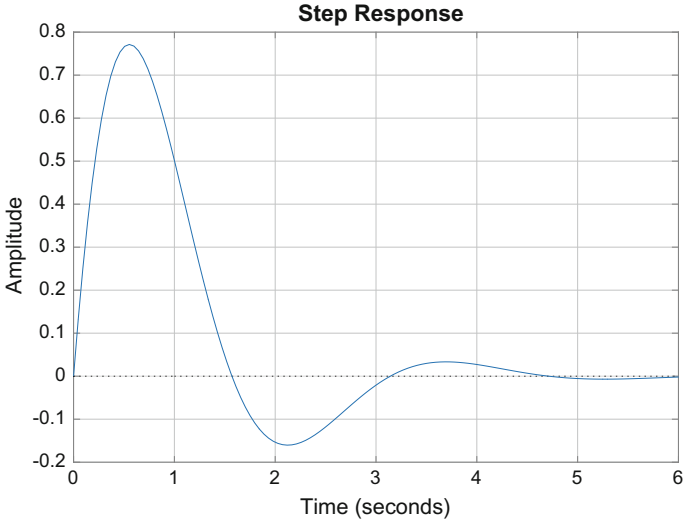


Fig. 4.15 Step response of the state-space model

$$A = \begin{bmatrix} 0 & 1 \\ 5 & -2 \end{bmatrix}, \quad B = \begin{bmatrix} 0 \\ 3 \end{bmatrix}, \quad C = [0 \quad 1], \quad D = [0]$$

Using the following scripts, we can build a state-space model:

```
A = [0 1;-5 -2];
B = [0;3];
C = [0 1];
D = 0;
sys = ss(A,B,C,D);
step(sys);
grid;
```

The step response of this system is shown in Fig. 4.15.

4.5 System Model Identification

We have provided very detailed discussions about the system models in the previous sections. Now, we need to take care of an important issue, which is how to obtain an actual model for a target system? This topic belongs to the system identification process. There are different ways to deal with system models.

First, for some popular physical systems that can be described by mathematical or dynamic equations, such as DC motors, simple spring–mass-block, and electrical circuits, we can build models starting from those equations directly. However, for some complicated control plants, such as airplane force controls and complicated chemical processes. For the second case, we need to use some devices to excite the plant and collect measured data to derive the system model. Anyway, the process of constructing models from experimental data is called system identification.

The system identification process can be categorized into two groups; **parametric model** and **nonparametric model**. For those systems whose transfer functions or state-variable descriptions are given or known, we can obtain the poles and zeros of system directly, and what we only need to do is to design our controller based on these poles and zeros, which we called parameters. However, in some situations, the systems are given by their frequency responses, not parameters, such as Bode, Nyquist, or Nichols curves of amplitude and phase of the transfer functions. Of course, we can compute the transfer function based on those frequency responses. But if we can directly design our controllers in terms of those frequency responses, we do not need to get those poles and zeros via transfer functions anymore. In this way, we can do our design without knowing those parameters. Therefore, we called those function curves represented in the frequency domain as nonparametric model.

4.5.1 Identifying Models from Experimental Data

The data to be used for the system identification process are composed of two types: the inputs $r(t)$ and the output $y(t)$ with some error assumptions. In fact, the system identification process is an iteration process by repeatedly checking the errors between the nominal model and the desired actual model with the collected input and output data, until the errors are within the acceptable range.

The experimental data can be categorized into four types: (1) the transient response, such as a step response, (2) the sinusoidal steady-state signals with various frequencies, (3) the pseudorandom noises, and (4) the stochastic steady-state signals.

The transient response data is easy and quick to obtain. However, in order to get high signal-to-noise ratio, the transient response must be highly noticeable. Thus, this method is not very popular for most normal operations, and the data must be collected as part of special tests. Anyway, this method may be simple for some special cases or complex for the general cases.

Frequency response data is simple to obtain but it may be more time consuming than the transient response method. The good point is that the frequency response data can be directly used for controller design without any other parameters of the plant.

The pseudorandom noise can be generated by any computer easily. A spectrum analysis is needed to construct the system model based on those pseudorandom noise signals.

If a high signal-to-noise ratio cannot be easily obtained, alternatively we can use the averaging or statistical estimation method. The shortcoming of using this method is that a lot of equations need to be derived based on stochastic processes.

In summary, the identification is a process by which a model is built from prior knowledge plus experimental data and includes four basic components: the data, the model set, the criterion by which one proposed model is to be compared with another, and the process of validation by which the model is confirmed to mimic. Because of the large data to be collected to obtain desired model and the complexity of many algorithms to be used, in this chapter, we prefer to use computer aids to help us to perform this identification process. The MATLAB® System Identification Toolbox™ is an ideal tool to perform this job.

4.6 Introduction to MATLAB® System Identification ToolBox™

This toolbox mainly provides identification techniques such as maximum likelihood, prediction error minimization (PEM), and subspace system identification for LTI systems. To represent nonlinear system dynamics, one can estimate Hammerstein–Wiener models and nonlinear ARX models with wavelet network, tree-partition, and sigmoid network nonlinearities. The System Identification Toolbox performs gray-box system identification for estimating parameters of a user-defined model. One can use the identified model for system response prediction and plant modeling in MATLAB® Simulink®. This toolbox also supports time-series data modeling and time-series forecasting.

In fact, the MATLAB System Identification Toolbox provides a sequence or group of methods to help users to perform system identification processes, which includes both linear and nonlinear system identification methods.

For linear system identifications, the following methods are provided:

- Process models,
- Input–output polynomial models,
- State-Space models,
- Transfer Function models,
- Linear gray-box models, and
- Frequency response models.

For nonlinear system identifications, the following methods are available:

- Nonlinear ARX models,
- Hammerstein–Wiener models, and
- Nonlinear gray-box models.

In addition to these identification models, this toolbox also provides the following functions:

- Model validation,
- Model analysis,
- Time-series analysis,
- Online estimations, and
- Diagnostics and prognostics.

MATLAB also provides a graphical user interface (GUI) identification tool called system identification tool, which is a GUI for estimating and analyzing linear and nonlinear models in System Identification Toolbox™. Using system identification tool, you perform the primary tasks of the identification process. You execute commands by making selections in the GUI menus, pop-up menus, and checkboxes.

Because of the space limitation, we can only discuss some popular methods in this section.

For linear system identifications, we may concentrate on the following methods:

(1) Process models and
(2) System identification tool (GUI)—which is also called system identification app.

For nonlinear system, we may discuss the nonlinear ARX model method.

4.6.1 Process Models

The ***process model*** is exactly a simple continuous transfer function that describes linear system dynamics in terms of one or more of the following elements:

- Static gain K_p.
- One or more time constants T_{pn}. For complex poles, the time constant is called T_ω, which is equal to the inverse of the natural frequency, and the damping ratio is ζ.
- Process zero T_z.
- Possible time delay T_d before the system output responds to the input (*dead time*).
- Possible enforced integration.

Process models are widely used to describe system dynamics in many industries and are applied to various production environments. The advantages of these models are that they are not only simple but also support transport delay estimation, and the model coefficients have an easy interpretation as poles and zeros.

One can create different model structures by varying the number of poles, adding an integrator, and adding or removing a time delay or a zero. One can also specify a first-, second-, or third-order model, and the poles can be real or complex.

For example, the following model structure is a first-order continuous process model, where K is the static gain, T_{p1} is a time constant, and T_d is the input-to-output delay:

$$G(s) = \frac{K}{1 + sT_{p1}} e^{-sT_d} \tag{4.45}$$

Such that $Y(s) = G(s)R(s) + E(s)$, where $Y(s)$, $R(s)$, and $E(s)$ represent the Laplace transforms of the output, input, and output error, respectively. The output error, $e(t)$, is white Gaussian noise with variance λ. One can account for colored noise at the output by adding a disturbance model, $H(s)$, such that $Y(s) = G(s)R(s) + H(s)E(s)$.

Three basic functions are available under this kind of identification models, they are

(1) procest(),
(2) idproc(), and
(3) pem().

Among them, the first function, **procest()** is more popular and let us have a closer look at it.

The basic syntax of this function is

```
sys = procest(data, type)
```

This function allows users to estimate process model **sys** using time or frequency **data**. The **type** defined the structure of the model sys. The input argument **data** must be an object (iddata, idfrd, or frd) and meet the following requirements:

- For time-domain estimation, the data must be specified as an **iddata** object containing both input and output signal values.
- For frequency-domain estimation, data can be one of the following:
  - Recorded frequency response data (**frd** or **idfrd**).
  - Iddata object with its properties is specified as follows:
    - InputData—Fourier transform of the input signal.
    - OutputData—Fourier transform of the output signal.
    - Domain—"Frequency".
- `data` must have at least one input and one output.

The input argument **type** must be a string and includes the following letters:

- P—all types of acronyms start with this letter.
- 0, 1, 2, or 3—number of time constants (poles) to be modeled. Possible integrations (poles in the origin) are not included in this number.
- I—integration is enforced (self-regulating process).
- D—time delay (dead time).
- Z—extra numerator term, a zero.

- U—underdamped modes (complex-valued poles) permitted. If U is not included in type, all poles must be real. The number of poles must be two or three.

An example of this type string can be "P1D", "P2DZ", and so on.

The output of this function sys is the identified model returned as an **idproc** model of a structure defined by type.

Following is an example of using this function to perform identification process:

```
data = idfrd(idtf([10 2],[1 1.3 1.2],'iod',0.45),logspace(-2,2,256));
type = 'P1D';
sys = procest(data,type);
sys
```

The identified model sys will be displayed in the idproc format.

4.6.2 System Identification Tool (GUI)

In addition to various functions in the System Identification Toolbox, MATLAB also provides a graphical user interface (GUI) identification tool called system identification tool (system identification app in MATLAB R2017b) to assist users to estimate a linear or a nonlinear system model. Basically, this GUI provides a user interface to connect users to all identification functions with a set of visible components such as icons, combo boxes, and blocks. Let us have a closer look at this tool.

Open the system identification tool by typing **ident** in the MATLAB® Command window after the *fx*≫ cursor and press the **Enter** key. The opened identification tool wizard is shown in Fig. 4.16. Two major parts are shown in this wizard: the **Data Views** on the left and the **Model Views** on the right of the wizard. The connection between these two parts is the **operations**.

Data Views

This part allows users to select the input data used to estimate the desired model, which includes

- **Import Data** combo box: Select the input data type (time or frequency domain, data object or example).
- Data Plot View: Preview of the input data in a plot format.
- Data View Style: Plot data in time or frequency domain, or spectrum format by checking the related checkbox.

Model Views

This part is used to import or input an existing or predefined model that can be compared with the identified model to confirm or check the identification result.

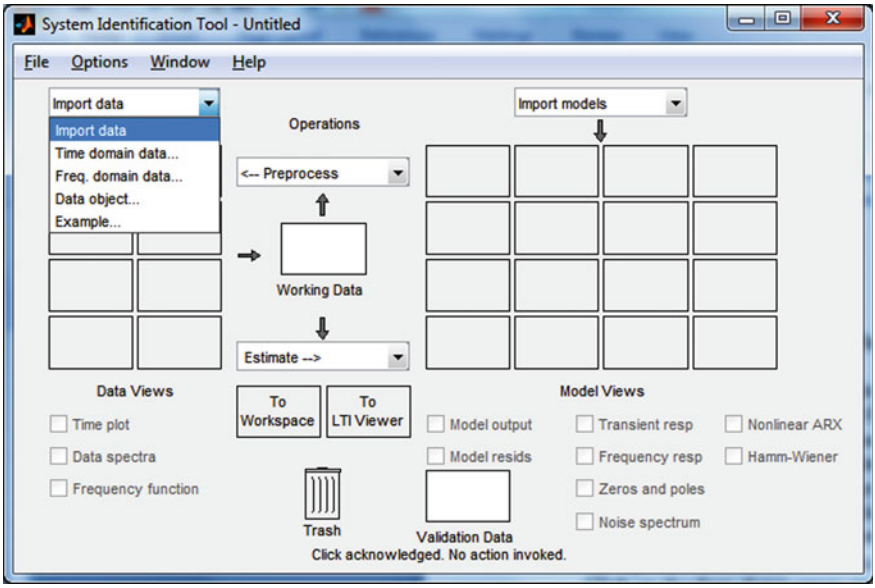


Fig. 4.16 The opened identification tool wizard. Reprinted with the permission of MathWorks, Inc.

The **Import Models** combo box allows users to select and import pre-built models into the Model Views blocks.

Operations

This part allows users to perform the following functions/selections to help the tool to do the desired identification process:

- The combo box **Preprocess**: Allow users to perform selection and filtering functions to select, filter, or remove some portions on the original data, as shown in Fig. 4.17.
- The combo box **Estimate**: Allow users to select the desired identification model method, as shown in Fig. 4.18.
- **To Workspace** cell: Allow users to transfer the identification result to the workspace and use it later. To do this transfer, users must drag the identification result from the Model Views block and place it into this cell.
- **To LTI Viewer** cell: Allow users to view the simulated results for the identified model, such as transient response, model output, model residuals, frequency response, zeros and poles as well as noise spectrum. To show these views, users must also drag the identification result from the Model Views block and place it into this cell. Also, the related checkboxes located on the lower-right corner, such as **Model output**, **Model resids**, **Transient resp**, **Frequency resp**, **Zeros and poles**, and **Noise spectrum**, should be checked in order to show those responses. **The Nonlinear ARX** and **Hamm–Wiener** checkboxes are used for nonlinear system identifications.

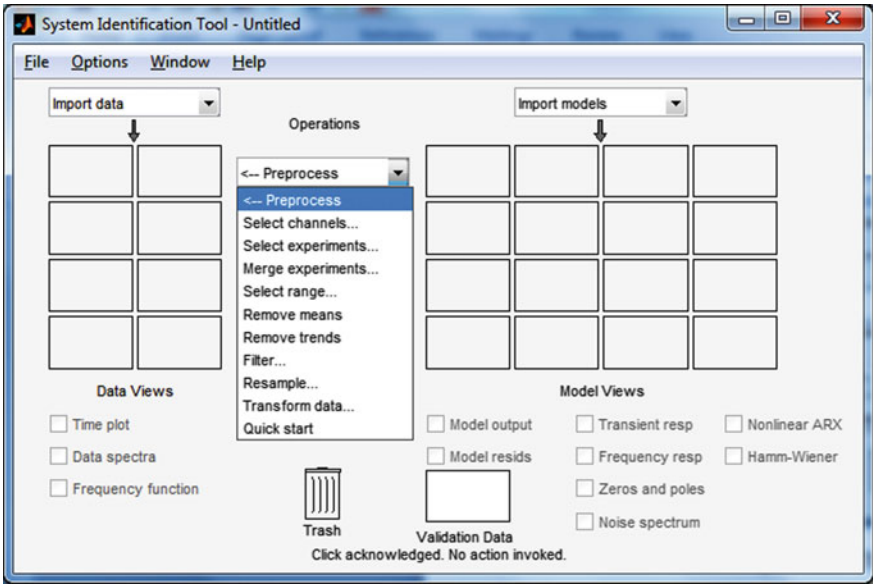


Fig. 4.17 The functions in the preprocess combo box. Reprinted with the permission of MathWorks, Inc.

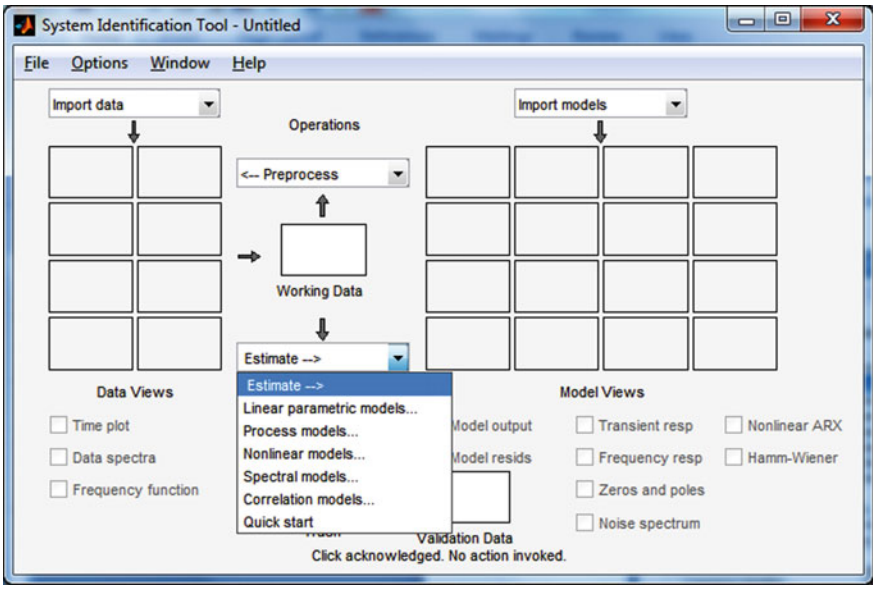


Fig. 4.18 The functions in the estimate combo box. Reprinted with the permission of MathWorks, Inc.

4.6.3 Nonlinear ARX Models

Use nonlinear ARX models to represent nonlinearities in a system using dynamic nonlinearity estimators such as wavelet networks, tree-partitioning, and sigmoid networks. In the toolbox, these models are represented as **idnlarx** objects. You can estimate nonlinear ARX models in the **System Identification Tool** or at the command line using the **nlarx** command.

Because of simplification, we will discuss the system identification tool method. Perform the following steps to estimate a nonlinear ARX model with this App:

1. Import data into the system identification app.
2. To estimate a nonlinear ARX model using the imported estimation data and chosen nonlinearity estimators, the following steps are performed:

   (1) In the **System Identification** app, select **Estimate > Nonlinear models** to open the nonlinear model dialog box.
   (2) In the **Configure** tab, verify that nonlinear ARX is selected in the **Model type** list.
   (3) (Optional) Edit the **Model name** by clicking . The name of the model should be unique to all nonlinear ARX models in the system identification app.
   (4) (Optional) If you want to refine the parameters of a previously estimated model or configure the model structure to match that of an existing model the following steps must be followed:

      a. Click **Initialize**. An initial model specification dialog box opens.
      b. In the initial model drop-down list, select a nonlinear ARX model. The model must be in the model board of the system identification app and the input/output dimensions of this initial model must match that of the estimation data, selected as **Working Data** in the app.
      c. Click **OK**. The model structure and the parameter values are updated to match that of the selected model. Clicking **Estimate** causes the estimation to use the parameters of the initial model as the starting point.

   (5) To obtain regularized estimates of model parameters, in the **Estimate** tab, click the **Estimation Options**. Specify the regularization constants in the **Regularization_Tradeoff_Constant** and **Regularization_Weighting** fields.
   (6) Click **Estimate** to add this model to the system identification app.
   (7) The **Estimate** tab displays the estimation progress and results.

Now that we have a clear picture about the MATLAB System Identification Toolbox and tool, next let us take care of the identification process for a DC motor that we will use in our real project through this book.

4.7 Case Study: Identification of a Real DC Motor Mitsumi 448 PPR Motor

The DC motor that we have used in this book is modeled **Mitsumi 448 PPR Motor** with an optical rotary encoder including 448 lines/pulses speed disk, as shown in Fig. 4.19. This motor can be purchased from some sites such as ebay for about $5.00.

4.7.1 The Mitsumi 448 PPR Motor and Rotary Encoder

The DC motor **Mitsumi 448 PPR** motor is a general voltage-driven motor with rotation speed as the output. This motor also provides two phases, PhA and PhB, optimal rotary encoder as feedback to a closed-loop control system. With the quadrature encoder interface (QEI) provided by the TM4C123GH6PM MCU, those two-phase feedback signals can be fed into the QEI and detected by the micro-controller. In order to effectively identify this DC motor dynamic model, let us take a closer look at the rotary encoder and its working principle.

4.7.1.1 Introduction to Quadrature Encoder

Typically, two types of encoders are very popularly used in most applications: linear and rotary. Linear encoders are used for items moving only in a single dimension or direction, and convert the linear position into an electronic signal. These are often used in conjunction with actuators. Rotary encoders are used for items moving around an axis, like a motor, and convert the rotary positions or angles into electronic signals. Since the rotary encoders are more popular and

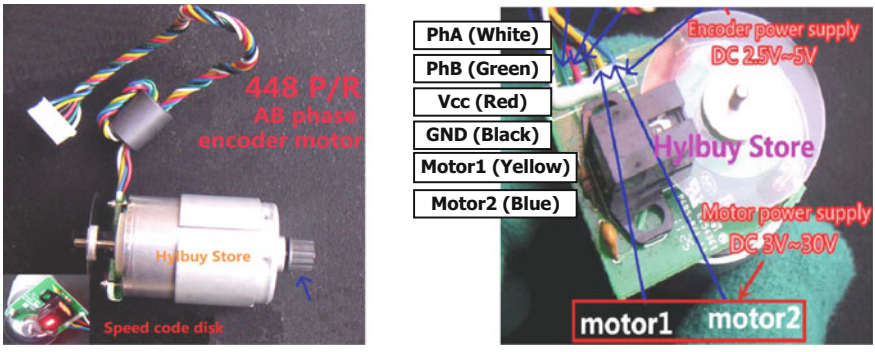


Fig. 4.19 The DC motor—Mitsumi 448 PPR motor with an optical encoder

widely implemented in the motor-related manufactures, such as automobiles and robots, we concentrate our discussions on this type of encoder in this section.

There are three common types of rotary encoderswhich are given below:

- Absolute position encoder,
- Incremental position encoder, and
- Incremental sinusoidal encoder.

One of the most popular rotary encoders is the optical encoder. An optical encoder has a disk with specific patterns mounted to the motor shaft. The patterns on the disk either block light or allow it to pass through. Therefore, a light-emitting transmitter is used along with a photocell receiver. The receiver signal output can be correlated to the motor's rotary position.

For absolute position value rotary encoders, the pattern on the disk is broken up into a very specific format based on its location. As an example, if the absolute position encoder has a 3-bit digital output, it would have eight different patterns, evenly spaced around the center of the disk, as shown in Fig. 4.20a. Since this is on a disk and is evenly spaced out, the spacing between each pattern is 360°/8 = 45°. Now the position of the rotary motor is known to be within 45° for a 3-bit absolute position value rotary encoder.

For the incremental rotary encoder, the pattern on the disk outputs a digital High or Low, which is a TTL signal. As shown in Fig. 4.20b, the TTL-output disk pattern is relatively simple compared to the absolute position value rotary encoder because it needs to represent only one digital High or digital Low. In addition to the TTL signal, there is also a reference mark or index which is essential in determining the motor's base or current rotary position. The reference mark or index can be thought of as a position 0 or angle 0°. Thus, simply counting the digital pulses can determine the exact rotary position of the motor.

Figure 4.20b illustrates multiple periods/lines in one revolution of the motor shaft. Different encoder vendors provide various incremental rotary encoders with 50 ~ 5,000 periods/lines per revolution. As with absolute position value rotary

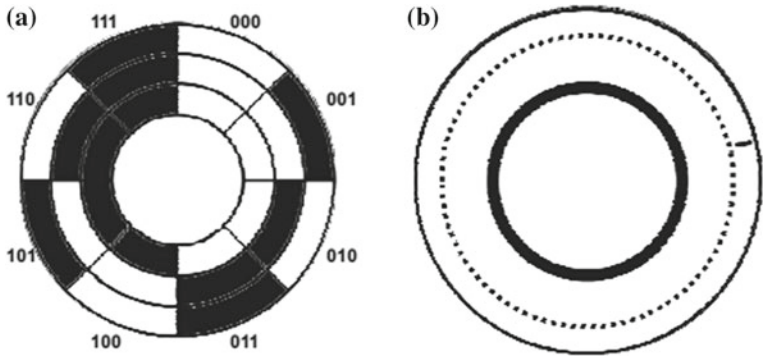


Fig. 4.20 Disk patterns used in the absolute and the incremental encoders

encoders, the output is already in a digital format, and therefore no analog signal converter is required.

For the incremental sinusoidal rotary encoder, the output and disk pattern is quite similar to the incremental rotary encoder. Instead of a digital output, the output is a sinusoidal wave output. Actually, it has both sine and cosine outputs along with the reference mark signal. These outputs are all analog, so an analog-to-digital converter is required.

4.7.1.2 The Working Principle of the Increment Rotary Encoder

As we mentioned in the last section, the incremental rotary encoders provide a specific number of equally spaced pulses per revolution (PPR) or per millimeter of linear motion.

The code disk inside an increment rotary or a quadrature encoder contains two 90° installed tracks usually denoted as Channel A (Phase A) and Channel B (Phase B). These tracks or channels are coded 90 electrical degrees out of phase, as indicated in Fig. 4.21. These channels are the key design elements that will provide the quadrature encoder its functionality.

In applications where direction sensing is required, a controller can determine direction of movement of the motor based on the phase relationship between Phases A and B. As illustrated in Fig. 4.21, when the detected Phase A is leading Phase B, the quadrature encoder is rotating in a clockwise direction, and the reverse will happen when the quadrature encoder rotates counterclockwise.

When a higher resolution is needed, it is possible for the counter to count the rising and the falling edges of the quadrature encoder's pulse train from one channel, which doubles (×**2**) the number of pulses per revolution (Fig. 4.22). Counting both rising and falling edges of both channels of a quadrature encoder will quadruple (×**4**) the number of pulses per revolution, as shown in Fig. 4.22. As a result, a 2,500 PPR quadrature encoder can create 10,000 pulses per turn with this method. Typically, this 4 × signal will be accurate to better than ±1 count.

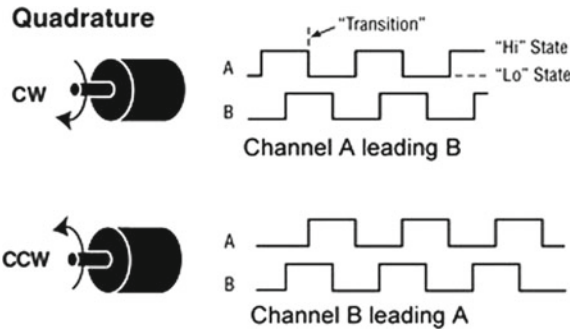


Fig. 4.21 Quadrature encoder rotation direction and phases A and B

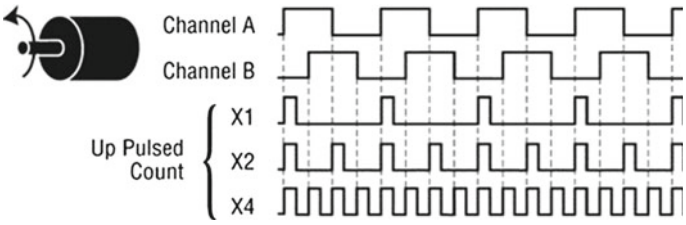


Fig. 4.22 Double and quadruple the number of pulse per revolution

4.7.1.3 The Increment Rotary Encoder Applied in the Closed-Loop Control System

Traditionally, the control systems can be divided into open-loop and closed-loop control systems. Figure 4.23 shows an open-loop and a closed-loop control system diagram.

For the open-loop control system shown on the top of Fig. 4.23, the desired set point **S**, either desired position or desired velocity, is directly applied to the controller and the controller output is implemented to the motor driver and motor itself. For this kind of control strategy, the actual motor position and velocity cannot be guaranteed to be the desired value set by the set point. This means that the target motor position or speed may be or may not be equal to the desired setup input value and the output of the system is out of control.

However, for the closed-loop control strategy shown on the bottom of Fig. 4.23, the output of the motor, either position or velocity, is feedback to the input via the encoder and is compared with the input setup value. The error between the input setup value and the output feedback value works as the net input signal to the controller. The advantage of using a closed-loop control system is

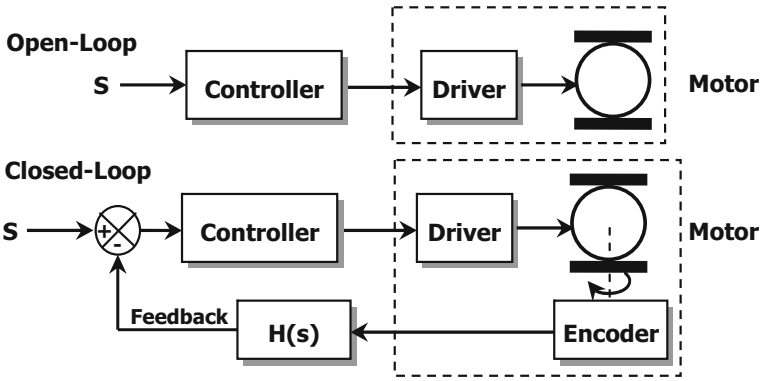


Fig. 4.23 The open-loop and the closed-loop control system

- The output of the motor, either position or the velocity, can be exactly equal to the input setup value theoretically since the output is feedback to the input.
- Compared with the open-loop control system, the closed-loop control system has the automatic tuning ability to make output equal to the input and make the error to 0.
- When the output is far away from the input, a greater error is obtained and applied to the input of the controller to obtain bigger control output to the motor. As the output is close to the input, a smaller error is obtained and applied to the controller to make the control output smaller.
- If the output is exactly equal to the input, the error is 0 and the control output is also 0 until some errors are coming.

Generally, the encoder provides a position feedback and a tachometer provides a velocity feedback of the motor for a given voltage applied on the motor. An internal timer can be used to count the number of pulses of either Phase A or Phase B, or both of them to calculate an estimated velocity by dividing that number of pulses by a predefined time interval.

The ***H(s)*** in Fig. 4.23 is a transfer function or a coefficient (scalar factor) used to convert the pulse numbers to the positions of the motor.

4.7.2 QEI Modules and Registers in the TM4C123GH6PM MCU System

In TM4C123GH6PM MCU system, two QEI modules, **QEI0** and **QEI1**, are provided to work as interfaces to the increment rotary encoders. A functional block diagram of the QEI modules is shown in Fig. 4.24. Since both modules have the same architecture, only one diagram is provided in this section.

The QEI module supports two operation modes: phase mode and clock/direction mode.

In the phase mode, the encoder produces two clocks that are 90° out of phase. The edge relationship is used to determine the direction of rotation. In clock/direction mode, the encoder produces a clock signal to indicate steps and a direction signal to indicate the direction of rotation of the motor.

When in phase mode, edges on the first channel or edges on both channels can be counted; counting edges on both channels provide higher encoder resolution if required. In either mode, the input signals can be swapped before being processed, allowing wiring mistakes to be corrected without modifying the hardware circuit board.

The index pulse can be used to reset the position counter, allowing the position counter to maintain the absolute encoder position. Otherwise, the position counter maintains the relative position and is never reset.

The velocity capture has a timer to measure equal periods of time. The number of encoder pulses over each time period is accumulated as a measure of the encoder

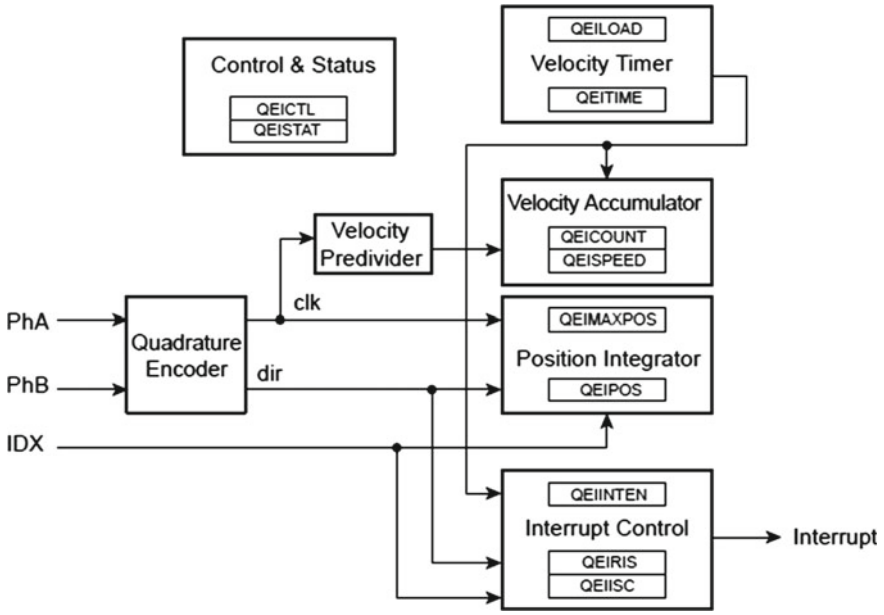


Fig. 4.24 Functional block diagram of the QEI Modules. *Courtesy* Texas instruments

velocity. The running total for the current time period and the final count for the previous time period are available to be read. The final count for the previous time period is usually used as the measured velocity.

The QEI module generates interrupts when the index pulse is detected, when the velocity timer expires, when the encoder direction changes, and when a phase signal error is detected. These interrupt sources can be individually masked so that only the events of interest cause a processor interrupt.

A point to be noted is that both PhA and PhB signals shown in Fig. 4.24 are not the external Phase A and Phase B signals coming from the motor encoder, but instead they are the internal PhA and PhB signals that have passed through inversion and swapping logic that is enabled through the **QEI Control (QEICTL)** register. Figure 4.25 shows this inversion and swapping logic circuit for the external Phase A (**PhAn**) and Phase B (**PhBn**) signals.

Based on Fig. 4.24, each QEI module is composed of related registers and these registers can be divided into different groups based on their functions as shown below:

- QEI control and status registers—control and monitor the QEI running status.
- QEI position registers—calculate the positions of the motor based on Phases A and B.
- QEI velocity registers—calculate the motor velocity based on Phases A and B.
- QEI interrupt processing registers—configure and handle the QEI interrupts.

Now let us have a closer look at these registers.

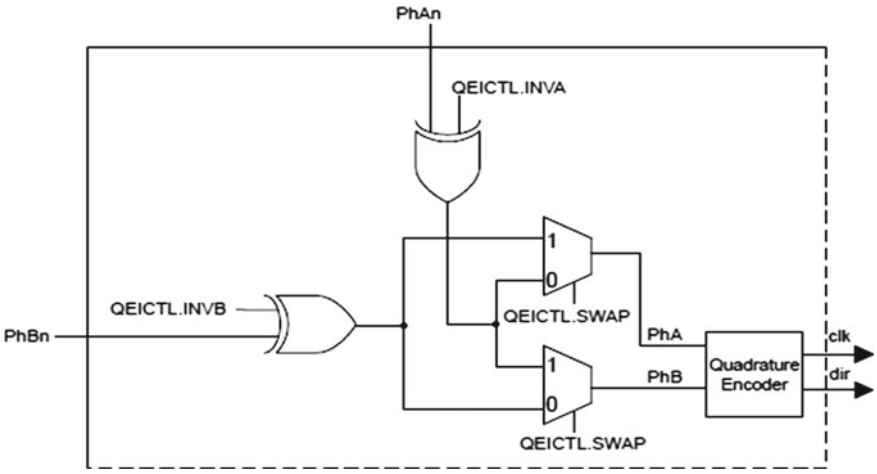


Fig. 4.25 The inversion and swapping logic circuit. *Courtesy* Texas instruments

4.7.2.1 QEI Control and Status Registers

Three registers are included in this group: **QEI Control Register (QEICTL)**, **QEI Status Register (QEISTAT)**, and **Quadrature Encoder Interface Run Mode Clock Gating Control Register (RCGCQEI)**.

The bit field configurations for the **QEICTL** register are shown in Fig. 4.26. The bit function for this register is shown in Table 4.4.

Refer to Figs. 4.25, 4.26, and Table 4.4, it can be found that the inversion and swapping logic circuit shown in Fig. 4.25 is under the control of the bits **INVA**, **INVB**, and **SWAP** or bits 10, 9, and 1 in the **QEICTL** register. When these bit values are set to 1 s, the corresponding external Phase A (**PhAn**) and Phase B (**PhBn**) are inverted and swapped, respectively.

One point to be noted is the **VELDIV** bit field in this register. The value in this bit field is used to perform a pre-divider function for the input quadrature pulses before it can be loaded into the **QEICOUNT** register. The main purpose of this pre-divider is to extend and enlarge the counting range for the input quadrature pulses.

Another important bit in this register is **CAPMODE** bit. As we discussed in Sect. 4.7.1.2, a 2 × resolution can be obtained by only capturing the rising and falling edges of the Phase A signal when this **CAPMODE** is cleared to 0. However,

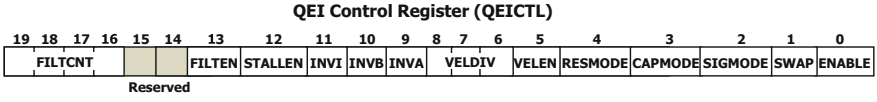


Fig. 4.26 The bit field functions of the QEICTL register

Table 4.4 Bit value and its function for QEICTL register

Bit	Name	Reset	Function
31:20	Reserved	0x0	Reserved
19:16	FILTCNT	0x0	Input filter prescale count: This field controls the frequency of the input update When this field is **0x0**, the input is sampled after two system clocks. When this field ix **0x1**, the input is sampled after three system clocks. Similarly, when this field is **0xF**, the input is sampled after 17 clocks
15:14	Reserved	0x0	Reserved
13	FILTEN	0	Enable Input Filter: **0:** The QEI inputs are not filtered**; 1:** Enable the digital noise filter on the QEI input signals
12	STALLEN	0	Stall QEI: **0:** The QEI module does not stall when the microcontroller is stopped by a debugger **1:** The QEI module stalls when the microcontroller is stopped by a debugger
11	INVI	0	Invert index pulse: **0:** No effect**, 1:** Invert the IDX input
10	INVB	0	Invert PhB: **0:** No effect**, 1:** Invert the PhBn input
9	INVA	0	Invert PhA: **0:** No effect**, 1:** Invert the PhAn input
8:6	VELDIV	0x0	Predivide velocity: This field defines the pre-divider of the input quadrature pulses before being applied to the **QEICOUNT** accumulator **0x0** ÷1; **0x1** ÷2; **0x2** ÷4; **0x3** ÷8; **0x4** ÷16; **0x5** ÷32; **0x6** ÷64; **0x7** ÷128
5	VELEN	0	Capture velocity: **0:** No effect; **1:** Enable capture of the velocity of the quadrature encoder
4	RESMODE	0	Reset mode: **0:** The position counter is reset when it reaches the maximum as defined by the **MAXPOS** field in the **QEIMAXPOS** register **1:** The position counter is reset when the index pulse is captured
3	CAPMODE	0	Capture mode: **0:** Only the PhA edges are counted **1:** The PhA & PhB edges are counted, providing twice the positional resolution but half range When **SIGMODE** = 1, the **CAPMODE** setting is not applicable and is reserved
2	SIGMODE	0	Signal mode: **0:** The internal PhA and PhB signals operate as quadrature phase signals **1:** The internal PhA input operates as the clock (**CLK**) signal and the internal PhB input operates as the direction (**DIR**) signal

(continued)

Table 4.4 (continued)

Bit	Name	Reset	Function
1	SWAP	0	Swap signals: **0:** No effect, **1:** Swaps the PhAn and PhBn signals If the **INVA** or **INVB** bit is set, the inversion of the signals occurs prior to the swap
0	ENABLE	0	Enable QEI: **0:** No effect, **1:** Enables the quadrature encoder module Once, the QEI module has been enabled by setting the **ENABLE** bit, it cannot be disabled. The only way to clear the **ENABLE** bit is to reset the module using the **Quadrature Encoder Interface Software Reset** (**SRQEI**) register

a 4 × resolution can be obtained if both Phase A and Phase B rising and falling edges are captured or counted when this **CAPMODE** bit is set to 1. A point to be noted for this bit is that when **SIGMODE** bit is set to 1, the setting for the **CAPMODE** bit and its function is ignored since the Phase A and Phase B will be working as different roles for that setting. This means that when **SIGMODE** bit is cleared to 0, the internal PhA and PhB signals work together to provide quadrature phase signals as a normal rotary increment encoder did.

However, when this **SIGMODE** bit is set to 1, the internal PhA input operates and provides the clock (**CLK**) step signal and the internal PhB input operates and provides the direction (**DIR**) signal for the detected motor. In this way, we do not need to set up any additional hardware component and circuit to interpret and convert the quadrature signals to the position clock steps and direction signal for the detecting motor.

The **QEI Status Register (QEISTAT)** is used to check and monitor the running status of the QEI modules. This is a 32-bit register but only the lowest two bits, bits 1 and 0, are used to provide the encoder running information and status.

The bit 1, or the **DIRECTION** bit in this register, provides the rotation direction of the encoder; a value of 0 indicates that the encoder is rotating in a forward (clockwise) direction and a value of 1 indicates that the encoder is rotating in a reverse (counterclockwise) direction.

The bit 0, or the **ERROR** bit in this register, provides the running status of the QEI module. 0 means that no error is detected, but 1 indicates that an error is detected for the QEI module.

The **RCGCQEI** is a 32-bit register but only the lowest two bits, bits 1 (**R1**) and 0 (**R0**), are used to enable and clock the QEI modules. If the **R1** bit is set, the QEI module 1 is enabled and clocked. Similarly, if the **R0** bit is set, the QEI module 0 is enabled and clocked.

4.7.2.2 QEI Position Control Registers

Two registers are included in this group: **QEI Position Register** (**QEIPOS**) and **QEI Maximum Position Register (QEIMAXPOS).**

The **QEIPOS** is a 32-bit register and it contains the current value of the position integrator or position accumulator. The value can be updated by the status of the QEI phase inputs and can be set to a specific value by writing to it.

The **QEIMAXPOS** is also a 32-bit register containing the maximum value of the position integrator. When moving forward, the position register **QEIPOS** resets to zero when it increments past this value. When moving in reverse, the position register **QEIPOS** resets to this value when it decrements to zero.

4.7.2.3 QEI Velocity Control Registers

Four registers are coved in this group and they are

- QEI Timer Load Register (**QEILOAD**).
- QEI Timer Register (**QEITIME**).
- QEI Velocity Counter Register (**QEICOUNT**).
- QEI Velocity or Speed Register (**QEISPEED**).

All of these registers are 32-bit registers and used to store related encoder speed information.

The **QEILOAD** register contains the loaded value for the velocity timer. Because this value is loaded into the timer in terms of the clock cycle after the timer is zero, this value should be one less than the number of clocks in the desired period. For example, to have 2000 decimal clocks per timer period, this register should contain 1999 decimal.

The **QEITIME** register contains the current numbers or counts of the clock cycles in this velocity timer. The total counts in the **QEICOUNT** register will be divided by this number of clock cycle to get an estimated encoder speed value.

The **QEICOUNT** register contains the current running counts of velocity pulses for the current time period stored in the **QEITIME** register. Because this count is a running total, the time period to which it applies cannot be known accurately, which means that a reading of this register does not necessarily correspond to the time returned by the **QEITIME** register because there is a small window of time between the two reads and during which either value may have changed. The **QEISPEED** register should be used to determine the actual encoder velocity. Thus, this register only provides velocity-related information.

The **QEISPEED** register contains the most recently measured velocity of the quadrature encoder. This value corresponds to the number of velocity pulses (**QEICOUNT**) counted in the previous velocity timer period (**QEITIME**).

4.7.2.4 QEI Interrupt Processing Registers

Three registers are included in this group and they are

- QEI Interrupt Enable Register (**QEIINTEN**).

- QEI Raw Interrupt Status Register (**QEIRIS**).
- QEI Interrupt Status and Clear Register (**QEIISC**).

All of these three registers are 32-bit registers but only the lowest 4 bits, bits 3–0, are used to configure, set up, and clear four related QEI module operational interrupts that are given below:

- Phase error interrupt,
- Direction change interrupt,
- Timer expires interrupt, and
- Index pulse detected interrupt.

Figure 4.27 shows the bit fields for these registers.

If any of the lowest four bits on these registers is set to one, which means that the selected interrupt is enabled for the **QEIINTEN** register, the corresponding interrupt is generated for the **QEIRIS** register, and the related interrupt can be cleared for the **QEIISC** register.

Otherwise, if any bit on these lowest four bits is zero, which means that the selected interrupt is disabled for the **QEIINTEN** register, the corresponding interrupt is not generated for the **QEIRIS** register, and the related interrupt has not been cleared for the **QEIISC** register if that interrupt has occurred.

4.7.2.5 The QEI Interfacing Signals and Related GPIO Pins

Table 4.5 shows the QEI module external signals and related GPIO pin distributions.

Like other peripheral devices, the QEI signals are alternate functions for some related GPIO signals on the associated GPIO pins. The column in Table 4.5 titled **GPIO pin** lists the possible GPIO pin placements for these QEI signals. The **AFSEL** bits in the **GPIOAFSEL** register should be set to choose the QEI function. The number in parentheses is the encoding that must be programmed into the **PMCx** field in the **GPIOPCTL** register to assign the QEI signal to the specified GPIO port pin.

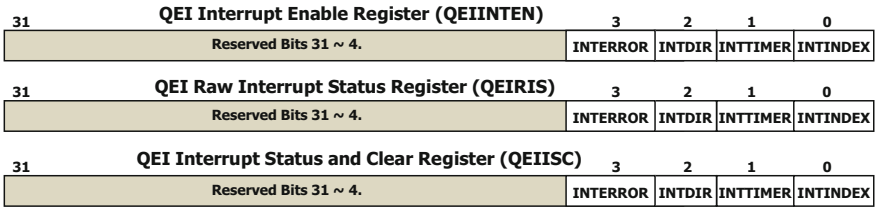


Fig. 4.27 QEI interrupt processing registers

Table 4.5 QEI external control signals and GPIO pin distributions

QEI Pin	GPIO pin	Pin type	Buffer type	Pin function
IDX0	PF4 (6) PD3 (6)	I	TTL	QEI Module 0 index
IDX1	PC4 (6)	I	TTL	QEI Module 1 index
PhA0	PF0 (6) PD6 (6)	I	TTL	QEI Module 0 Phase A
PhA1	PC5 (6)	I	TTL	QEI Module 1 Phase A
PhB0	PD7 (6) PF1 (6)	I	TTL	QEI Module 0 Phase B
PhB1	PC6 (6)	I	TTL	QEI Module 1 Phase B

4.7.2.6 The QEI Initialization and Configuration Process

Before the rotary encoder can be used in any application, the QEI module should be selected and initialized to make it ready to measure the desired motor rotating position and speed. Regularly, the initialization and configuration process should include the following steps:

1. Enable the clock for the QEI module by configuring the **RCGCQEI** register.
2. Enable the clock for the appropriate GPIO module via the **RCGCGPIO** register (refer to Table 4.5).
3. Configure the appropriate GPIO pins by setting the related bits on the **GPIOAFSEL** register to enable related pins to work as alternate functions (refer to Table 4.5).
4. Configure the appropriate bit field in the **GPIOPCTL** register to enable selected GPIO pins to work as QEI functions (refer to number inside the parentheses in Table 4.5).
5. Configure the QEI module to capture both rising and falling edges for the **PhAn** and **PhBn** signals and maintain an absolute position by resetting on index pulses. For a 1000-line encoder with four edges per line, it can generate 4000 pulses per revolution. Therefore, set the maximum position to 3999 (4000-1) since the count is zero based. Perform the following two writing operations to complete this configuration:
   - Write the **QEICTL** register with the value of **0x28** to enable the capture mode to capture the velocity of the motor.
   - Write the **QEIMAXPOS** register with the value of **3999** as the maximum position value.
6. Enable the QEI module by setting bit 0 (**ENABLE**) of the **QEICTL** register. The point to be noted is that once the QEI module has been enabled by setting the **ENABLE** bit in the QEICTL register, it cannot be disabled. The only way to clear the **ENABLE** bit is to reset the module using the **Quadrature Encoder**

Interface Software Reset (SRQEI) register. A two-step operation is needed to complete this reset operation.
7. Delay a period of time until the encoder position is required.
8. Read the encoder position by reading the **QEI Position (QEIPOS)** register value. One point to be noted is that if the application requires the QEI module to have a specific initial position, this value must be programmed in the **QEIPOS** register after the QEI module has been enabled by setting the **ENABLE** bit in the **QEICTL** register.
9. Read the velocity of the motor rotation by reading the **QEI Velocity (QEISPEED)** register value.

Before we can start to perform the identification process for our DC motor, we need to know some fundamental knowledge and control functions for pulse-width modulation technique used in the TM4C123GH6PM MCU system since we need to use this module to access and control DC motor to perform some data collection process for our identification purpose.

4.7.3 PWM Modules in the TM4C123GH6PM MCU System

The Pulse-Width-Modulation (PWM) is a powerful technique to enable a digital signal to be encoded to an analog signal to drive a power supply or a DC servo or step motor. This technique is widely implemented in most power supplies and motors as well as robots control applications. In order to have a full understanding and clear picture about this technique, let us first have a closer look at the principle of the PWM technique.

4.7.3.1 The PWM Principle and Implementations

In all modern power supply products, most of them use the PWM idea to build power supplies. The so-called PWM is to modify or modulate a periodic rectangular digital waveform to make it with a different pulse width or duty cycle, and therefore to generate different magnitudes on the output analog signal.

Compare two PWM signals, S_1 and S_2, shown in Fig. 4.28a, b. Both signals have the same period and frequency, but the average output magnitudes M_1 for S_1 and M_2 for S_2 are different since they have different duty cycles, 25 and 50%.

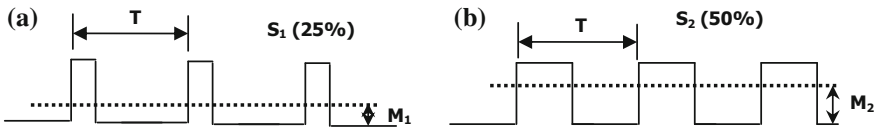


Fig. 4.28 An example of PWM signals

In actual circuits, this voltage average function is performed by capacitors that play an integrator role or a low-pass filter.

The PWM circuit can be easily programmed to modify the output magnitudes for their outputs. The higher the duty cycle (the wider the pulse width), the higher the output voltage. The lower the duty cycle (the narrower the pulse width), the lower the output voltage.

Now let us take a look at the PWM circuits in the TM4C123GH6PM MCU system.

4.7.3.2 PWM Modules Applied in the TM4C123GH6PM MCU System

The TM4C123GH6PM MCU contains two PWM modules, PWM0 and PWM1. Each module has four PWM generator blocks and a control block, and each generator block can create two PWM output signals, and therefore a total of 16 PWM output signals can be generated by these two modules. The control block determines the polarity of the PWM signals, and which signals are passed through to the output pins.

Each PWM generator block produces two PWM signals that share the same timer (counter) and frequency and can either be programmed with independent actions or as a single pair of complementary signals with dead-band delays inserted. The output signals, **pwmA′** and **pwmB′**, of the PWM generation blocks are managed by the output control block before being passed to the device pins as **MnPWM0** and **MnPWM1** or **MnPWM2** and **MnPWM3**, and so on.

Each TM4C123GH6PM PWM module provides a great deal of flexibility and can generate simple PWM signals, such as those required by a simple charge pump as well as paired PWM signals with dead-band delays, such as those required by a half H-bridge driver.

In the TM4C123GH6PM MCU system, each PWM generator block is composed of two components, one 16-bit counter or timer and two PWM comparators, cmpA and cmpB.

The counter provides a timing base for two comparators and it is mainly used to perform count-down or count-up/down functions to provide a comparison source for two comparators.

The PWM Counter (Timer)

The 16-bit counter or timer in each PWM generator runs in one of two modes, count-down mode or count-up/down mode:

- In count-down mode, the timer counts from the LOAD value to 0, and then it goes back to the LOAD value and continues to perform the counting down.
- In count-up/down mode, the timer counts from 0 up to the LOAD value, and then counts down back to 0 back up to the LOAD value, and so on.

Generally, count-down mode is used for generating left-aligned or right-aligned PWM signals, and the count-up/down mode is used to generate center-aligned PWM signals.

The timers output three signals that can be used in the PWM generation process as given below:

(1) The direction signal. This signal is always low in count-down mode, but alternates between Low and High in count-up/down mode.
(2) A single-clock-cycle-width High pulse when the counter gets zero.
(3) A single-clock-cycle-width High pulse when the counter is equal to the load value.

Note that in the count-down mode, the zero pulse is immediately followed by the load pulse. These signals are labeled **dir**, **zero**, and **load** in all figures in the following sections.

The PWM Comparators

As we mentioned, each PWM generator has two comparators that monitor the value of the counter and output a single-clock-cycle-width High when either comparator's value is equal to the counting value in the counter. These outputs are labeled **cmpA** and **cmpB** in all figures in this chapter. In the count-up/down mode, these comparators match both when counting up and when counting down, and thus are qualified by the counter direction signal. If either comparator match value is greater than the counter load value that comparator never outputs a High pulse.

Figure 4.29a shows the behavior of the counter and the relationship of these pulses when the counter is in count-down mode. Figure 4.29b shows the behavior of the counter and the relationship of these pulses when the counter is in count-up/down mode.

The following definitions are applied for these operational modes in this figure:

- **LOAD** is the value stored in the **PWMnLOAD** register.
- **COMPA** is the value stored in the **PWMnCMPA** register.
- **COMPB** is the value stored in the **PWMnCMPB** register.
- **0** is the value zero.
- **load** is the internal signal that provides a single-clock-cycle-width High pulse when the counter is equal to the load value in the counter.
- **zero** is the internal signal that provides a single-clock-cycle-width High pulse when the counter is zero.
- **cmpA** is the internal signal that provides a single-clock-cycle-width High pulse when the counter is equal to **COMPA**.
- **cmpB** is the internal signal that provides a single-clock-cycle-width High pulse when the counter is equal to **COMPB**.
- **dir** is the internal signal that indicates the count direction.

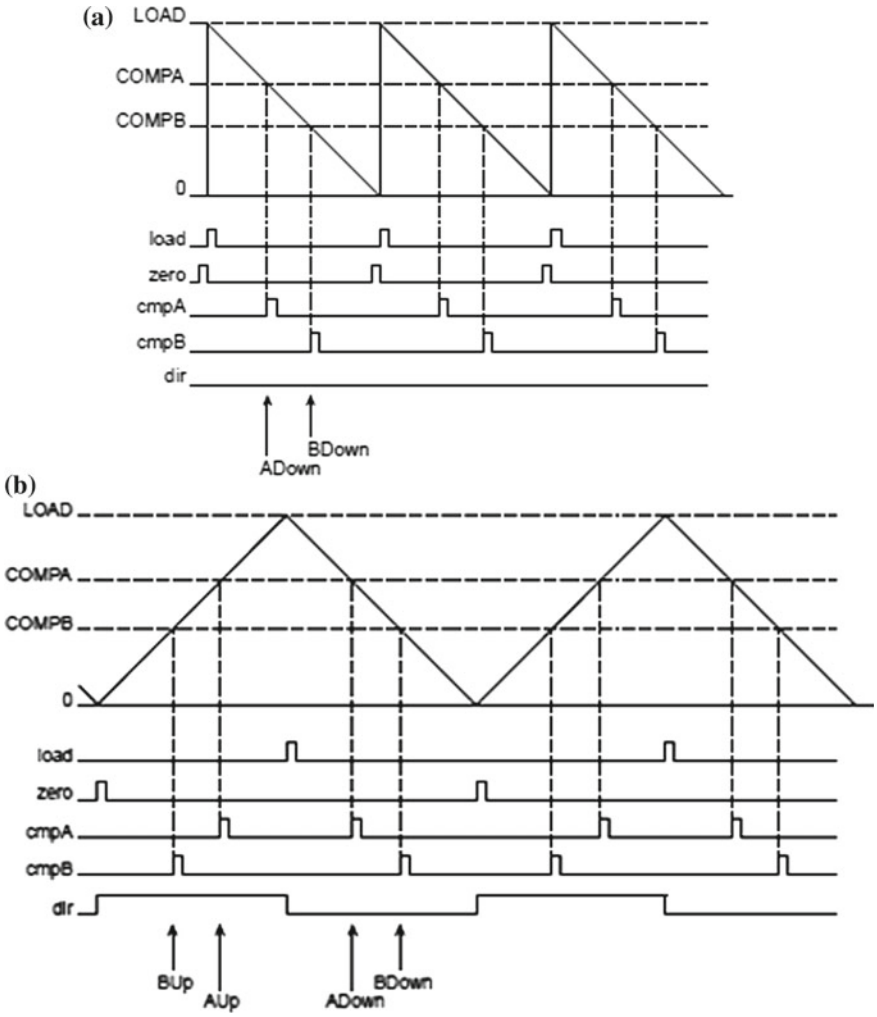


Fig. 4.29 PWM count-down and count up/down modes. *Courtesy* Texas Instruments

Based on these elements and internal signals, now let us have a closer look at how to use them to generate PWM output signals like those shown in Fig. 4.28.

The PWM Output Signal Generator

Each PWM generator uses the **load**, **zero**, **cmpA**, and **cmpB** pulses the **dir** signal to generate two internal PWM signals, **pwmA** and **pwmB**, as shown in Fig. 4.29.

In count-down mode, four events can affect these signals: **zero**, **load**, match A down, and match B down. In count-up/down mode, six events can affect these

signals: **zero**, **load**, match A down, match A up, match B down, and match B up. The match A or match B events are ignored when they coincide with the **zero** or **load** events. If the match A and match B events coincide, the first signal, **pwmA**, is generated based only on the match A event, and the second signal, **pwmB**, is generated based only on the match B event.

Each event can affect each output PWM signal by programming to make the output to be

- Left alone (ignoring the event),
- Toggled, and
- Driven Low or High.

These actions can be used to generate a pair of PWM signals of various positions and duty cycles, which do or do not overlap. Figure 4.30 shows the use of count-up/down mode to generate a pair of center-aligned, overlapped PWM signals that have different duty cycles. The **pwmA** and **pwmB** signals shown in this figure are before they have passed through the dead-band generator.

In Fig. 4.30, the first PWM generator is set to output High on match A up (**COMPA**), output Low on match A down (**COMPA**), and ignore the other four events. The second PWM generator is set to output High on match B up (**COMPB**), output Low on match B down (**COMPB**), and ignore the other four events. Changing the value of comparator A (**COMPA**) changes the duty cycle of the **pwmA** signal and changing the value of comparator B (**COMPB**) changes the duty cycle of the **pwmB** signal.

In addition to normal PWM outputs, these two PWM output signals can be combined together to form a so-called dead-band output to drive a half H-bridge circuit for some motors.

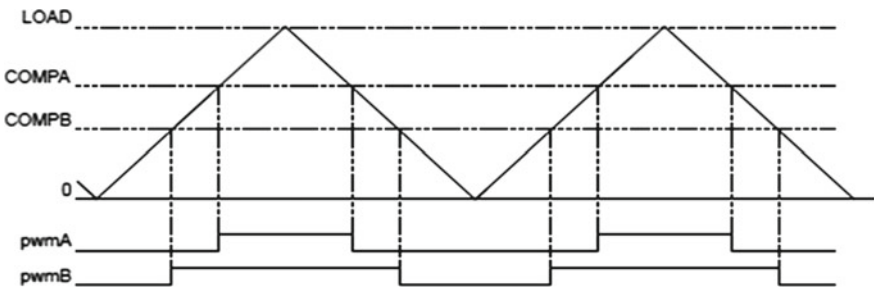


Fig. 4.30 An example of using the count-up/down mode to generate PWM outputs. *Courtesy* Texas Instruments

4.7.3.3 PWM Generator Functional Block Diagram

First let us have a closer look at the PWM generator block to see how this block creates two PWM output signals based on the discussion we made above.

Figure 4.31 shows a detailed structure for a PWM generator block. This block contains a 16-bit counter or timer and two comparators as we discussed above.

In fact, both the counter (timer) and two comparators are under the control of the register **PWMnCTL**, and each generator block contains one of these kinds of controllers to configure the setups and control the operations of each generator.

It can be found from Fig. 4.31, five events or internal signals, **zero**, **load**, **dir**, **cmpA**, and **cmpB**, are combined and work together to generate two PWM output signals, **pwmA** and **pwmB**. Depending on the availability (enabled or disabled) of the dead-band generator, these two PWM signals can be directly passed to the outputs, **pwmA′**, and **pwmB′** if the dead-band generator is disabled, or a dead-band signal can be generated if it is enabled.

These five events or internal signals can also be used to generate PWM-related interrupts to inform the processor that some events related to PWM have occurred.

The **Interrupt and Trigger Generator** block is used to configure and control the generating and handling of PWM-related interrupts and trigger source selections process. The **Fault Condition** block is used to configure and control any fault condition and generation process for the selected PWM generator.

Now let us take care of some useful registers in some block to see their functions.

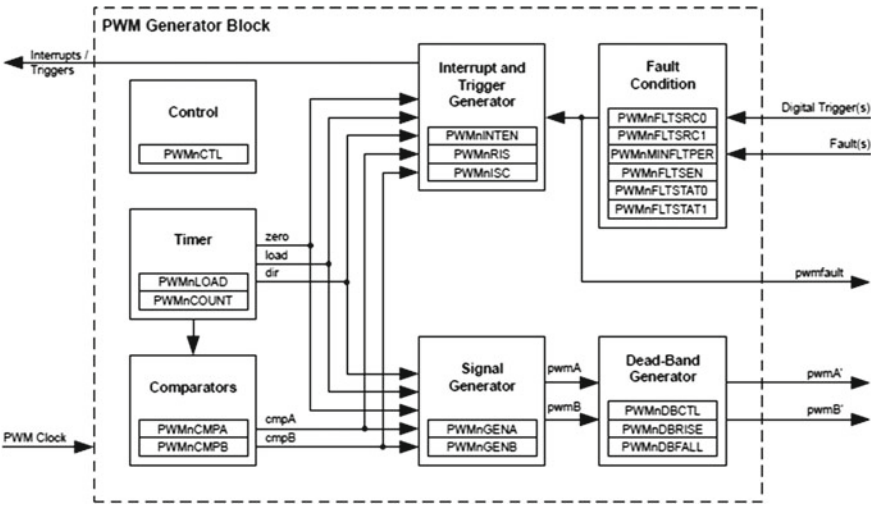


Fig. 4.31 The detailed block diagram for the PWM generator block. *Courtesy* Texas Instruments

PWM Generator Block Control Register (**PWMnCTL**)

As we mentioned, two PWM modules, PWM0 and PWM1, are installed and applied for the TM4C123GH6PM MCU system and each PWM module contains four PWM generator blocks. Each PWM generator block has an independent control register. For example, for the PWM0 module, four block control registers are **PWM0CTL** ~ **PWM0CTL3**, each one controls one generator block with the same bit field distribution.

The bit field values for these registers are shown in Fig. 4.32. The bit field value and related function of these registers are shown in Table 4.6.

These registers are used to configure the PWM signal generation blocks. The **PWM0CTL** controls the PWM generator 0 block, **PWM1CTL** controls the PWM generator 1 block, and so on. The blocks produce the PWM signals that can be either two independent PWM signals from the same counter or a paired set of PWM signals with dead-band delays added.

Each generator block can create two PWM output signals, and therefore the PWM0 block produces the **MnPWM0** and **MnPWM1** outputs, the PWM1 block produces the **MnPWM2** and **MnPWM3** outputs, the PWM2 block produces the **MnPWM4** and **MnPWM5** outputs, and the PWM3 block produces the **MnPWM6** and **MnPWM7** outputs.

Although there are so many parameters in this register to be configured, in fact only a few of them are important to us and they are bits **ENABLE**, **MODE**, **CMPAUPD**, **CMPBUPD**, and **LOADUPD**. Some other update bits can be configured as an immediate mode to allow them to take effects immediately if no synchronized operations are needed.

PWM Generator Block Load Register (**PWMnLOAD**)

These registers are 32-bit registers but only the lower 16 bit is used to meet the need of the 16-bit counter. Each of these registers (**PWM0LOAD** ~ **PWM3LOAD**) contains the load value for the PWM counter and controls the related PWM generator block. Based on the counter mode configured by the **MODE** bit in the **PWMnCTL** register, this value is either loaded into the counter after it reaches zero (**MODE** = 0: Count-down) or is the limit of up-counting after which the counter

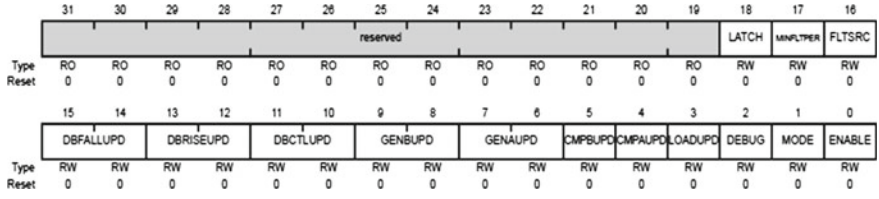


Fig. 4.32 Bit fields in the PWM generator control register. *Courtesy* Texas Instruments

Table 4.6 Bit field value and its function for PWMnCTL registers

Bit	Name	Reset	Function
31:19	Reserved	0×0	Reserved
18	LATCH	0	Latch fault input **0**: Fault condition not latched. **1**: Fault condition latched
17	MINFLTPER	0	Minimum fault period **0:** The FAULT input deassertion is unaffected **1:** The **PWMnMINFLTPER** one-shot counter is active and extends the period of the fault condition to a minimum period
16	FLTSRC	0	Fault condition source **0**: The fault condition is determined by the **Fault0** input **1:** The fault condition is determined by the configuration of the **PWMnFLTSRC0** and **PWMnFLTSRC1** registers
15:14	DBFALLUPD	0×0	**PWMnDBFALL** update mode **0x0:** Immediate. **0x1:** Reserved. **0x2:** Locally synchronized (update to the register is reflected to the generator the next time the counter is 0). **0x3:** Globally synchronized (update to the register is delayed until the next time the counter is 0 after a synchronous update has been requested through the **PWMCTL** register)
13:12	DBRISEUPD	0×0	**PWMnDBRISE** update mode. Same as **PWMnDBFALL** update mode above
11:10	DBCTLUPD	0×0	**PWMnDBCTL** update mode. Same as **PWMnDBFALL** update mode above
9:8	GENBUPD	0×0	**PWMnGENB** update mode. Same as **PWMnDBFALL** update mode above
7:6	GENAUPD	0×0	**PWMnGENA** update mode. Same as **PWMnDBFALL** update mode above
5	CMPBUPD	0	Comparator B update mode **0:** Locally synchronized (updates to the **PWMnCMPB** register is reflected to the generator the next time the counter is 0) **1:** Globally synchronized (update to the register is delayed until the next time the counter is 0 after a synchronous update has been requested via the **PWMCTL** register)
4	CMPAUPD	0	Comparator A update mode. Same as Comparator B update mode above
3	LOADUPD	0	Load register update mode. Same as Comparator B update mode above

(continued)

Table 4.6 (continued)

Bit	Name	Reset	Function
2	DEBUG	0	Debug mode **0:** The counter stops running when it next reaches 0 and continues to run again when no longer in debug mode. **1:** The counter always runs when in debug mode
1	MODE	0	Counter mode. **0:** Count-down mode. **1:** Count-up/Down mode
0	ENABLE	0	PWM block enable **0:** Entire block is disabled (no clock). **1:** Entire block is enabled and clocked

decrements back to zero (**MODE** = 1: count-up/down). When this value matches the counter, a High pulse is output that can be configured to drive the generation of the **pwmA** and/or **pwmB** signal via the **PWMnGENA** or **PWMnGENB** register, or drive an interrupt ADC trigger via the **PWMnINTEN** register.

If the load value update mode is locally synchronized (**LOADUPD** = 0 in the **PWMnCTL** register), the 16-bit **LOAD** value is used the next time the counter reaches zero. If the update mode is globally synchronized, it is used the next time the counter reaches zero after a synchronous update has been requested through the **PWM Master Control (PWMCTL)** register. If this register is rewritten before the actual update occurs, the previous value is never used and is lost.

PWM Generator Block Count Register (**PWMnCOUNT**)

These registers are 32-bit registers but only the lower 16 bit is used to perform the counting function. Each of these registers (**PWM0COUNT** ~ **PWM3COUNT**) contains the current value of the PWM counter for the related PWM generator block. When this value matches zero or the value in the **PWMnLOAD**, **PWMnCMPA**, or **PWMnCMPB** registers, a pulse is an output that can be configured to drive the generation of a PWM signal or drive an interrupt or ADC trigger.

The point to be noted is that disabling the PWM by clearing the **ENABLE** bit in the **PWMnCTL** register does not clear the **COUNT** field (lower 16 bit) in the **PWMnCOUNT** register. Before re-enabling the PWM (**ENABLE** = 1), the **COUNT** field should be cleared by resetting the PWM registers through the **Pulse Width Modulator Software Reset (SRPWM)** register in the system control module.

PWM Generator Block Comparator a Register (**PWMnCMPA**)

These registers are 32-bit registers but only the lower 16 bit is used to store a predefined value to be compared with the value in the counter. Each of these registers (**PWM0CMPA** ~ **PWM3CMPA**) contains a value to be compared

against the counter for the related PWM generator block. When this value matches the counter, a pulse is an output that can be configured to drive the generation of the **pwmA** and **pwmB** signals via the **PWMnGENA** and **PWMnGENB** registers, or drive an interrupt or ADC trigger via the **PWMnINTEN** register. If the value of this register is greater than the **PWMnLOAD** register, then no pulse is ever output.

If the Comparator A update mode is locally synchronized (**CMPAUPD** = 0 in the **PWMnCTL** register), the 16-bit **COMPA** value is used the next time the counter reaches zero. If the update mode is globally synchronized (**CMPAUPD** = 1), it is used the next time the counter reaches zero after a synchronous update has been requested through the **PWM Master Control (PWMCTL)** register. If this register is rewritten before the actual update occurs, the previous value is never used and is lost.

PWM Generator Block Comparator B Register (**PWMnCMPB**)

Similar to **PWMnCMPA**, these registers contain a value to be compared against the counter for the related PWM generator block. When this value matches the counter, a pulse is an output that can be configured to drive the generation of the **pwmA** and **pwmB** signals via the **PWMnGENA** and **PWMnGENB** registers, or drive an interrupt or ADC trigger via the **PWMnINTEN** register. If the value of this register is greater than the **PWMnLOAD** register, no pulse is ever output.

If the Comparator B update mode is locally synchronized (**CMPBUPD** = 0 in the **PWMnCTL** register), the 16-bit **COMPB** value is used the next time the counter reaches zero. If the update mode is globally synchronized, it is used the next time the counter reaches zero after a synchronous update has been requested through the **PWM Master Control (PWMCTL)** register.

If this register is rewritten before the actual update occurs, the previous value is never used and is lost.

PWM Generator A Register (**PWMnGENA**)

These registers are 32-bit registers but only the lower 12 bit is used to control the generation of the **pwmA** signal based on the **load** and **zero** output pulses from the counter as well as the compare A and compare B pulses from the comparators. Each of these registers, **PWM0GENA** ~ **PWM3GENA**, controls the related PWM generator blocks. When the counter is running in count-down mode, only four of these events occur, when running in count-up/down mode, all six occur. The lower 12 bits on this register is divided into six 2-bit segments and each segment can be configured to take a related action when a match condition has occurred.

The bit field values for these registers are shown in Fig. 4.33. The bit field value and related function of these registers are shown in Table 4.7.

The **PWM0GENA** register controls generation of the **pwm0A** signal, the **PWM1GENA** is used to control the generation of the **pwm1A** signal, the

PWM2GENA is for the **pwm2A** signal, and **PWM3GENA** is for the **pwm3A** signal.

If a zero or load event coincides with a Compare A or Compare B event, the zero or load action is taken and the Compare A or Compare B action is ignored. If a Compare A event coincides with a Compare B event, the Compare A action is taken and the Compare B action is ignored.

If the Generator A update mode is immediate (the **GENAUPD** field = 00 in the **PWMnCTL** register), the **ACTCMPBD**, **ACTCMPBU**, **ACTCMPAD**, **ACTCMPAU**, **ACTLOAD**, and **ACTZERO** values are used immediately. If the update mode is locally synchronized, these values are used the next time the counter reaches zero. If the update mode is globally synchronized, these values are used the next time the counter reaches zero after a synchronous update has been requested through the **PWM Master Control (PWMCTL)** register. If this register is

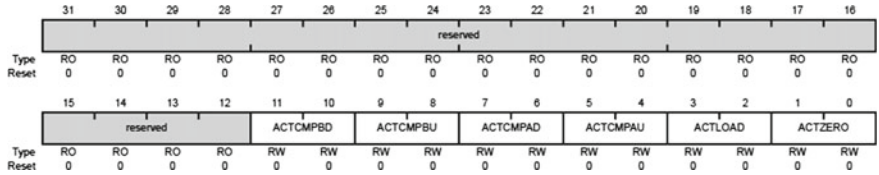


Fig. 4.33 Bit fields in the PWM generator A register. *Courtesy* Texas Instruments

Table 4.7 Bit field value and its function for PWM Generator A register

Bit	Name	Reset	Function
31:12	Reserved	0×0	Reserved
11:10	ACTCMPBD	0×0	Action for Comparator B down **0x0**: Do nothing **0x1**: Invert pwmA **0x2:** Drive pmwA Low **0x3:** Drive pwmA High
9:8	ACTCMPBU	0×0	Action for Comparator B up **0x0**: Do nothing **0x1**: Invert pwmA **0x2:** Drive pmwA Low **0x3:** Drive pwmA High
7:6	ACTCMPAD	0×0	Action for Comparator A down **0x0**: Do nothing **0x1**: Invert pwmA **0x2:** Drive pmwA Low **0x3:** Drive pwmA High
5:4	ACTCMPAU	0×0	Action for Comparator A up **0x0**: Do nothing **0x1**: Invert pwmA **0x2:** Drive pmwA Low **0x3:** Drive pwmA High
3:2	ACTLOAD	0×0	Action for Counter = LOAD **0x0**: Do nothing **0x1**: Invert pwmA **0x2:** Drive pmwA Low **0x3:** Drive pwmA High
1:0	ACTZERO	0×0	Action for Counter = 0 **0x0**: Do nothing **0x1**: Invert pwmA **0x2:** Drive pmwA Low **0x3:** Drive pwmA High

rewritten before the actual update occurs, the previous value is never used and is lost.

PWM Generator B Register (**PWMnGENB**)

Similar to PWM Generator A register, the PWM Generator B register plays the similar function. The only difference between them is that the PWM Generator B register is used to control the generation of the **pwmB** signal based on the **load** and **zero** output pulses from the counter as well as the Compare A and Compare B pulses from the comparators. Each of these registers, **PWM0GENB** ~ **PWM3GENB**, controls the related PWM generator blocks.

The **PWM0GENB** register controls generation of the **pwm0B** signal, the **PWM1GENB** is used to control the generation of the **pwm1B** signal, the **PWM2GENB** is for the **pwm2B** signal, and **PWM3GENB** is for the **pwm3B** signal.

The bit field values for these registers are identical with those shown in Fig. 4.33. The bit field value and related function of these registers are shown in Table 4.8. It can be found from Table 4.8 that the bit field values in this register are identical with those in PWM Generator A register and the only difference is that the control target signal is **pwmB**.

Table 4.8 Bit field value and its function for PWM Generator B register

Bit	Name	Reset	Function
31:12	Reserved	0×0	Reserved
11:10	ACTCMPBD	0×0	Action for Comparator B down **0x0**: Do nothing **0x1**: Invert pwmB **0x2:** Drive pmwB Low **0x3:** Drive pwmB High
9:8	ACTCMPBU	0×0	Action for Comparator B up **0x0**: Do nothing **0x1**: Invert pwmB **0x2:** Drive pmwB Low **0x3:** Drive pwmB High
7:6	ACTCMPAD	0×0	Action for Comparator A down **0x0**: Do nothing **0x1**: Invert pwmB **0x2:** Drive pmwB Low **0x3:** Drive pwmB High
5:4	ACTCMPAU	0×0	Action for Comparator A up **0x0**: Do nothing **0x1**: Invert pwmB **0x2:** Drive pmwB Low **0x3:** Drive pwmB High
3:2	ACTLOAD	0×0	Action for Counter = LOAD **0x0**: Do nothing **0x1**: Invert pwmB **0x2:** Drive pmwB Low **0x3:** Drive pwmB High
1:0	ACTZERO	0×0	Action for Counter = 0 **0x0**: Do nothing **0x1**: Invert pwmB **0x2:** Drive pmwB Low **0x3:** Drive pwmB High

4.7.3.4 PWM Module Architecture and Functional Block Diagram

Figure 4.34 shows a functional block diagram of the PWM modules used in the TM4C123GH6PM MCU system.

The entire PWM module is composed of the following components:

- Control and status block.
- PWM generator blocks.
- Output control block.
- Interrupt control block.

We have provided a detailed discussion about the PWM generator blocks, now let us concentrate on the rest of the three blocks.

The Control and Status Block

This block provides the global or master controls to all four PWM generators and monitors the running status of them. This block is driven by a system clock as the clock source. Generally, the PWM module has two kinds of clock sources to be used that are given below:

- The system clock and
- A pre-divided system clock.

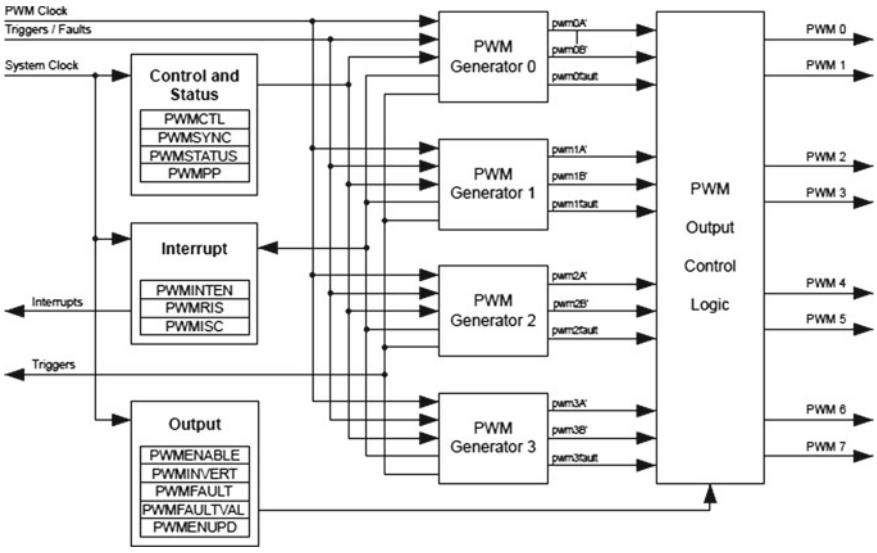


Fig. 4.34 Architecture and functional block diagram of PWM module. *Courtesy* Texas Instruments

The clock source can be selected by programming the **USEPWMDIV** bit in the **run-mode clock configuration (RCC)** register. The **PWMDIV** bit field specifies the divisor of the system clock that is used to create the PWM clock.

The control functions in this block are provided by four registers as follows:

(1) PWM Master Control Register (**PWMCTL**),
(2) PWM Timer Base Synchronous Register (**PWMSYNC**),
(3) PWM Status Register (**PWMSTATUS**), and
(4) PWM Peripheral Properties Register (**PWMPP**).

Let us take a closer look at these registers one by one. First, let us have a closer look at the **run-mode clock configuration (RCC)** register.

The Run-Mode Clock Configuration Register (RCC)

We have discussed this register in Sect. 3.2.1.2.1 in Chap. 3. In this section, we only concentrate on the bit field **PWMDIV** in this register.

Figure 4.35 shows the bit fields on this register. The **PWMDIV** bit field includes bits 19–7. Table 4.9 shows this bit field and its function. After a system reset operation, the default value for this bit field is **0** × **7**, which means that the PWM clock source is the system clock that is divided by 64.

The PWM Master Control Register (PWMCTL)

This register is a master control register used to control four PWM generator blocks.

Two PWM modules, each one has an independent **PWMCTL** register and it is mainly used to control the updating and synchronizing of four PWM generator blocks.

Although this is a 32-bit register, only the lower four bits are used and each bit is used to control one PWM generator block.

The bit field values of this register are shown in Fig. 4.36.

The bit field function for each related PWM generator is similar. When the bit is 0, it means that no updating or synchronizing action is needed for the selected PWM generator. If the bit is 1, it indicates that any updating to a load or comparator

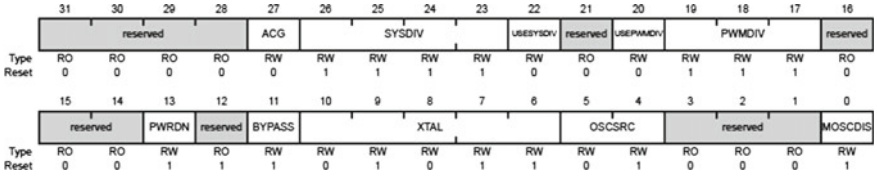


Fig. 4.35 Bit fields in the run-mode clock configuration (RCC) register. *Courtesy* Texas Instruments

Table 4.9 The PWMDIV bit field value and its function in RCC register

Bit	Name	Reset	Function
31:28	Reserved	0×0	Reserved
19:17	PWMDIV	0×7	PWM unit clock divisor. **0x0:** ÷2. **0x1:** ÷4. **0x2:** ÷8. **0x3:** ÷16. **0x4:** ÷32. **0x5:** ÷64. **0x6:** ÷64. **0x7:** ÷64 (default)

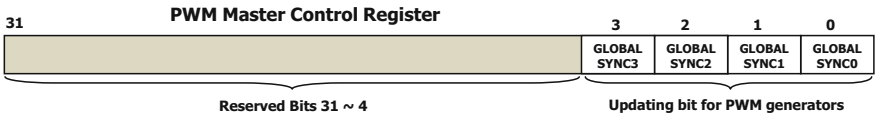


Fig. 4.36 Bit fields in the PWM master control register

register in the related PWM generator is applied the next time the corresponding counter becomes zero.

The bit **GLOBALSYNC0** is for the PWM Generator 0 block, **GLOBALSYNC1** is for the PWM Generator 1 block, and so on.

The PWM Timer Base Synchronous Register (PWMSYNC)

Similar to the PWM master control register, each PWM module has an independent PWM **Timer Base Synchronous (PWMSYNC)** register. The lower 4 bits on this register are used to control four PWM generator blocks.

This register provides a method to perform synchronization of the counters in the PWM generation blocks. Setting a bit in this register causes the specified counter to reset back to 0 while setting multiple bits reset multiple counters simultaneously. The bits are auto-cleared after the reset has occurred. Reading them back as zero indicates that the synchronization has completed.

The bit fields on this register are identical with those in the **PWMCTL** register shown in Fig. 4.36. The difference is that the four lower bits are **SYNC3** ~ **SYNC0**, and each bit controls the synchronization for one related PWM generator block.

The Output Control Block and Its Registers

The output control block contains five control registers as given below:

(1) PWM Output Enable Register (**PWMENABLE**),
(2) PWM Output Inversion Register (**PWMINVERT**),
(3) PWM Output Fault Register (**PWMFAULT**),

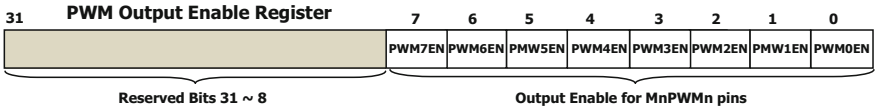


Fig. 4.37 Bit fields in the PWM output enable (PWMENABLE) register

(4) PWM Fault Condition Value Register (**PWMFAULTVAL**), and
(5) PWM Enable Update Register (**PWMENUPD**).

Let us discuss these registers one by one in more details.

The PWM Output Enable Register (PWMENABLE)

Similar to the PWM master control register, each PWM module has an independent **PWM Output Enable (PWMENABLE)** register. The lower 8 bits on this register are used to control whether the **pwmA′** and **pwmB′** signals can be transferred to eight PWM output pins **MnPWMn**.

The bit field values of this register are shown in Fig. 4.37.

This register provides a master control of which generated **pwmA′** and **pwmB′** signals are output to the MnPWMn pins. By disabling a PWM output, the generation process can continue without driving PWM signals to the pins. When related bits in this register are set to 1 s, the corresponding **pwmA′** or **pwmB′** signal is passed through to the output stage.

When bits are clear, the **pwmA′** or **pwmB′** signal is replaced by a zero value which is also passed to the output stage.

The bit field value and related function of this register are shown in Table 4.10.

4.7.3.5 PWM Module Components and Signal Descriptions

As we have discussed in the last section, each PWM module is composed of four blocks. The control and status block provides the global control for four PWM generator blocks and monitors their running status. The output control block controls and directs related **pwmA** and **pwmB** signals to the related output **MnPWMn** pins via dead-band block. The interrupt block provides the interrupt generations and status for PWM generators.

All PWM output pins **MnPWMn** can be used to drive PWM-related peripherals via the related GPIO pins. In other words, any **MnPWMn** pin cannot be directly exposed to outside peripheral devices and must use the related GPIO pin as an interface to be connected to any peripheral device.

In TM4C123GH6PM MCU system, the related GPIO ports and pins that are connected to different **MnPWMn** pins are shown in Table 4.11.

Table 4.10 Bit field value and its function for PWM output enable register

Bit	Name	Reset	Function
31:8	Reserved	0×0	Reserved
7	PWM7EN	0	**MnPWM7** output enable **0:** The **MnPWM7** pin has a zero value **1:** The generated **pwm3B'** signal is passed to the **MnPWM7** pin
6	PWM6EN	0	**MnPWM6** output enable **0:** The **MnPWM6** pin has a zero value **1:** The generated **pwm3A'** signal is passed to the **MnPWM6** pin
5	PWM5EN	0	**MnPWM5** output enable 0: The **MnPWM5** pin has a zero value 1: The generated **pwm2B'** signal is passed to the **MnPWM5** pin
4	PWM4EN	0	**MnPWM4** output enable 0: The **MnPWM4** pin has a zero value 1: The generated **pwm2A'** signal is passed to the **MnPWM4** pin
3	PWM3EN	0	**MnPWM3** Output Enable 0: The **MnPWM3** pin has a zero value 1: The generated **pwm1B'** signal is passed to the **MnPWM3** pin
2	PWM2EN	0	**MnPWM2** output enable 0: The **MnPWM2** pin has a zero value 1: The generated **pwm1A'** signal is passed to the **MnPWM2** pin
1	PWM1EN	0	**MnPWM1** output enable 0: The **MnPWM1** pin has a zero value 1: The generated **pwm0B'** signal is passed to the **MnPWM1** pin
0	PWM0EN	0	**MnPWM0** output enable 0: The **MnPWM0** pin has a zero value 1: The generated **pwm0A'** signal is passed to the **MnPWM0** pin

The PWM-related control signals are alternate functions for some GPIO ports and default to be GPIO signals at reset. In Table 4.11, the first column lists the corresponding GPIO pins for these PWM signals. The **AFSEL** bit in the GPIO Alternate Function Select (GPIOAFSEL) register should be set to 1 to choose the PWM function. The number in parentheses is the **PMCx** encoding that must be programmed into the **PMCn** field in the **GPIO Port Control (GPIOPCTL)** register to assign the PWM signal to the specified GPIO port pin. Refer to Sects. 3.2.1.2.3, Table 3.2, and Fig. 3.7 in Chap. 3 to get more details about the **GPIOAFSEL** and **GPIOPCTL** registers and **PMCx** encoding related to the PWM control signals.

Table 4.11 GPIO pins' assignment for PWM MnPWMn output pins

GPIO pin (PMCx)	PWM pin	Pin type	Description
PF2 (4) PD6 (4) PD4 (4)	M0FAULT0	Input	**Module 0 PWM Fault 0**
PB6 (4)	M0PWM0	Output	**Module 0 PWM 0**. This signal is controlled by Module 0 PWM Generator 0
PB7 (4)	M0PWM1	Output	**Module 0 PWM 1**. This signal is controlled by Module 0 PWM Generator 0
PB4 (4)	M0PWM2	Output	**Module 0 PWM 2**. This signal is controlled by Module 0 PWM Generator 1
PB5 (4)	M0PWM3	Output	**Module 0 PWM 3**. This signal is controlled by Module 0 PWM Generator 1
PE4 (4)	M0PWM4	Output	**Module 0 PWM 4**. This signal is controlled by Module 0 PWM Generator 2
PE5 (4)	M0PWM5	Output	**Module 0 PWM 5**. This signal is controlled by Module 0 PWM Generator 2
PC4 (4) PD0 (4)	M0PWM6	Output	**Module 0 PWM 6**. This signal is controlled by Module 0 PWM Generator 3
PC5 (4) PD1 (4)	M0PWM7	Output	**Module 0 PWM 7.** This signal is controlled by Module 0 PWM Generator 3
PF4 (5)	M1FAULT0	Input	**Module 1 PWM Fault 0**
PD0 (5)	M1PWM0	Output	**Module 1 PWM 0.** This signal is controlled by Module 1 PWM Generator 0
PD1 (5)	M1PWM1	Output	**Module 1 PWM 1.** This signal is controlled by Module 1 PWM Generator 0
PA6 (5) PE4 (5)	M1PWM2	Output	**Module 1 PWM 2.** This signal is controlled by Module 1 PWM Generator 1
PA7 (5) PE5 (5)	M1PWM3	Output	**Module 1 PWM 3.** This signal is controlled by Module 1 PWM Generator 1
PF0 (5)	M1PWM4	Output	**Module 1 PWM 4.** This signal is controlled by Module 1 PWM Generator 2
PF1 (5)	**M1PWM5**	Output	**Module 1 PWM 5.** This signal is controlled by Module 1 PWM Generator 2
PF2 (5)	M1PWM6	Output	**Module 1 PWM 6.** This signal is controlled by Module 1 PWM Generator 3
PF3 (5)	**M1PWM7**	Output	**Module 1 PWM 7.** This signal is controlled by Module 1 PWM Generator 3

4.7.3.6 PWM Module Initialization and Configuration

Any PWM module must be initialized and configured before it can be implemented in any actual application. This initialization and configuration process can be divided into three parts as given below:

- Initialize and configure clock sources for PWM module and GPIO ports.
- Initialize and configure GPIO ports and pins related to PWM modules.
- Initialize and configure PWM module and generators.

Now let us start this initialization and configuration process with an example. In this example, the PWM module 1 and generator 2 is configured to

(1) Generate a 25% duty cycle on the **M1PWM4** pin.
(2) Generate a 75% duty cycle on the **M1PWM5** pin.
(3) Use a divided system frequency, which is 5 kHz, as the input to the counter.
(4) Use a 40 MHz system clock as the clock source.

Initialize and Configure Clock Source for PWM Module and GPIO Ports

Perform the following operations to fulfill this initialization and configuration:

1. Configure the **Run-Mode Clock Configuration Register (RCC)** register in the system control module:
   a. To use PLL and divisor 5 with 16 MHz XTAL to get 40 MHz system clock.
   b. To use the PWM divide (**USEPWMDIV**) by setting this bit (bit-20) and set the divider bits **PWMDIV** (bits 19:17) to divide by 2 (**000**) to get 20 MHz PWM clock.
2. Enable the PWM clock by writing a value of **0x0010.0000** to the **RCGC0** register in the system control module. Alternatively, use the **Pulse Width Modulator Run Mode Clock Gating Control (RCGCPWM)** register to enable PWM module 1 with clock.
3. Enable the clock to the appropriate GPIO module via the **RCGC2** register in the system control module. Alternatively, use the **General-Purpose Input/Output Run Mode Clock Gating Control (RCGCGPIO)** register to enable GPIO ports with clock.

Initialize and Configure GPIO Ports and Pins Related to PWM Modules

Perform the following operations to fulfill this initialization and configuration:

1. In the GPIO module, enable the appropriate pins for their alternate function using the **GPIOAFSEL** register. Refer to Sect. 3.2.1.2.3, Table 3.2, and Fig. 3.7 in Chap. 3 to get more details about the **GPIOAFSEL** and **GPIOPCTL** registers, and **PMCx** encoding related to the PWM control signals.
2. Configure the **PMCx** fields in the **GPIOPCTL** register to assign the PWM signals to the appropriate pins.
3. May need to use the **GPIOLOCK** and **GPIOCR** registers to make all modifications on **GPIOAFSEL** and **GPIOPCTL** registers above committed immediately.

Initialize and Configure PWM Module and Generators

Perform the following operations to fulfill this initialization and configuration:

1. Configure the PWM generator for count-down mode with immediate updates to the parameters.
   a. Write the **PWM1CTL** register with a value of **0x0000.0000** (see Sect. 4.7.3.3.1).
   b. Write the **PWM1GENA** register with a value of **0x0000.008C** (see Sect. 4.7.3.3.6).
   c. Write the **PWM1GENB** register with a value of **0x0000.080C** (see Sect. 4.7.3.3.7).
2. Set the period for the counter. For a 5-KHz input frequency, the period = 1/5000 or 200 μs. The PWM clock source is 20 MHz after the 20 MHz. Thus, there are 20 MHz/5 kHz = 4000 clock ticks per period.
   a. Use this value to set the **PWM1LOAD** register. In Count-down mode, set the **LOAD** field in the **PWM1LOAD** register to the requested period minus one (4000 − 1 = 3999).
3. Set the pulse width of the M1PWM4 pin for a 25% duty cycle. Write the **PWM1CMPA** register with a value of **3000**.
4. Set the pulse width of the M1PWM5 pin for a 75% duty cycle. Write the **PWM1CMPB** register with a value of **1000**.
5. Start the counters in PWM1 generator 2. Write the **PWM1CTL** register with a value of **0x0000.0001**.
6. Enable PWM outputs. Write the **PWMENABLE** register with a value of **0x0000.0003**.

In the above configuration operations, steps 1, 3, and 4 are easy to be confused. Let us have a closer look at these steps.

Refer to Sects. 4.7.3.3.6 and 4.7.3.3.7 for **PWM1GENA** and **PWM1GENB** registers. To generate a **pwmA** signal for the Comparator A down mode, the bit fields, **ACTCMPAD** (bits 7:6), **ACTLOAD** (bits 3:2), and **ACTZERO** (bits 1:0), are important and need to be configured. As shown in Fig. 4.38, in order to make the **pwmA** a Low when the counter matches the value in the **PWM1CMPA** register when it is counting down, the bit field **ACTCMPAD** (bits 7:6) should be set to **0x2** (10). In order to drive the **pwmA** a High when the counter matches the **LOAD** value in the **PWM1LOAD** register, the bit fields **ACTLOAD** (bits 3:2) should be set to **0x3** (11). For the counter equaling to 0, no action should be taken, and therefore the bit fields **ACTZERO** (bits 1:0) should be **0x0** (00). So the **PWM1GENA** register should be initialized to **0x008C**.

Similarly, for the **PWM1GENB** register, in order to make the **pwmB** output a Low when the counter matches the value in the **PWM1CMPB** register when it is counting down, the bit field **ACTCMPBD** (bits 11:10) should be set to **0x2** (10).

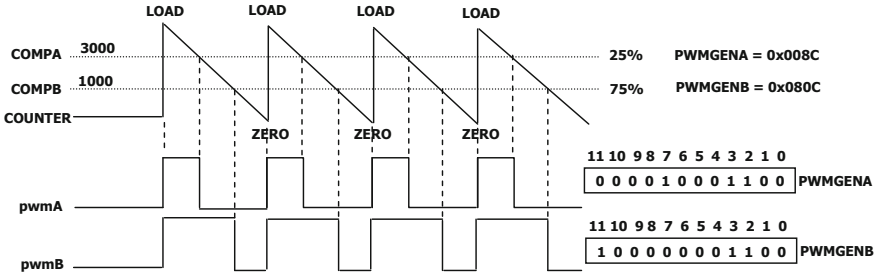


Fig. 4.38 The initialization and configuration process for PWM Module 0

In order to drive the **pwmB** a High when the counter matches the **LOAD** value in the **PWM1LOAD** register, the bit fields **ACTLOAD** (bits 3:2) should be set to **0x3** (11). For the counter equaling to 0, no action should be taken, and therefore the bit fields **ACTZERO** (bits 1:0) should be **0x0** (00). So the **PWM1GENB** register should be initialized to **0x080C**.

For the **PWM1CTL** register, since we do not use any fault source and condition, no debug with local synchronization mode and immediate update mode with disabling this PWM generator block, and therefore all bit fields are 0. The point to be noted is that in order to initialize and configure any PWM generator, the generator must be first disabled. No PWM generator can be configured when it is enabled.

For steps 3 and 4, a 25% duty cycle is calculated as 4000 * 0.75 = 3000. A 75% duty cycle is 4000 * 0.25 = 1000. Refer to Fig. 4.38, the higher the duty cycle value, the lower the setup value installed in the **PWM1CMPA** or **PWM1CMPB** register. This is just opposite to the common sense.

In step 5, after the PWM generator has been configured, the PWM1CTL is set by **0x01** to enable this generator. In step 6, the PWM master control register **PWMENABLE** register is configured with **0x03** to enable **pwm0A**' signal to output to the **M1PWM4** pin and **pwm0B**' signal to output to the **M1PWM5** pin, respectively.

Now we can begin to perform the identification process for our target DC motor.

4.7.4 Calibration of the Rotary Encoder to Identify the Feedback Path

In TM4C123GH6PM MCU system, all motors are driven by an integrated amplifier chip TB6612FNG via the PWM outputs from a PWM module in this microcontroller system. We have provided detailed discussions about the PMW module in the TM4C123GH6PM MCU system in the last section, so we are ready to go for our identification process.

The purpose of the calibration of the rotary encoder is to get a correct relationship or a mapping between the controller PWM outputs and the encoder outputs that work as feedback to the controller inputs. This step should be performed in next chapter for PID controller design, but in order to put everything related to the encoder together we do this step in this section.

In real implementations, this calibration process can be performed by building an implementation program to collect a sequence of encoder speeds in terms of a sequence of PWM output values. Ideally, the relationship between the encoder feedback speeds and PWM outputs should be a linear equation and it should be a straight line in a 2D plane, but in practice, it is a nonlinear function with a nonlinear trajectory in a 2-D coordinate system, as shown in Fig. 4.39. When the voltage applied on the motor is higher than certain value, the motor speed becomes saturated.

Our target DC motor, **Mitsumi 448 PPR** motor, has an optical rotary encoder including a 448 lines/pulses speed disk. If this 448-line/pulse per revolution rotary encoder is used in a closed-loop control system, a total of 1792 (**448**×**4**) pulses per revolution (PPR) can be obtained if both rising and falling edges for Phase A and Phase B inputs are utilized. To calibrate this encoder, a user program can be built to continuously send the PWM outputs to the motor starting from 5 to 100%. Then, the user program can collect all related encoder speeds on those PWM output values and store them in a data array.

Then a data fitting method, such as the least square method, can be used to get the actual equation for this relationship (**ES = *k* × PWM + *b***), where ***b*** is the intersection across the vertical axis and ***k*** is the slope of this linear equation. The estimated linear function should be a straight line similar to the linear range shown in Fig. 4.39.

In this calibration project, we use the following components to perform this calibration:

- The QEI0 with its **PhA0 (PD6)** and **PhB0 (PD7)** as two external phase input signals.

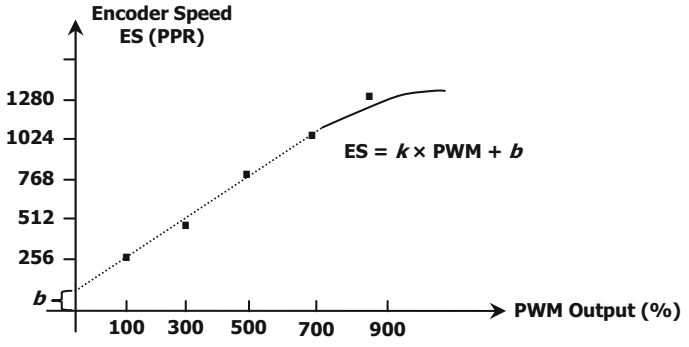


Fig. 4.39 Relationship between the encoder speed (ES) and the PWM

- The **PWM1_2B** or **M1PWM5** pin as PWM output signal to control a DC motor.
- The DC motor is a **Mitsumi 448 PPR motor** with an optical rotary encoder including a 448 lines/pulses speed disk.

Before the encoder calibration process can be performed, this DC motor and its encoder must be connected to the TM4C123GXL EVB via GPIO Ports B, D, and F to enable this calibration process to be started. The hardware configuration and connection for this DC motor are shown in Fig. 4.40.

As shown in Fig. 4.40, all components and connections inside the dash-line box have been done by the TM4C123GXL EVB and the EduBASE ARM® Trainer, and the users do not need to touch these connections. However, the users need to do those connections outside of this dash-line box. These connections include the following:

1. DC motor connections via Motor 1 (Yellow wire) and Motor 2 (Blue wire) on the Mitsumi 448 PPR DC motor to the **T1** connector in the EduBASE ARM® Trainer.
2. Motor external 12 V DC power connection via connectors 1 and 2 on the **T3** connector in the EduBASE ARM® Trainer to provide the external power supply to the motor.
3. Jumper connection by connecting pins 1 and 2 on the J4 connector in the EduBASE ARM® Trainer to enable the motor to use the external power supply.
4. Two encoder phases connections via PhA (White wire) and PhB (Green wire) phase outputs in the Mitsumi 448 PPR DC motor to **PD6** and **PD7** pins on the **J4** connector in the TM4C123GXL EVB since we used **PhA0** and **PhB0** as two-phase inputs for this project. Refer to Table 4.5 to get the related GPIO pins corresponding to these phases.
5. Encoder power supply connections via Vcc (Red wire) and GND (Black wire) on the Mitsumi 448 PPR DC motor to 5 V and GND pins on the **J3** connector in the TM4C123GXL EVB.

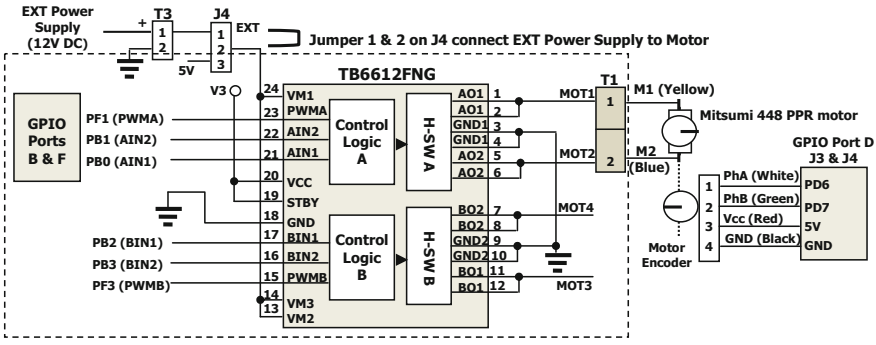


Fig. 4.40 The hardware configuration for the encoder calibration process

The finished hardware connection should be matched one that is shown in Fig. 4.40. Figure 4.41 shows the block diagram of this user program.

Before we can start building this user program to calibrate the encoder, we should have a clear picture about the relationships between the motor rotating speed, the encoder output phases, and the maximum pulse load value for the QEI velocity timer.

The Mitsumi 448 PPR DC motor has an optical rotary encoder including a 448 lines/pulses speed disk, this means that 448 pulses for both PhA and PhB are output for each revolution of the motor rotation. The relationship between the motor rotating speed (revolution per minute—**rpm**), the system clock frequency (**clock**), the output pulses per revolution (**ppr**), and the maximum load pulse value for the QEI velocity timer (**LOAD**) are

$$\textbf{rpm} = (\textbf{clock} \times \textbf{(2 ^ VELDIV)} \times \textbf{SPEED} \times \textbf{60}) \div (\textbf{LOAD} \times \textbf{ppr} \times \textbf{edges}),$$

where the parameter **edges** are 2 or 4, based on the capture mode bit in the QEI Control (**QEICTL**) register (2 for **CAPMODE** clear and 4 for **CAPMODE** set). The **VELDIV** is the bit field in the **QEICTL** register to determine the velocity pre-divider's value.

In this project, we select and set up the following parameters for this encoder calibration:

1. The rotating speed of the Mitsumi 448 PPR DC motor is 8700 revolution per minute. This result in **rpm** = 8700.
2. System clock is 6.67 MHz when the 400 MHz PLL clock source is used and divided by a system divider of 30 (**clock** = 6.67 MHz).
3. The QEI module velocity pre-divider is set to ÷1 (**VELDIV** bit is clear).
4. Clock and count on both PhA and PhB edges by setting **CAPMODE** bit to 1 in the **QEICTL** register to get **edges** = 4.

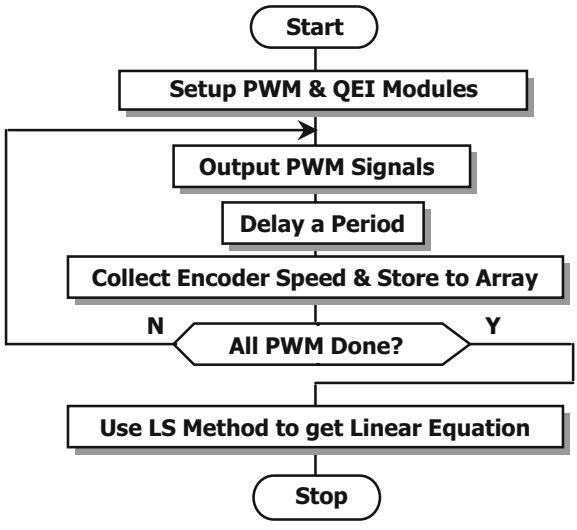


Fig. 4.41 The block diagram of the encoder calibration program

5. Since the motor running at 8700 rpm and each revolution creates 448 pulses in both PhA and PhB. Thus, a total of 1792 pulses can be generated for each revolution since the **edges** have been set to 4 in step 4 above ($4 \times 448 = 1792$).
6. This result in 259,840 pulses per second ($(1792 \times 8700)/60$) since the motor rotates 8700 revolution per minute or 145 revolution per second, or 0.26 pulses per μs.
7. Since the clock frequency is 6.67 MHz with a period of 1.5×10^{-7} s, if we select the **LOAD** value as 2500, it would count 98 pulses per update ($1.5 \times 10^{-7} \times 2500 = 375\ \mu s \times 0.26 = 98$).

Using above equation, we have

$$\mathbf{rpm = (6.67 \times 106 \times 1 \times 98 \times 60) \div (2500 \times 448 \times 4) = 8754}$$

```
1  //*****************************************************************************
2  // CalibEncoder.c - User Program used to Calibrate Rotary Optical Encoder
3  //*****************************************************************************
4  #include <stdint.h>
5  #include <stdbool.h>
6  #include "driverlib/sysctl.h"
7  #include "driverlib/gpio.h"
8  #include "driverlib/qei.h"
9  #include "TM4C123GH6PM.h"
10 #define GPIO_PD6_PHA0          0x00031806
11 #define GPIO_PD7_PHB0          0x00031C06
12 #define GPIO_PORTD_BASE        0x40007000
13 #define GPIO_PORTF_CR_R        (*((volatile uint32_t *)0x40025524))
14 #define GPIO_PORTD_CR_R        (*((volatile uint32_t *)0x40007524))
15 void InitPWM(void);
16 void InitQEI(void);
17 int main(void)
18 {
19   uint32_t pw, index = 0, esData[20];
20   // Set the clocking to run directly from the crystal with 6.67 MHz clock frrequency.
21   SysCtlClockSet(SYSCTL_SYSDIV_30|SYSCTL_USE_PLL|SYSCTL_XTAL_4MHZ|SYSCTL_OSC_MAIN);
22   SYSCTL->RCGC2 = 0x2A;                    // enable & clock GPIO Ports B, D & F
23   GPIOB->DEN = 0xF;                        // enable PB3 ~ PB0 as digital function pins
24   GPIOB->DIR = 0xF;                        // configure PB3 ~ PB0 as output pins
25   GPIOD->DEN = 0xC0;                       // enable PD7 ~ PD6 as digital function pins
26   GPIOD->DIR = ~0xC;                       // configure PD7 ~ PD6 as input pins
27   GPIOF->DEN = 0xF;                        // enable PF3 ~ PF0 as digital function pins
28   GPIOF->DIR = 0xF;                        // configure PF3 ~ PF0 as output pins
29   InitPWM();                               // configure PWM module 1
30   InitQEI();                               // configure QEI module 0
31   PWM1->_2_CTL = 0x1;                      // enable PWM1_2B or M1PWM5
32   PWM1->ENABLE = 0x20;
33   for (pw = 100; pw < 3999; pw += 200)     // send PWM signal to motor from 2.5% to 100%
34   {
35     PWM1->_2_CMPB = pw;                    // output the PWM values to the motor
36     SysCtlDelay(1000);
37     esData[index] = QEI0->SPEED;           // collect the encoder speed values
38     index++;
39   }
40   PWM1->_2_CTL = 0x0;                      // disable PWM1_2B or M1PWM5
41   PWM1->ENABLE = ~0x20;
42   while(1);                                // check and get the encoder speed values array
43 }
```

Fig. 4.42 The first part of the codes for the project CalibEncoder.c

As this program is developed and the rotary encoder has been calibrated, the real PWM value matched to any desired encoder speed can be estimated based on the linear equation. In the real control program, this estimated PWM value can be considered as a nominal PWM output value used to control the motor, and it can be adjusted (either increasing or decreasing) based on the actual feedback encoder speed by comparing the PWM value and the encoder feedback speed value in a loop in the control program.

Now let us build our calibration program **CalibEncoder**.

Create a new project **CalibEncoder** in the folder **C:\C-M Control Class Projects\Chapter 4**. The detailed codes for this user program are shown in Figs. 4.42 and 4.43.

Figure 4.42 shows the first part of this calibration encoder codes and Fig. 4.43 shows the second part of these codes. Let us have a closer look at these codes to see how they work.

(1) The codes in lines 4～9 are used to include some system header files to be implemented in this project.
(2) The codes in lines 10～4 are used to define some system constants and macros used in this project.

```
44 void InitPWM(void)
45 {
46   SYSCTL->RCGCPWM |= 0x2;                    // enable PWM1 with clock
47   while((SYSCTL->PRPWM & 0x2) == 0) {};      // wait for PWM1 ready
48   SYSCTL->RCC |= 0x00100000|0x00000000;      // 20 MHz clock
49   GPIOF->LOCK = 0x4C4F434B;                  // unlock GPTOF commit register
50   GPIO_PORTF_CR_R |= 0x2;                    // enable PF1 to be committed
51   GPIOF->LOCK = 0x0;                         // lock GPIOF commit register
52   // PWM1_2B - PF1 - M1PWM5 pin - Module 1 Generator 2 - pwm2B
53   PWM1->_2_CTL = 0x0;                        // disable PWM1_2B or M1PWM5
54   PWM1->_2_GENB = 0x0000080C;                // high on LOAD, low on CMPB down
55   PWM1->_2_LOAD = 3999;                      // load = 4000 - 1
56   GPIOF->AFSEL |= 0x2;                       // PF1 - Alternate Function: PWM1-2B
57   GPIOF->PCTL = 0x00000050;                  // M1PWM5 on PF1 = 0x00000050
58   GPIOF->AMSEL &= ~0x02;                     // disable analog function on PF1
59   GPIOB->DATA |= 0x1;                        // enable PB0 for motor CW-rotation
60 }
61 void InitQEI(void)
62 {
63   SYSCTL->RCGCQEI = 0x1;                     // enable and clock QEI0 module
64   GPIOD->LOCK = 0x4C4F434B;                  // unlock GPTOD commit register
65   GPIO_PORTD_CR_R |= 0x80;                   // enable PD7 to be committed
66   GPIOD->LOCK = 0x0;                         // lock GPIOD commit register
67   GPIOPinConfigure(GPIO_PD6_PHA0);           // set PD6 as PhA0 pin
68   GPIOPinConfigure(GPIO_PD7_PHB0);           // set PD7 as PhB0 pin
69   GPIOPinTypeQEI(GPIO_PORTD_BASE, GPIO_PIN_6);
70   GPIOPinTypeQEI(GPIO_PORTD_BASE, GPIO_PIN_7);
71   QEI0->CTL = 0x0;                           // disable QEI0 to enable configuration
72   QEI0->CTL = 0x08;                          // enable QEI0 quadrature A & B mode
73   QEI0->LOAD = 2499;                         // set the max number of clock cycles
74   QEIVelocityEnable(QEI0_BASE);              // enable the QEI0 velocity mode
75   QEIEnable(QEI0_BASE);                      // enable the quadrature encoder.
76 }
```

Fig. 4.43 The second part of the codes for the project CalibEncoder.c

(3) Two user-defined functions, **InitPWM()** and **InitQEI()**, are declared in lines 15 ~ 6 and used to initialize the PWM and QEI modules.
(4) The main program starts at line 17. Some **uint32_t** variables and data array are declared first in the main program and they are used as the PWM output variable **pw**, loop counter **index,** and the data array **esData[]** to save the collected encoder speed values in line 19.
(5) In line 21, the system clock used for this QEI module and this project is configured as 6.67 MHzclock (PLL = 400 MHz/2 = 200 MHz divided by **SYSDIV_30** = 6.67 MHz).
(6) The codes in lines 22 ~ 28 are used to enable and clock the PWM module 1 and GPIO Ports B, D, and F.
(7) Two initialization functions, **InitPWM()** and **InitQEI()**, are called in lines 29 and 30 to configure the PWM1 and QEI0 modules.
(8) In lines 31 and 32, the **PWM1_2B** output and the PWM module 1 is enabled to make it ready to drive the DC motor.
(9) The codes in lines 33 ~ 39 provide a fixed number loop operation to continuously send PWM outputs to the target DC motor from 5 to 100% to rotate the motor with different speeds. After each PWM signal is sent out, the **SysCtlDelay()** function is called to delay the program a period of time to make the motor rotation speed stable. Then the encoder speed is collected by reading the **QEISPEED** register and assigning them to the data array **esData[]** to store them into this array.
(10) After all encoder speed values are collected, the **PWM1_2B** and the PWM module 1 are disabled in lines 40 and 41.
(11) An infinitive **while(1)** loop in line 42 provides a temporary stop state to enable users to check the encoder speed values stored in the **esData[]** array.

Now let us have a closer look at the second part of the codes for this encoder calibration project, which is shown in Fig. 4.43.

(12) The detailed codes for the function **InitPWM()** start at line 44.
(13) The PWM module 1 is clocked and enabled in line 46.
(14) A **while()** loop is used to wait for this clock and enables process to be completed and the PWM module 1 to be stable in line 47.
(15) In line 48, the register **RCC** is configured to enable to use the **PWMDIV** parameter to make the PWM module driven clock to be 20 MHz.
(16) In lines 49 and 50, the lock register for the Port F is unlocked to enable the **PF1** pin to be committed by setting bit 1 in the **GPIOCR** register to 1. Then, in line 51, this register is locked again to prevent it from modifying again in the future.
(17) The codes in lines 52 ~ 55 configure the related registers in the PWM1 module. First, the **PWMCTL** register is disabled to start this configuration process in line 53.
(18) Then the **PWMGENB** register is configured to output High when loading and output Low when performing compare-down operations in line 54.

(19) The **PWMLOAD** register is loaded with the upper bound value of **3999** in line 55.
(20) The codes in lines 56 ~ 60 are used to configure **PF1** as an alternate function pin, exactly as a PWM output signal pin for **PWM1_2B** generator.
(21) The code in lines 59 is used to enable **PB0** pin as an output and digital function pin to control the motor rotation direction.
(22) The **InitQEI()** function starts at line 61.
(23) The QEI module 0 is enabled and clocked by configuring the **RCGCQEI** register in line 63.
(24) In lines 64 and 65, the lock register for the Port D is unlocked to enable the **PD7** pin to be committed by setting bit 1 in the **GPIOCR** register to 1. Then, in line 66, this register is locked again to prevent it from modifying again in the future.
(25) In lines 67 and 70, the GPIO Port D is configured to make **PD7** and **PD6** as alternate function pins, exactly as two encoder phase input pins, **PhB0** and **PhA0**.
(26) In line 71, the **QEICTL** register is disabled to enable the configuration process to be started.
(27) In line 72, the **QEI** module 0 is configured to enable QEI0 quadrature A and B mode.
(28) The **LOAD** register is configured in line 73 by loading the maximum number of clock cycles (2500–1) into the velocity timer register as the upper bound of the timer.
(29) After these configurations have been done, the **QEI0** velocity capture mode is enabled in line 74 by calling the API function **QEIVelocityEnable()**.
(30) The QEI module 0 is enabled in line 75 by calling the API function **QEIEnable()**.

Before one can run this project to calibrate the motor encoder, the following environments need to be set:

- Add the path for all system header files used in this project into the **Include Paths** box in the **C/C++** tab under the **Project|Options for Target 'Target 1'** menu item. The correct path should be **C:\ti\TivaWare_C_Series-2.1.4.178**.
- Select the correct debug driver **Stellaris ICDI** in the debug tab under the **Project|Options for Target 'Target 1'** menu item.
- Add the TivaWare™ peripheral driver library **driverlib.lib** file into the project. This library file is located at: **C:\ti\TivaWare_C_Series-2.1.4.178\driverlib\rvmdk**. By right-clicking on the **Source Group 1** in the **Project** pane and selecting the **Add Existing File**… item, one can add this library file into the project.

Now run the project to collect the motor encoder speed values based on the given PWM values. The collected encoder speed values should be stored in the **esData[]** array. After the project runs, go to the **Debug|Stop** menu item to stop

the project. Then, open the **Call Stack + Locals** window, one can expand the **esData[]** array and record these speed values to be used later.

Now that the encoder has been calibrated, we are ready to identify the dynamic model for our target DC motor, **Mitsumi 448 PPR motor**, use MATLAB system identification tool.

4.7.5 Identify the Dynamic Model for Our Target DC Motor

Since we have collected the encoder speed values for given PWM outputs in the last section, we can go ahead to identify the dynamic model of our DC motor, which includes the motor driver and motor itself.

In order to make our identification process simple and successful, we cannot use all of collected encoder speed output values. Instead, we can select the collected encoder speed values that are in the linear range of the output–input relationship shown in Fig. 4.39. In other words, we should not use the encoder speed values close to the starting point and the point from which the motor speed enters the saturation region. For example, if we collect 10 or 20 encoder speed values based on 10 or 20 PWM outputs, we can use the collected data between 3 and 10 collected encoder speed values or between 5 and 15 collected encoder speed values to this identification process.

As we mentioned, the MATLAB® system identification tool is a good tool to identify the dynamic model for this DC motor. In order to do this identification job, the following operational sequence is needed:

1. Collect and format the motor input and output data, or the PWM output values that can be considered as the input to the motor and the encoder speed values that can be considered as the output of the motor, to make them as two data arrays to be in a data file.

   This step needs to build a MATLAB® script file to complete this format.

2. In the MATLAB® workspace, load this data file into the MATLAB® workspace.
3. Open the MATLAB® system identification tool to begin the identification process.

Now let us begin this identification process and readers need to have MATLAB® and Identification Toolbox™ to fulfill this process.

4.7.5.1 Format the Input and Output Data for the DC Motor

In the last section, we built an encoder calibration program **CalibEncoder** to collect a group of encoder speed values based on a sequence of PWM values

outputs to the DC motor, and save the encoder speed values into a data array **esData[]**. The collected encoder speed values and the corresponding PWM values are shown in Table 4.12.

In order to get the optimal and satisfied identification results, as we mentioned, we cannot use all of these collected data. Instead, we need to select some of these data as shown in Table 4.12. The reason for that is because the data items at the beginning and the data items close to the ending part belong to very nonlinear data and they are not appropriate to be used to estimate the optimal and true motor plant model. In order to avoid this incorrect identification result based on those nonlinear data, we need to do this kind of selection. The data we have chosen are eight pieces of them starting from data number 3 and ending at data number 10.

In this section, we need to format those PWM and encoder speed values to form another data array to enable the MATLAB® Identification Toolbox™ to use that array to perform identification process.

We need to build a MATLAB® script file **getMData.m** to do this data array format.

Open MATLAB® and create a new MATLAB® script file by going to **File|New|Script** menu item to open a new script file. Enter the codes shown in Fig. 4.44 into this script file.

Let us have a closer look at this piece of codes to see how it works.

(1) The initial PWM value **500** is assigned to an accumulated variable **S** in line 2.
(2) An 8 × 1 array **u** is declared in line 3 and initialized with all zeros. This array is a group of PWM outputs to the motor and it can be considered as the input to the DC motor.

Table 4.12 The collected QEI encoder speed values and the PWM values

Number	PWM values (u)	Encoder velocity values (es) PPR	
		Hexadecimal	Decimal
1	100	49	73
2	300	F4	244
3	500	15C	348
4	700	19F	415
5	900	1E0	480
6	1100	1EF	495
7	1300	1FA	506
8	1500	201	513
9	1700	206	518
10	1900	20C	524
12	2100	20D	525
13	2300	20C	524
14	2500	20E	526
15	2700	20E	526

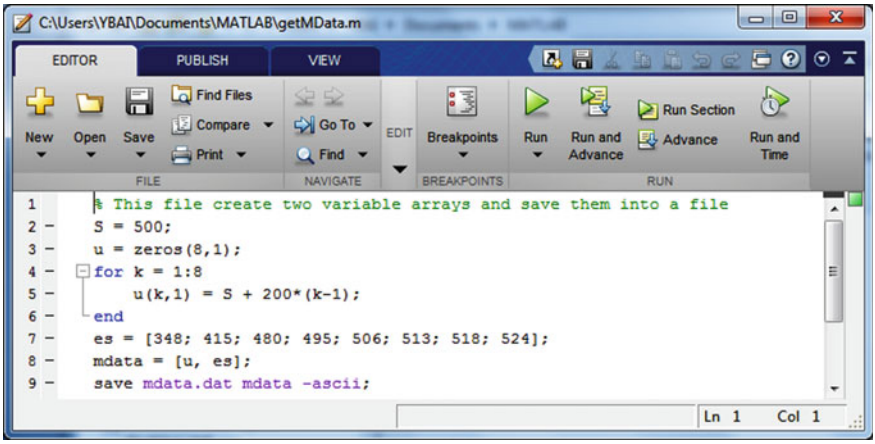


Fig. 4.44 The MATLAB script file getMData.m

(3) A **for** loop is used to fill the array **u** with a sequence of PWM values starting from **500** to **1900** in lines 4∼6.
(4) The encoder speed values collected in the **CalibEncoder** project are assigned to the data array **es** in line 7. One has to convert the collected encoder speed values in the array **esData[]** from the hexadecimal to the decimal and put them in this new data array **es**. This array can be considered as the output of the DC motor.
(5) A new data array **mdata** used for identification purpose is created in line 8. This array includes two 8 × 1 vectors or one 8 × 2 matrix.
(6) This new formatted data array **mdata** is saved to a MATLAB® ASCII file named **mdata.dat** in line 9.
(7) Save this script as **getMData** and run this script file by clicking on the **Run** arrow on the toolbar to create this new data array and data file **mdata.dat**.

A new data array file **mdata.dat** is created and it is located at the MATLAB® default user folder **C:\User\User_Name\My Documents\MATLAB**.

Next, let us load this data file into the MATLAB® workspace to enable the Identification Toolbox™ to recognize it and use it. In the opened MATLAB® Command window with the ***fx*>>** cursor ready, type

load mdata.dat;

and press the **Enter** key on your keyboard. Then, type **mdata** and press the **Enter** key on your keyboard again. All data items in this array are displayed in this window, as shown in Fig. 4.45. If the Workspace window is opened, you can find that two arrays, **u** and **es**, as well as the data array mdata have been added into the workspace located at the upper-right corner. You can open this Workspace window by going to **Desktop** menu and checking the **Workspace** item.

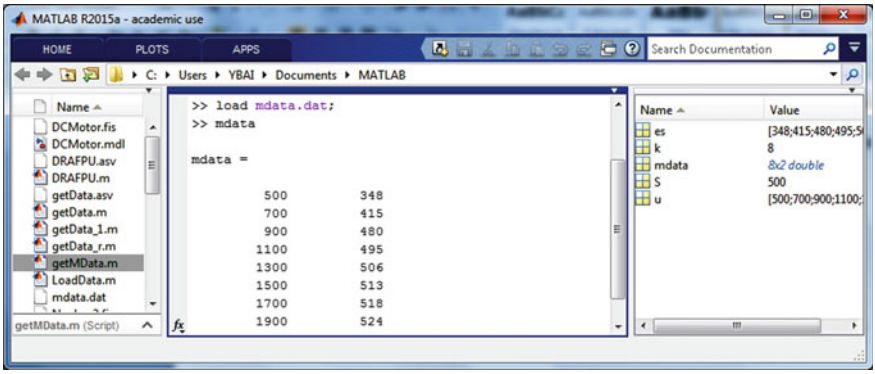


Fig. 4.45 Load the data array mdata.dat into the MATLAB workspace

4.7.5.2 Identify the DC Motor Dynamic Model with Identification Toolbox™

Now open the MATLAB® system identification tool by typing **ident** in the MATLAB® Command window after the ***fx>>*** cursor and press the **Enter** key.

The opened Identification Toolbox is shown in Fig. 4.46a.

Click on the drop-down arrow on the **Import data** combo box and select the **Time-domain data** item from this box to open the **Import data** wizard, as shown in Fig. 4.46b.

Now type the following items into the **Input, Output, Data name, Starting time,** and **Sampling interval** textboxes:

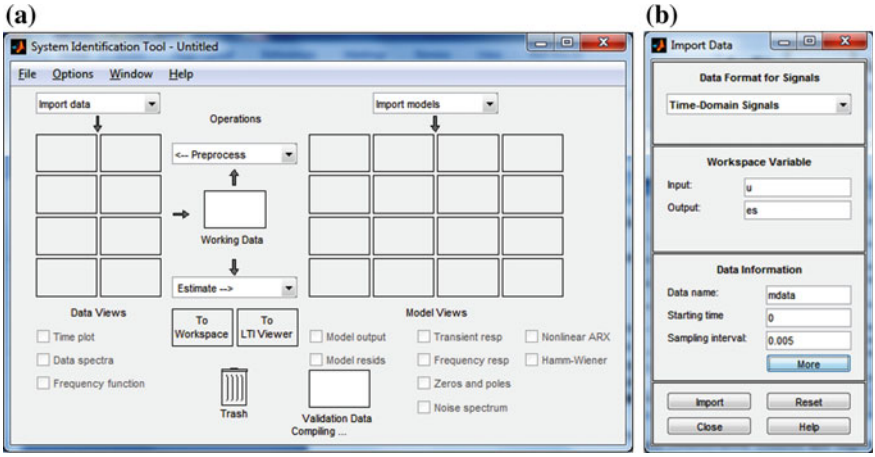


Fig. 4.46 The opened Identification Toolbox and import data wizard. Reprinted with the permission of MathWorks, Inc.

- Input: **u,**
- Output: **es,**
- Data name: **mdata,**
- Starting time: **0, and**
- Sampling interval: **0.005.**

Then, click on the **import** button to import these items into the Identification Toolbox.

Now, in the opened Identification Toolbox, perform the following operations to begin this identification process:

1. Click on the drop-down arrow in the **Preprocess** combo box and select **Remove trends** item to remove any possible trend for this data array. The modified data array **mdatad** with removed trends is shown in the second graphic cell, as shown in Fig. 4.47.
2. Click on the drop-down arrow in the **Estimate** combo box and select **Process Models** item to identify the desired model.
3. The Process Model wizard appears, as shown in Fig. 4.48. Keep all default setups and click on the **Estimate** button.
4. The identified dynamic model for this DC motor is complete and the identified result is shown in Fig. 4.48. The values for the coefficient K and the pole are listed in the **K** and **Tp1 Value** boxes, respectively. The dynamic model is

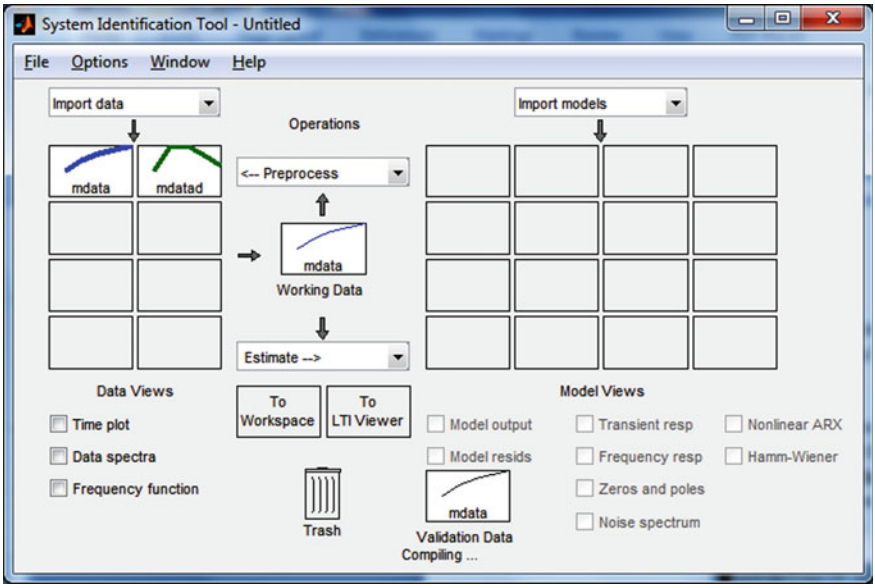


Fig. 4.47 The modified data array mdatad in the Identification Toolbox. Reprinted with the permission of MathWorks, Inc.

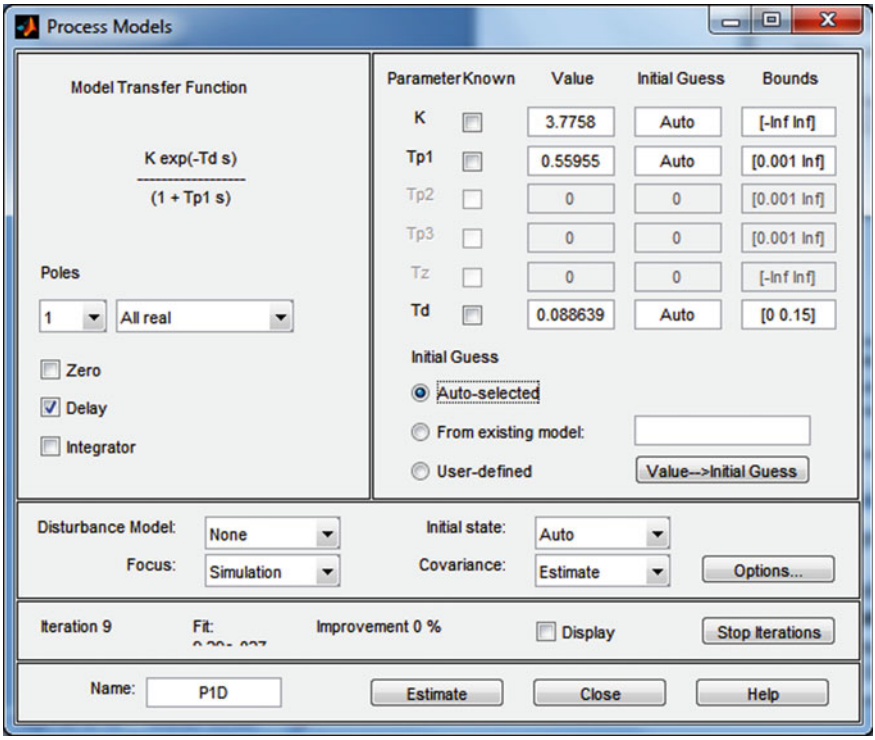


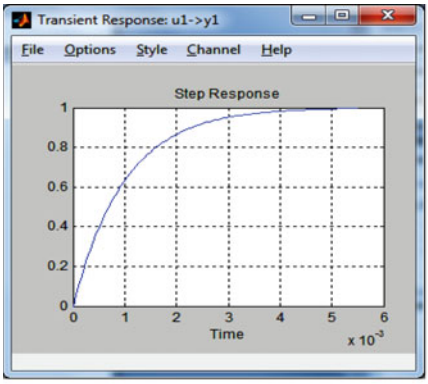
Fig. 4.48 The opened process models wizard. Reprinted with the permission of MathWorks, Inc.

$$G(s) = \frac{3.776}{(1+0.56s)} e^{-0.09s} \cong \frac{6.74}{(s+1.79)}$$

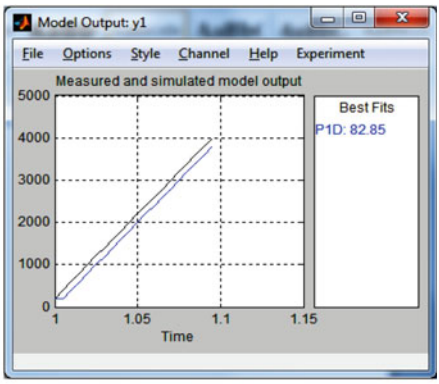
5. Now, return to the System Identification Toolbox, and drag the identified model **P1D** and place it into the **LTI Viewer** box. The simulated step response of this model is pop up, as shown in Fig. 4.49a. The **LTI viewer** (**L**inear **T**ime-**I**nvariant Viewer) allows users to view the simulated results for the identified model, such as transient response, model output, model residuals, frequency response, zeros and poles as well as noise spectrum.

Check the following checkboxes in the System Identification Toolbox™ window to open the related plotting wizards to check the identification result:

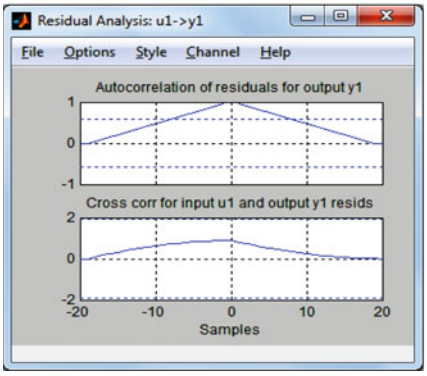
- **Model output**
- **Model resids**
- **Transient resp**



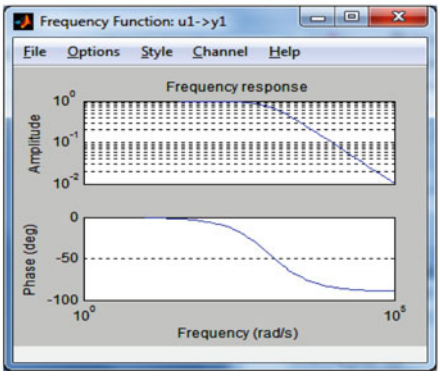
(a) Transient or step response.



(b) Identified model output.



(c) Model residual analysis.



(d) Model frequency response.

Fig. 4.49 The identified model responses and analysis. Reprinted with the permission of MathWorks, Inc.

- **Frequency resp**

The related identification results are shown in Fig. 4.49b ∼ d.

From these analyses and responses, one can find that our identified model is very close to the actual motor system and it is good for us to continue to build a PID control system to control this DC motor to get the optimal control performances. We will discuss this PID controller design topic in the next chapter.

4.8 Summary

To build a successful closed-loop control system to get the desired objectives, an accurate and correct mathematical model for the control plant is necessary and critically important. Most popular mathematical models for the most often used systems in our world are discussed and introduced in this chapter. Different methods are introduced to set up, analyze, and identify the system models to enable users to successfully design desired control systems based on those models.

Starting Sect. 4.2, various mathematical models for most popular physical systems are introduced and analyzed, which include the mechanical rotational system, electrical circuit system, and electromechanical system. Following this introduction, the relationship between the step response and the poles of a system is analyzed, and a discussion about the time-domain specifications for step responses is given.

Some important properties related to system models, such as final value theorem, steady-state errors, and stability and linear approximation of physical systems, are discussed in Sect. 4.4. Some popular and powerful MATLAB functions provided by the System Control Toolbox are introduced and discussed in detail since these functions are helpful and useful for the system modeling and conversion operations.

The discussion about system identifications is provided in Sect. 4.5. Some popular and important identification methods provided by MATLAB® System identification Toolbox are discussed in Sect. 4.6.

A case study is provided in Sect. 4.7 to show users how to practically perform the identification process for a typical DC motor using the MATLAB® System Identification Toolbox™. The QEI and PWM modules in TM4C123GH6PM MCU system are also introduced in this section since we need to use these modules to interface the motor amplifiers and motor optical encoders to build and collect the real data required by the identification process.

Homework and Labs

4.1 Develop the differential equations for the following electrical circuits in Fig. 4.50:

(a) The lead RC circuit.
(b) The lag RC circuit.

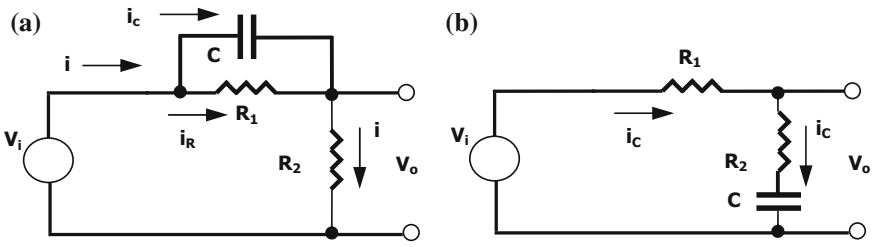


Fig. 4.50 Lead and Lag RC circuits

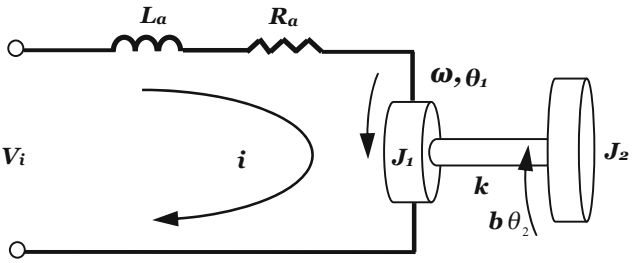


Fig. 4.51 A typical electromechanical position control system

4.2 Figure 4.51 shows a typical electromechanical position control system that is an electrical motor driving a load. The motor has electrical constant K_e, toque constant K_t, armature inductance L_a, and resistance R_a. The rotor has inertia J_1 and viscous friction B. The load has inertia J_2. The two inertias are connected by a shaft that has spring constant k and equivalent viscous damping b. Write the differential equations of motion for this motor system.

4.3 A second-order system is given below:

$$G(s) = \frac{3}{s^2 - 2s - 3}$$

(a) Find the DC gain for this system.
(b) What is the final value of the step response of this system?

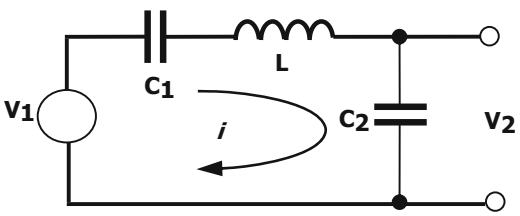
4.4 Convert the differential equations in question 4.1 to transfer functions.

4.5 A given control system is described by a transfer function as

$$G(s) = \frac{Y(s)}{R(s)} = \frac{500(s+100)}{s^2 + 60s + 500}$$

(a) If $r(t)$ is a unit step input, find the output $y(t)$.
(b) What is the final value of $y(t)$?

Fig. 4.52 A filter circuit



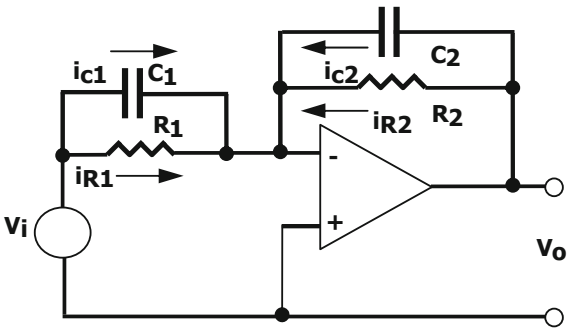


Fig. 4.53 An ideal operational amplifier circuit

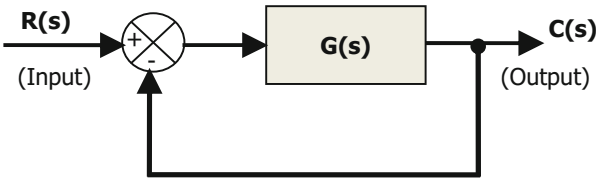


Fig. 4.54 A unit negative feedback control system

4.6 A filter circuit to filter out the high-frequency signals is shown in Fig. 4.52. Derive the transfer function for the output $V_2(s)/V_1(s)$.

4.7 For a given operational amplifier circuit shown in Fig. 4.53, find the transfer function $V_o(s)/V_i(s)$. Assume that this is an ideal operational amplifier.

4.8 A system is shown in Fig.4.54.

(a) Find the closed-loop transfer function $C(s)/R(s)$, when $G(s) = 24/(s^2 + 30s + 176)$.
(b) Determine the output $C(s)$ when the input $r(t)$ is a unit step.
(c) Find the $c(t)$ and plot it with MATLAB script.

4.9 A closed-loop feedback control system is shown in Fig. 4.55a.

(a) Determine transfer function $C(s)/R(s)$ for Fig. 4.55a.
(b) Determine equivalent $G(s)$ and $H(s)$ for block diagram shown in Fig. 4.55b that are equivalent to those of the block diagram of Fig. 4.55a.

4.10 The transfer function for a given control system is

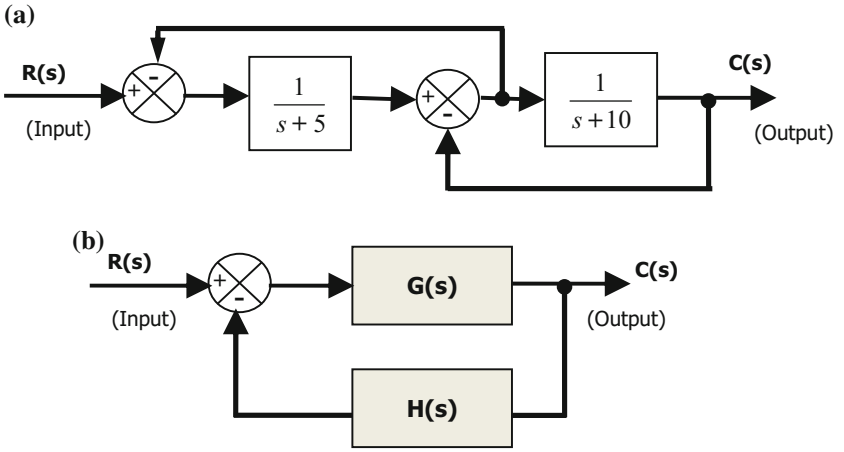


Fig. 4.55 Closed-loop control systems

$$\frac{C(s)}{R(s)} = \frac{10(s+2)}{s^2+8s+15}$$

Find the $c(t)$ when the input $r(t)$ is a unit step input and plot the output c(t) with MATLAB.

4.11 A spring used in an auto shock absorber develops a force, f, represented by the relation $f = kx^3$, where x is the displacement of the spring. Approximate this nonlinear system to a linear system around the operating point at $x_0 = 1$.

4.12 A nonlinear system has the relationship between the input and output as given below:

$$y = x + 0.4x^3$$

(a) Find the values of the output for steady-state operation at two points $x_0 = 1$ and $x_0 = 2$.
(b) Get a linearized model for both points and compare them.

4.13 Use MATLAB function residue() to decompose the transfer function in 4.10, and find the output $c(t)$ when the input $r(t)$ is a unit step input and plot the output $c(t)$ with MATLAB.

4.14 Use MATLAB functions to redo 4.8.

4.15 Determine the residues for the partial fraction expansion for G(s) (a) using MATALB function `residue()` and (b) plot step response for this system.

$$G(s) = \frac{540}{s^2+8s+540}$$

4.16 A system is represented by a transfer function $G(s)$. (a) find the partial fraction expansion and (b) plot step response of this system (any method can be used).

$$G(s) = \frac{8}{s^3 + 8s^2 + 17s + 10}$$

4.17 A closed-loop control system is shown in Fig. 4.56.

(a) Find the closed-loop transfer function $T(s) = C(s)/R(s)$.
(b) Determine poles and zeros of the $T(s)$.
(c) Use a unit step input, $r(t) = 1$ or $R(s) = 1/s$, and obtain the partial fraction expansion of $C(s)$.
(d) Find the step response of $T(s)$ and plot it.

4.18 An input $r(t) = u(t)$, a step input as $t \geq 0$, is applied to a black box with a transfer function $G(s)$. The output is $c(t) = 2–3e^{-t} + e^{-2t}\cos(2t)$ when $t \geq 0$. Find the $G(s)$ for this system.

4.19 A closed-loop control system is shown in Fig. 4.57.

(a) Get the closed-loop transfer function using `feedback()` and `series()` functions.
(b) Get the step response of this closed-loop system and verify that the final value of the output is −0.25.

4.20 Samples from a step response are given in Table 4.13.

(a) Plot this data set with MATLAB `plot()` function.
(b) Using MATLAB System Identification Toolbox function `process()` to identify the system model.

4.21 The step response $y(t)$ of a linear system $G(s)$ to a step input is shown in Fig. 4.58. Note that the peak output is 7 and the stead y-state value is $y_\infty = 6$. It is given that the 2% settling time is $t_{s,2\%} = 14$ s. Find the damping coefficient ξ and the natural frequency ω_n of $G(s)$ explaining every step. Then, find $G(s)$, its pole locations and its peak time t_p.

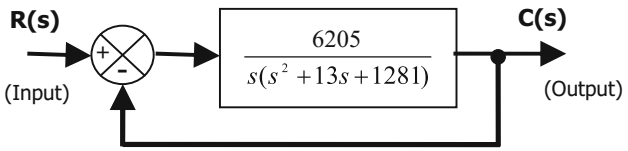


Fig. 4.56 A closed-loop control system

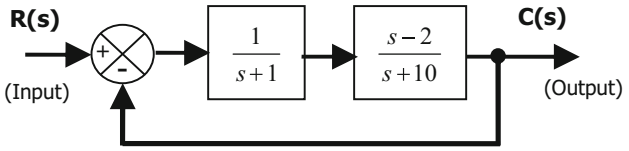


Fig. 4.57 A negative feedback control system

Table 4.13 The collected data for a step response

t	y	t	y
0	0	0.28	0.668
0.02	0.001	0.3	0.771
0.04	0.005	0.32	1.979
0.06	0.014	0.34	2.624
0.08	0.031	0.36	3.253
0.1	0.057	0.38	3.851
0.12	0.091	0.4	4.409
0.14	0.135	0.42	4.924
0.16	0.187	0.44	6.904
0.18	0.248	0.46	8.121
0.2	0.138	0.48	8.86
0.22	0.395	0.5	9.309
0.24	0.481	0.52	9.581
0.26	0.571	0.54	9.746

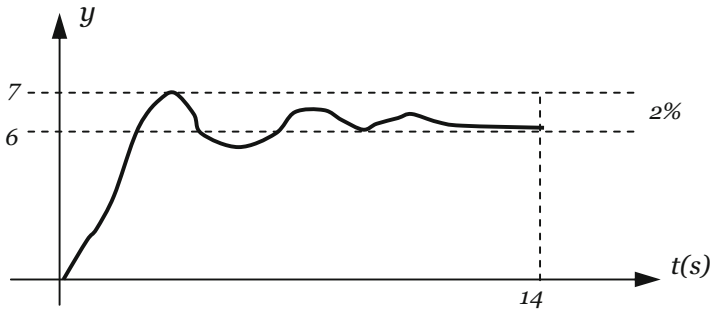


Fig. 4.58 A step response

Table 4.14 The collected data for a step response

t	y
0.0	0.000
0.1	0.005
0.2	0.034
0.3	0.085
0.4	0.140
0.5	0.215
1.0	0.510
1.5	0.700
2.0	0.817
2.5	0.890
3.0	0.932
4.0	1.000
5.0	1.050
6.0	1.010

4.22 The poles and zeros of a process $G(s)$ are described as the poles are at: $s = -4$, $s = -1 \pm j10$, $s = -40$. The zero is at $s= -80$. The DC gain of $G(s)$ is 1. Find $G(s)$. What is its order? Can some of the poles and zeros be neglected? If so which ones and why? Write down the transfer function of the reduced-order model and compare both transfer functions with plotting of step responses.

Lab Projects—System Identifications

Lab Project 4.1
Redo 4.20 using MATLAB Identification Toolbox to identify the system model.

Lab Project 4.2
Table 4.14 shows a collected data set for a step response for a system. Use MATLAB System Identification Toolbox to identify the system model for this system.

Chapter 5
Classical Linear Control Systems—PID Control Systems



As we discussed in previous chapters, in order to obtain the optimal control performances for a control system, a controller is necessary to be added into the control system to control and coordinate the entire system to get the desired outputs. In other words, the main objective of design of a controller for a selected control system is to enable the control system to provide perfect performances to meet the needs of the control target.

Different control strategies and techniques have been developed in recent years, including the linear and nonlinear controllers. In this chapter, we will concentrate on the linear controllers, especially on the Proportional–Integral–Derivative (PID) controller designs since this is a traditional and classic controller widely implemented in each corner in our world.

Traditionally, different control strategies and methods have been developed for the PID controllers used for linear control systems, such as frequency design method, root locus design method, and state-space design method. In this chapter, we will cover all of these methods to provide readers a global and detailed picture about this kind of controller and control system.

5.1 Introduction to PID Control Systems

In order to successfully design and implement a PID controller to any linear control system, the following two components are necessary to know:

(1) The mathematical or dynamic model for the control target or plant.
(2) The control specifications of the control objective.

The first element provides the detailed knowledge and structure about the control target or plant. The more detailed and more accurate about the knowledge for the plant, the better controller can be designed and applied, and the better control performances can be obtained.

© Springer Nature Switzerland AG 2019

Y. Bai and Z. S. Roth, *Classical and Modern Controls with Microcontrollers*,
Advances in Industrial Control, https://doi.org/10.1007/978-3-030-01382-0_5

The second element provides the control objectives represented by a set of concrete or actual parameters, such as settling time, rise time, overshoot, and steady-state error, as we discussed in Sect. 4.3.1 in Chap. 4.

A closed-loop control system is composed of the following components:

- The input $r(t)$ or $R(s)$.
- A comparator used to get the error $e(t)$ or $E(s)$ between the input and the feedback output.
- A controller, such as a PID controller $D(s)$.
- A possible external disturbance w.
- The control process or plant $G(s)$.
- The output of the control system $y(t)$ or $Y(s)$.

Figure 5.1 shows a complete closed-loop control system with all of these elements and signals. The output of the controller, such as a PID controller, is named u and it is the input to the plant. The transfer function for this closed-loop control system is

$$\frac{Y(s)}{R(s)} = \frac{D(s)G(s)}{1 + D(s)H(s)G(s)} \tag{5.1}$$

A typical PID controller is actually composed of three-part system:

(1) **Proportional Compensation K_P**: The main function of the proportional compensator is to introduce a gain that is proportional to the error reading which is produced by comparing the system's output and input.
(2) **Derivative Compensation K_D**: In a unitary feedback system, the derivative compensator will introduce the derivative of the error signal multiplied by a gain K_D. In other words, the slope of the error signal's waveform is what will be introduced to the output. Its main purpose is to improve the transient response of the overall closed-loop system.
(3) **Integral Compensation K_I**: In a unitary feedback system, the integral compensator will introduce the integral of the error signal multiplied by a gain K_I. This means that the area under the error signal's curve will be affecting the output signal. It is important to note that this integral part will improve the steady-state error of overall closed-loop system.

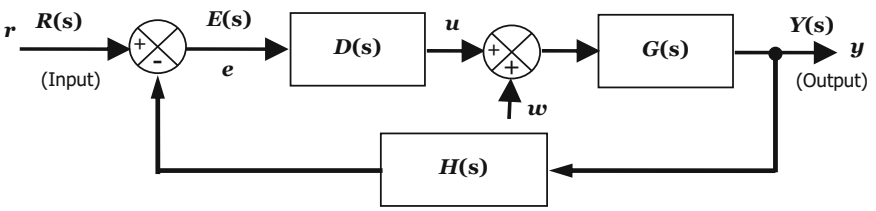


Fig. 5.1 A complete closed-loop control system with all elements

Table 5.1 shows the definitions for three control gains, P, I, and D, and their representations in two domains, time and s frequency domains.

It can be found from this table that each control gain has different functions, and each function can also be called a compensation to compensate the control system to get the desired control output.

- The function of the proportional gain K_P can be used to amplify the output and makes it linearly proportional to the error between the input and the feedback output, $u = K_P\, e(t)$. The advantage of increasing the K_P is that the steady-state error can be reduced (Sect. 4.4.2 in Chap. 4: $e_{ss} = 1/1 + K_P$), but a very large proportional gain may make the system unstable or instability. It can be seen that it cannot effectively improve the entire performance for a control system by only using the proportional gain K_P. Therefore, there is a trade-off in selecting the proportional gain to make the system both stable and small steady-state errors.
- The integral control with control gain K_I plays an accumulation role to accumulate all previous and current errors together, such as

$$u(t) = K_I \int_0^t e(x)\mathrm{d}x \quad \text{and} \quad D(s) = \frac{K_I}{s} \tag{5.2}$$

  Therefore, previous errors will charge up the integrator to some value that will remain, even if the current input error becomes zero. This feature means that any disturbance w can be accommodated with zero error because it is no longer necessary for the error to be finite to produce a control that will negate the disturbance. The primary purpose of the integral control is to reduce or eliminate the steady-state errors or reduce the effects of the disturbances. However, the benefit always comes at the cost of reducing the stability of the entire system.
- The purpose of the derivative control with control gain K_D is to improve the stability of a system. However, generally it is always used in conjunction with a proportional and/or integral part together to obtain that purpose. The derivative control has the form

Table 5.1 The definitions of the PID controller and three parameters

Compensation	Time domain	*S*-domain
`Proportional`	$K_P e(t)$	K_P
`Derivative`	$K_D \frac{\mathrm{d}e(t)}{\mathrm{d}t}$	$K_D s$
`Integral`	$K_I \int_0^t e(x)\mathrm{d}x$	$\frac{K_I}{s}$
`PID Compensation`	$K_P e(t) + K_D \frac{\mathrm{d}e(t)}{\mathrm{d}t} + K_I \int_0^t e(x)\mathrm{d}x$	$K_P + K_D s + \frac{K_I}{s} = \frac{K_P s + K_D s^2 + K_I}{s}$

$$u(t) = K_{\mathrm{D}} \frac{\mathrm{d}e(t)}{\mathrm{d}t} \quad \text{and} \quad D(s) = K_{\mathrm{D}} s \tag{5.3}$$

If we combine all of three controls together, we get a PID controller with the following control function:

$$K_{\mathrm{P}} e(t) + K_{\mathrm{D}} \frac{\mathrm{d}e(t)}{\mathrm{d}t} + K_{\mathrm{I}} \int_0^t e(x)\mathrm{d}x \tag{5.4}$$

With all of three control gains, a PID control system is able to provide an acceptable degree of error reduction simultaneously with required stability and damping ratio to obtain an optimal control performance for a closed-loop control system. The transfer function is given by

$$D(s) = \left(K_{\mathrm{P}} + K_{\mathrm{D}} s + \frac{K_{\mathrm{I}}}{s}\right) E(s) = \frac{K_{\mathrm{P}} s + K_{\mathrm{D}} s^2 + K_{\mathrm{I}}}{s} E(s) \tag{5.5}$$

The PID controllers are considered as conventional or traditional controllers since they have been proved to be so efficient to provide acceptable control function in most modern control applications, including the chemical process industries, petroleum refining, paper production, metal forming, automobile making, ATM machine controlling, and so on.

The process of successfully designing and applying a PID controller to a control system regularly needs the following procedure:

(1) Obtain the dynamic or mathematical model of the control target or the plant.
(2) Obtain the control objectives or specifications, such as rise time, settling time, overshoot, and steady-state error for a step response.
(3) Use different methods to design an optimal PID controller for the selected plant based on two sets of parameters shown above (1) and (2).
(4) Perform simulations to tune three gains to try to get the optimal control performances for the control system.
(5) Perform the actual tuning of three control gains to make the control system as perfect as possible to the ideal control system to get satisfactory output performance.

In the following sections, we discuss how to perform these steps to complete the design and tuning processes with three methods.

5.2 Design PID Controllers with Root Locus Method

The performance of a closed-loop control system can be described in terms of the specifications in the time domain or in the frequency domain. Furthermore, the stability and steady-state error are also important criteria to evaluate a stable control system. These performance specifications can be defined in terms of the desired location of the poles and zeros of the closed-loop transfer function in the *s*-plane. Also, the locus of the roots of the closed-loop control system can be obtained from the variation of one system parameter. However, if the locus of roots does not result in a suitable root configuration, some additional compensating elements or networks should be added to change the root locus as the parameter is changed. Therefore, we can use the root locus method to determine a suitable compensator network so that the resultant root locus can obtain the desired closed-loop root configuration.

Traditionally, a PID controller can be obtained by a cascaded PI and PD controllers or called compensators since they do play a compensating role to make the system better performances. First let's have a closer look at the design of PD controller, which is also called a phase lead compensator for a control system since it provides a phase leading function.

First, let's have a closer look at the root locus method.

5.2.1 Introduction to Root Locus Method

When using the root locus to design control systems, it can be used to design the damping ratio ζ and natural frequency ω_n of a feedback system. Lines of constant damping ratio can be drawn radially from the origin and lines of constant natural frequency can be drawn as arcs whose center points coincide with the origin. By selecting a point along the root locus that coincides with a desired damping ratio and natural frequency, a gain K can be calculated and implemented in the controller.

5.2.1.1 The Effects of Poles and Zeros on Time-Domain Specifications

As we discussed in Sect. 4.3 in Chap. 4, there are certain relationships between the poles location and the specifications of the related step response in the time domain for a standard second-order system, which is shown in Fig. 5.2.

For a standard second-order control system with no finite zeros, these relationships are

$$t_r \approx \frac{1.8}{\omega_n} \tag{5.6}$$

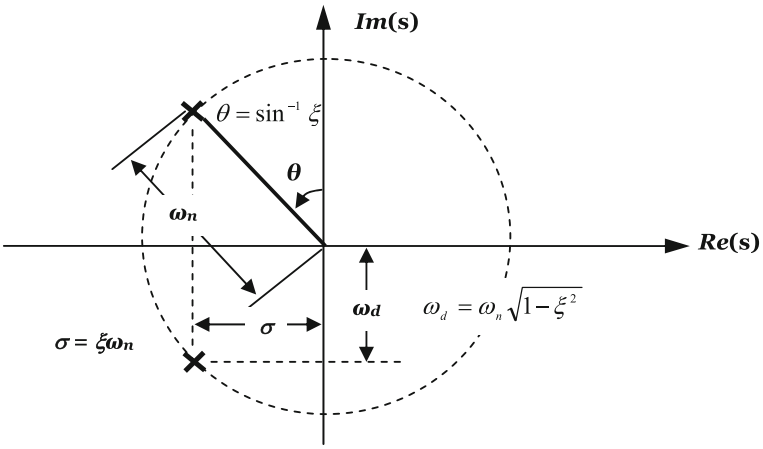


Fig. 5.2 Relationships between the poles location and specifications of step response

$$t_s \approx \frac{4.6}{\xi\omega_n} = \frac{4.6}{\sigma} \tag{5.7}$$

$$M_p \approx e^{-\pi\xi/\sqrt{1-\xi^2}} \approx \left(1 - \frac{\xi}{0.6}\right) \quad (0 \le \xi \le 0.6) \tag{5.8}$$

Figure 5.3 shows the effects of different damping ratios ξ on the step response.

It can be found from Fig. 5.3 that the system becomes unstable as the damping ratio is zero, but the system response will become more stable when the damping

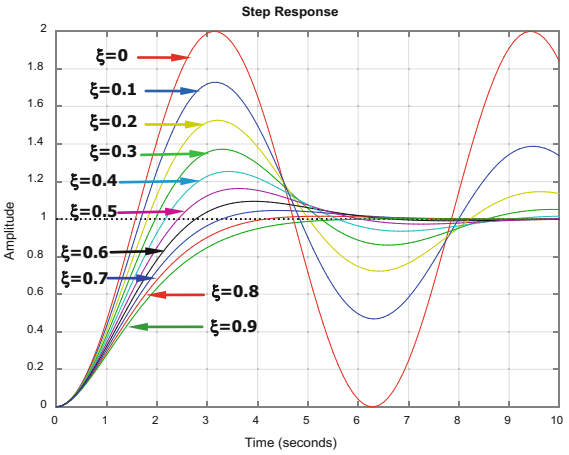


Fig. 5.3 Effects of damping ratio on step response

ratio increases. However, the settling time becomes longer and longer as the damping ratio increases. The ideal or the most optimal damping ratio is 0.707, at which the overshoot has the minimum value.

Figure 5.4 shows the effects of different poles locations on the damping ratios on the *s*-plane. The optimal damping ratio is 0.707, or the angle between the vector of the dominant pole and the Imaginary axis is 45°.

The effects of adding poles and zeros on the original system can be summarized as follows:

(1) If a phase lead (PD) compensator is designed as a controller, it is equivalent to adding a derivative function to the original system since the Laplace transform of d*f*/d*t* is *sF*(*s*), which is also equivalent to adding a zero to the transfer function. The effects of this phase lead compensator on the time domain are to increase the overshoot, but improve or reduce the rise time of the step response of the original system. The effects of adding zeros on the *s*-plane are to move the locus to the left. But adding a zero in the right plane may depress the overshoot, but may cause the step response to start in wrong direction.
(2) By adding a pole to a system, a phase lag or PI controller can be designed. The effects of adding a pole to the left plane on the original system are to reduce the steady-state error since the Laplace transform of an integral item is *F*(*s*)/*s*, which is equivalent to adding a pole to the transfer function. The effects of adding poles to the *s*-plane are to move the locus to the right. The shortcoming is that the rise time may be worse and becomes longer for a step response of the original system.

These effects and relationships are very useful when using root locus method to design PID controllers for a control system to improve the control performance in the following sections.

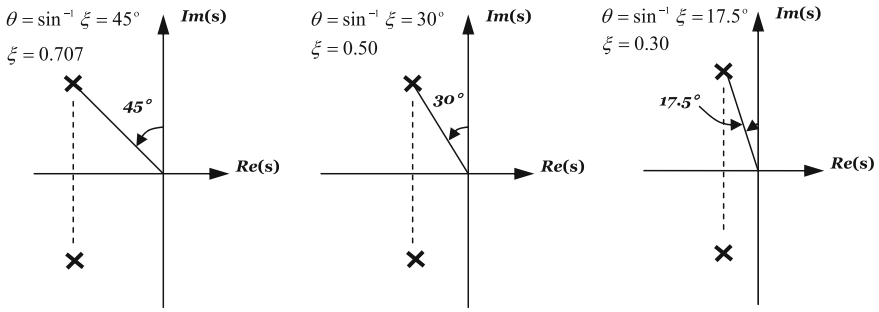


Fig. 5.4 Effects of variation of poles locations on the damping ratio

5.2.1.2 The Major Properties of the Root Locus Method

When using the root locus method to design the control system, two conditions must be met:

(1) Angle condition and
(2) Magnitude condition.

As we know, the root locus is a plot or a graphical representation of the poles' trajectory as the closed-loop gain K is changed. A typical closed-loop control system that has no compensator is shown in Fig. 5.5, and it can be represented by the closed-loop transfer function.

$$T(s) = \frac{Y(s)}{R(s)} = \frac{KG(s)}{1 + KG(s)H(s)} = \frac{N(s)}{D(s)} \tag{5.9}$$

where $N(s)$ is the numerator polynomial and $D(s)$ is the denominator polynomial of the transfer functions, respectively. The closed-loop poles of this transfer function are the roots of the characteristic equation $D(s) = 1 + KG(s)H(s) = 0$, or $KG(s)H(s) = -1$.

If the system has no pure delay, the product of $KG(s)H(s)$ is a rational polynomial function and may be expressed as (K is the original gain included in the $G(s)$)

$$KG(s)H(s) = K\frac{A(s)}{B(s)} = K\frac{(s+z_1)(s+z_2)\ldots(s+z_m)}{(s+p_1)(s+p_2)\ldots(s+p_n)} = -1 \tag{5.10}$$

where $-z_i$ are the m zeros, $-p_i$ are the n poles, and K is a scalar gain. Typically, a root locus diagram will indicate the transfer function's pole locations for all varying values of the parameter K. A root locus plot will be all those points in the s-plane where $KG(s)H(s) = -1$ for any value of K.

If we convert each factor $(s + z_i)$ and $(s + p_i)$ in (5.10) to their vector equivalents, $A_i e^{j\theta_i}$ and $B_i e^{j\varphi_i}$, respectively, Eq. (5.10) can be rewritten as

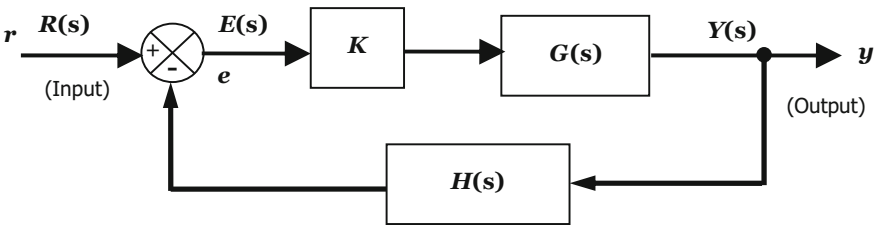


Fig. 5.5 A closed-loop control system

$$KG(s)H(s) = K\frac{A(s)}{B(s)} = K\frac{A_1A_2\ldots A_m e^{j(\theta_1+\theta_2+\cdots+\theta_m)}}{B_1B_2\ldots B_n e^{j(\varphi_1+\varphi_2+\cdots+\varphi_n)}}$$
$$= K\frac{A_1A_2\ldots A_m}{B_1B_2\ldots B_n} e^{j(\theta_1+\theta_2+\cdots+\theta_m-(\varphi_1+\varphi_2+\cdots+\varphi_n))} = -1 \tag{5.11}$$

Combining (5.10) and (5.11), we can get the angle condition (5.12) and magnitude condition (5.13) as below:

$$e^{j(\theta_1+\theta_2+\cdots+\theta_m-(\varphi_1+\varphi_2+\cdots+\varphi_n))} = -1 = e^{j(\pi+2k\pi)}$$
$$\text{means } (\theta_1+\theta_2+\cdots+\theta_m)-(\varphi_1+\varphi_2+\cdots+\varphi_n) = \pi+2k\pi$$
$$\text{or} \tag{5.12}$$
$$\sum_{i=1}^{m}\angle(s+z_i)-\sum_{i=1}^{n}\angle(s+p_i) = \pi+2k\pi \quad (k=0,1,2,\ldots)$$

$$K\left|\frac{A(s)}{B(s)}\right| = K\frac{|s+z_1||s+z_2|\ldots|s+z_m|}{|s+p_1||s+p_2|\ldots|s+p_n|} = \left|K\frac{A_1A_2\ldots A_m}{B_1B_2\ldots B_n}\right| = K\frac{A_1A_2\ldots A_m}{B_1B_2\ldots B_n} = 1 \tag{5.13}$$

When $k = 0$, Eq. (5.12) becomes

$$\sum_{i=1}^{m}\angle(s+z_i)-\sum_{i=1}^{n}\angle(s+p_i) = \pi \tag{5.14}$$

The angle condition and magnitude condition provide a limitation or criterion to find or check whether a point is on the root of the poles for a control system. Any pole located in the root locus plot must meet both conditions; otherwise, the point is not on the root locus.

5.2.2 *Sketching Root Locus*

The root locus is a powerful tool to design and analyze closed-loop control systems, especially in the phase domain or s-plane. It is not easy to sketch a complete and accurate root locus for any control system, especially for some complicated systems. Fortunately, with the help of modern computer-aided tools, such as MATLAB Control System Toolbox, this job can be performed easily and quickly. However, with some simple techniques, we can still develop some procedures to help users to quickly draw or sketch a root locus for most often used systems.

Now we need to convert the magnitude equation into a more convenient form. Equation (5.10) can be rewritten as

$$K\frac{A(s)}{B(s)} = -1 \quad \text{or} \quad \frac{A(s)}{B(s)} = -\frac{1}{K} \tag{5.15}$$

The following points should be kept in mind when drawing a root locus graph:

- The number of the branches of the root locus is equal to the number of closed-loop poles.
- On the real axis for gain $K > 0$, the root locus exists to the left of an odd number of real axis, finite open-loop poles, and/or finite open-loop zeros.
- The root locus begins at the finite and infinite poles of $KG(s)H(s)$ and ends at the finite and infinite zeros of $KG(s)H(s)$.
- When the gain K is small, the poles start at the poles of the open-loop transfer function.
- When gain K becomes infinity, the poles move to overlap the zeros of the system. This means that on a root locus graph, all poles move toward zeros. Only one pole may move toward one zero, and this means that there must be the same number of poles as zeros.
- If there are fewer zeros than poles in the transfer function, there are a number of implicit zeros located at infinity, which the poles will approach.
- Since a root locus line starts from each pole, therefore any place that two poles appear to be connected by a root locus line on the real axis, the two poles actually move toward each other, and then they breakaway and move off the axis. The point where the poles break off the axis is called the **Breakaway Point**. From here, the root locus lines travel toward the nearest zero.
- It is important to note that the s-plane is symmetrical about the real axis, so whatever is drawn on the top half of the s-plane must be drawn in mirror image on the bottom half plane.
- Once a pole breaks away from the real axis, it can either move out toward infinity to meet an implicit zero, or it can travel to meet an explicit zero, or it can re-join the real axis to meet a zero that is located on the real axis. If a pole is moving toward infinity, it always follows an asymptote. The number of **asymptotes** is equal to the number of implicit zeros at infinity.
- The root locus approaches straight lines as the locus approaches the infinity. Also, the equation of the asymptotes is given by the real axis intercept, α, and angle, θ, as follows:

$$\begin{aligned}\alpha &= \frac{\sum \text{Finite Poles} - \sum \text{Finite Zeros}}{(\text{Number of finite poles}) - (\text{Number of finite zeros})} \\ &= \frac{\sum \text{Finite Poles} - \sum \text{Finite Zeros}}{P - Z}\end{aligned} \tag{5.16}$$

$$\theta_k = \frac{(2k+1)\pi}{(\text{Number of finite poles } P) - (\text{Number of finite zeros } Z)} = \frac{(2k+1)\pi}{P - Z} \tag{5.17}$$

where $k = 0, \pm1, \pm2, \pm3, \ldots$, and the angle is given in radians with respect to the positive extension of the real axis.

Keep those points in mind, and now follow the steps below to draw a root locus graph:

(1) Mark all open-loop poles with **x** and zeros with **o**.
(2) Draw the locus on the real axis to the left of an odd number of real poles plus zeros.
(3) Find asymptotes. Let P be the number of poles and Z be the number of zeros. The number of asymptotes = $P - Z$.
The asymptotes intersect the real axis at α points (called the centroid) and depart at angle ϕ_l given by

$$\alpha = \frac{\sum \text{Finite Poles} - \sum \text{Finite Zeros}}{(\text{Number of finite poles } P) - (\text{Number of finite zeros } Z)} = \frac{\sum p - \sum z}{P - Z} \tag{5.18}$$

$$\phi_l = \frac{180^\circ + (l-1)360^\circ}{P - Z} \quad (l = 1, 2, 3, \ldots P - Z) \tag{5.19}$$

One can find that (5.19) is the same as (5.17); yes, they are the same equation with different representing ways.
(4) Compute loci departure angles from the poles and arrival angles at the zeros using the phase condition.
(5) Compute breakaway/break-in points. The breakaway points are located at the roots of the following equation:

$$\frac{\mathrm{d}G(s)H(s)}{\mathrm{d}s} = 0 \tag{5.20}$$

(6) Compute the crossing of the imaginary axis. Use $j\omega$ to replace s in the characteristic equation, and make both real and imaginary parts in the equation equal to zero. By solving both equations (if the solution exist), we can get the crossing of the imaginary axis. If no solution exists, there is no crossing of the imaginary axis. Another way to confirm that if there is no crossing of the imaginary axis, the first column of the Routh array should have no sign change, or all coefficients in the first column are positive.

Not all of the above steps are necessary when plotting some simple root locus graphs for the lower order control system, and some steps can be ignored or skipped to make the plot simple.

For example, we have a system with the transfer function as

$$T(s) = K\frac{(s+1)}{(s+2)(s+3)}$$

Try to plot its root locus graph with the following sequence:

(1) Mark all open-loop poles with **x** and zeros with **o**.
(2) Draw the locus on the real axis to the left of an odd number of real poles plus zeros.
(3) Find asymptotes (not all points are in the root locus).

$$\alpha = \frac{\sum \text{Finite Poles} - \sum \text{Finite Zeros}}{P - Z} = \frac{-2 + (-3) - (-1)}{2 - 1} = -4$$
$$\theta_k = \frac{(2k+1)\pi}{P - Z} = \frac{(2k+1)\pi}{2 - 1} = (2k+1)\pi \text{ when } k = 0, \theta_k = \pi.$$

Since this is a very simple transfer function, therefore, we do not need to use all of the steps listed above. The complete root locus graph is shown in Fig. 5.6.

Now take a look at another example with the following characteristic equation or polynomial:

$$T(s) = \frac{1}{s(s+2)(s+3)}; \text{ or } KG(s)H(s) = K\frac{A(s)}{B(s)} = K\frac{1}{s(s+2)(s+3)}$$

(1) Mark all open-loop poles with **x** and zeros with **o** as shown in Fig. 5.7. Since we have three poles, $s_1 = 0$, $s_2 = -2$, and $s_3 = -3$, with no finite zeros, $m = 0$ and $n = 3$. Three branches of the root locus exist for this root locus graph.
(2) Find asymptotes.

$$\alpha = \frac{\sum \text{Finite Poles} - \sum \text{Finite Zeros}}{P - Z} = \frac{0 + (-2) + (-3) - 0}{3} = -\frac{5}{3} = -1.67$$
$$\theta_k = \frac{(2k+1)\pi}{P - Z} = \frac{(2k+1)\pi}{3 - 0} = \frac{(2k+1)\pi}{3}; \text{ as } k = 0,\ \theta_0 = \pm\frac{\pi}{3},\ k = 1,\ \theta_1 = \pm\pi.$$

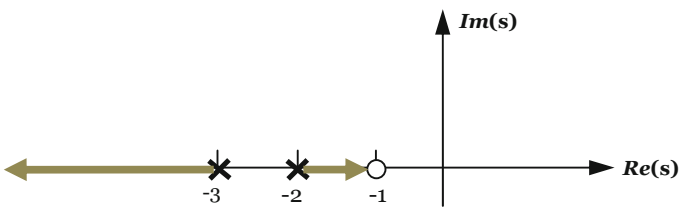


Fig. 5.6 The complete root locus graph

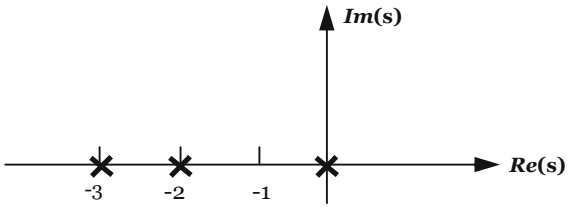


Fig. 5.7 Mark all open-loop poles and zeros

So the intersect of asymptotes is at $\alpha = -1.67$. Since there are three poles with no finite zeros, thus, three poles should move to try to meet three infinity zeros following the asymptotes, which are $\pm 60°$ and $\pm 180°$, respectively. Make these asymptotes and intersect point on the root locus graph, as shown in Fig. 5.8.

(3) Compute breakaway/break-in points. Since $A(s) = 1$, $B(s) = s^3 + 5s^2 + 6s$. The characteristic equation is $1 + KG(s)H(s) = 0$, or $KA(s) + B(s) = 0$, we have

$$s^3 + 5s^2 + 6s + K = 0;$$

and

$$\frac{\mathrm{d}(s^3 + 5s^2 + 6s)}{\mathrm{d}s} = 3s^2 + 10s + 6 = 0$$

Solving this equation, we have two roots, $s = -0.78$ and $s = -2.55$, since not all of these roots are on the locus. Of these two real roots, there exists one root at $s = -0.78$ on the locus (when $K > 0$). Thus, the breakaway (or break-in) point is $s = -0.78$, which is shown in Fig. 5.8 as a square.

(4) Find the points of the root locus crossing the imaginary axis. Using $s = j\omega$ and putting it into characteristic equation $1 + KG(s)H(s) = 0$, we have

$$(j\omega)^3 + 5(j\omega)^2 + 6j\omega + K = 0$$

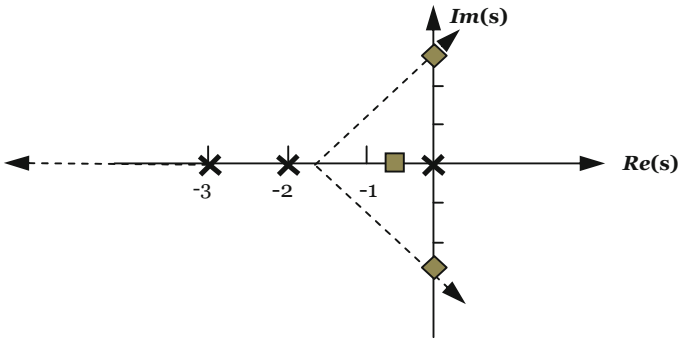


Fig. 5.8 Plot asymptotes and intersect point on the real axis

Equating the real and imaginary parts to zero, we get

$$-5\omega^2 + K = 0 \quad \text{(a)}$$

$$\omega(6 - \omega^2) = 0 \quad \text{(b)}$$

From (b), we get $\omega = \sqrt{6} = 2.54$; put it into (a), we get $K = 30$. Because of the symmetry property on the root locus graph, $\omega = -2.54$ is also a crossing point of the imaginary axis. Two crossing points on the imaginary axis are shown as diamond in Fig. 5.8.

(5) Compute loci departure angles from the poles and arrival angles at the zeros using the phase condition. Since there are no complex poles in loop gain, there are no angles of departure. Also, there are no complex zeros in loop gain, and hence no angles of arrival.

The root locus plotted with MATLAB is shown in Fig. 5.9. One can click on any point in the locus to get a black square point, and drag to move this point along the locus to get the real values for this root. Figure 5.9 shows all values for the point crossing imaginary axis.

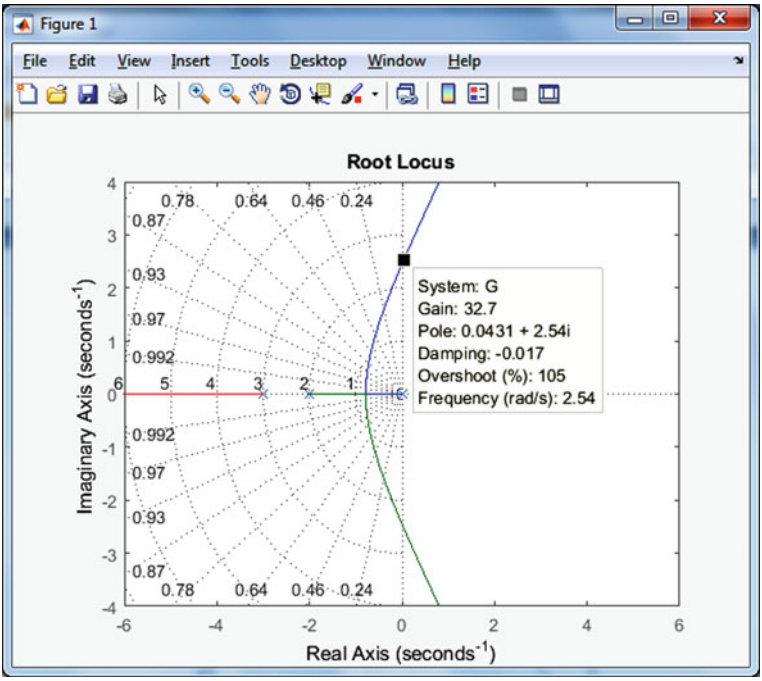


Fig. 5.9 The complete root locus graph for the example transfer function

Although the manual plotting method can be used to get quick approximations about the root locus graph, however, it is hard to obtain the accurate and correct results for some relatively complicated or higher order systems. Therefore, an easy and fast way to get accurate and correct root locus graphs is to use MATLAB Control System Toolbox functions, especially the `rlocus()` function we discussed in Sect. 4.4.5.3 in Chap. 4.

Next, let's discuss how to use the root locus method to design desired closed-loop control system. We need to discuss the phase lead (PD), phase lag (PI), and PID controllers by using this method. First, let's concentrate on the phase lead controller or compensator.

5.2.3 *General Considerations of Designing Compensator with Root Locus*

Generally, the root locus is a graphical representation or examination that is used to check how the roots of a closed-loop control system vary with changing of certain system parameter, such as the gain K. Exactly, the root locus plots the poles of the closed-loop transfer function in the complex s-plane as a function of a gain parameter K. This is a technique used as a stability criterion in the field of classical control theory developed by Walter R. Evans which can determine stability of the system.

Regularly, to manually design a compensator for a control system with the root locus method, the following steps should be followed:

(1) Obtain the system specifications and translate them into a desired root location for the dominant roots. Exactly, this step is used to evaluate all design transient response specifications to determine possible location for the dominant poles.
(2) Sketch the uncompensated root locus to determine whether the desired root locations can be obtained with the uncompensated system.
(3) If a compensator is needed, evaluate all steady-state error specifications to determine the system's type to decide how many zeros (derivatives) or poles (integrators) should the compensator $C(s)$ have.
(4) If the process $G(s)$ is unstable, use root locus sketches to determine the complexity of the $C(s)$ needed for stabilization of the loop, and meeting the requirements of steps 1 and 2.
(5) Set a root locus angle condition equation at the dominant poles location obtained from step 1, and solve it as you gradually increase the complexity of the $C(s)$.
(6) Find the needed gain K, using root locus magnitude condition.
(7) Check if the dominant poles are indeed dominant.

When using computer-aided root locus design method, such as MATLAB Control System Toolbox, to design a controller, only steps 3 and 4 are needed.

In most real designs, there is no pure derivative (D) or integral element (I) to work as a complete compensator. Instead most times it is to make a PD or a PI compensator to perform the control functions. As we mentioned, the effects of adding zeros or poles to a system in the s-plane or root locus can be summarized:

(1) Adding zeros to a system is to equivalent to moving the root locus to the left in the s-plane. The closer the zeros to the origin, the greater the effect. However, when zeros are far away to the origin, the effects will be reduced and become weaker and weaker as zeros are located further away from the origin.
(2) Adding poles to a system is equivalent to moving the root locus to the right in the s-plane. The closer the poles to the origin, the greater the effect. But when poles are far away to the origin, the effect will be decreased and becomes weaker and weaker as poles are further away from the origin.

Keep these properties in mind; now let's have a closer look at some important features about different compensators' design and applications.

Regularly, the most often used compensator is a first-order compensator that has single zero and pole with the following transfer function:

$$D(s) = \frac{s+z}{s+p} \tag{5.21}$$

Depending on the different locations the zero and pole are located, as shown in Fig. 5.10, we can have different compensators.

In Fig. 5.10a, for a given desired pole $s = \sigma + j\omega$ that should be in the root locus, the total transfer function angle $\theta_D = \theta_z - \theta_p$. If $|z| < |p|$ as shown in Fig. 5.10a, which means that the effect of zero is greater than that of the pole, the total phase $\theta_D = \theta_z - \theta_p > 0$ or is positive. This kind of compensator is called a phase lead controller. On the other hand, if $|z| > |p|$ as shown in Fig. 5.10b, the total phase $\theta_D = \theta_z - \theta_p < 0$ or is negative, and the compensator is known as a phase lag controller.

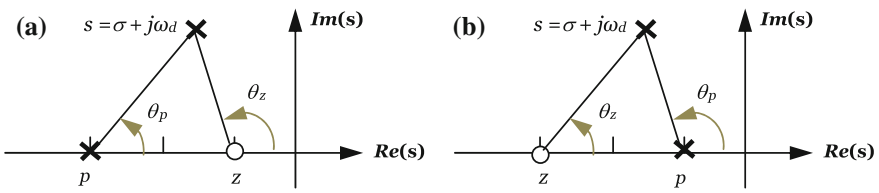


Fig. 5.10 Phase lead and phase lag compensator

5.2.4 Design of a Phase Lead Compensator with Root Locus Method

As we discussed in Sect. 5.2.1.1, if a phase lead or a PD compensator is designed as a controller and added into a system, it is equivalent to adding a derivative function to the original system since the Laplace transform of d*f*/d*t* is $sF(s)$, which is also equivalent to adding a zero to the transfer function. The effects of this phase lead compensator on the time domain are to increase the overshoot, but improve or reduce the rise time of the step response of the original system. Instead, adding a zero in the right plane may depress the overshoot but may cause the step response to start in wrong direction.

In fact, to use root locus method to design a controller for an existing control system is to manipulate the open-loop poles and zeros by adding, deleting, shifting, and canceling some of them while keeping track of the closed-loop poles via root locus plots and the closed-loop zeros.

Two criteria to make sure that the design result is correct are two conditions: Angle and magnitude conditions. By adding a zero or a pole on the s-plane to improve the control performance of the original control system, make sure that the added zero or pole is located on the root locus by using these two conditions.

Generally, the gain K has the following effects on the designed system:

- For small K: an over-damped response can be obtained.
- For larger K: an under-damped response can be obtained.
- The larger K the larger the overshoot, the larger the frequency of oscillations, the smaller the damping.

Regularly, to design a phase lead compensator for a control system with the root locus method, the following steps should be followed:

(1) Obtain the system specifications and translate them into a desired root location for the dominant roots. Exactly, this step is used to evaluate all design transient response specifications to determine possible location for the dominant poles.
(2) Sketch the uncompensated root locus to determine whether the desired root locations can be obtained with the uncompensated system.
(3) If a compensator is needed, place the zero of the phase lead network directly below the desired root location or to the left of the first two real poles.
(4) Determine the pole location so that the total angle at the desired root location is 180° and therefore is on the compensated root locus.
(5) Check the total system gain at the desired root location and then calculate the error constant.
(6) Repeat the steps above if the error constant is not satisfactory.

Now we use an example to illustrate how to use root locus method to design a phase lead compensator. The original open-loop plant has the transfer function as

$$G(s) = \frac{1}{s(s+1)} \tag{5.22}$$

The specifications for this closed-loop control system are (refer to Fig. 5.2)

(1) $\xi = 0.707$ and the dominant closed-loop poles are $s_{1,2} = -2 \pm j2$.
(2) The steady-state error is zero.

The damping ratio $\xi = 0.707$ is equivalent to having $\theta = 45°$ (refer to Fig. 5.2), and the dominant closed-loop $s_{1,2} = \sigma \pm j\omega_d = -2 \pm j2$ means that the $\omega_n = \sqrt{8} = 2.8$, and $\omega_d = 2$.

The controller or compensator is a phase lead ($p \gg z$) with the transfer function

$$D(s) = K\frac{s+z}{s+p} \tag{5.23}$$

When the pole p is far away from the origin, its effect on the entire system is smaller than that of the zero z, and it is phase lead compensator (phase of $z >$ phase of p).

Now we need to decide where these zero and pole should be located for the compensator. Based on condition (3) listed above, we should place a zero directly under the desired pole on the real axis, which should be -2 (since $s_{1,2} = \sigma \pm j\omega_d = -2 \pm j2$). Next, we need to select the pole's location for the compensator. To do this, we can try to use different poles. Figures 5.11b and 5.12 show the effects of poles when they are located at different points on the real axis.

Compared with the original uncompensated system in Fig. 5.11a, the locus is moved to the left after the zero $z = -2$ (with a pole $p = -5$) is added into the compensator.

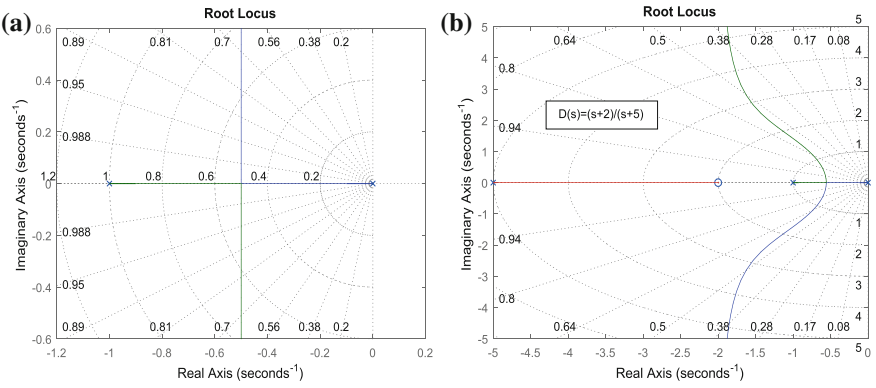


Fig. 5.11 The uncompensated and compensated systems

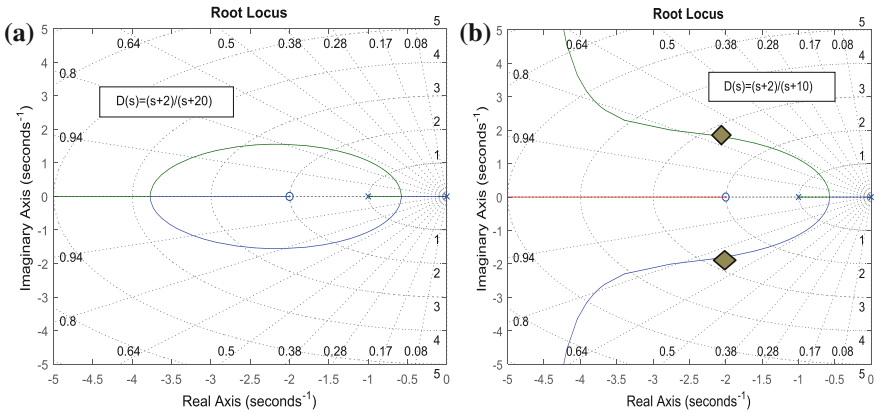


Fig. 5.12 Effects of different compensated systems

If we continue to increase the value of the pole to −20 that is shown in Fig. 5.12a, which is equivalent to moving the pole far away from the origin, the effect of the pole will be reduced and this makes the locus move to left a little bit.

By trying to use different poles with different locations, we found that in order to enable our desired poles $s_{1,2} = \sigma \pm j\omega_d = -2 \pm j2$ to be located on the root locus as two diamonds shown in Fig. 5.12b, the best pole's location should be $p = -10$ and the compensator should be

$$D(s) = K\frac{s+z}{s+p} = K\frac{s+2}{s+10} \tag{5.24}$$

The compensated root locus with $p = -10$ is shown in Fig. 5.12b.

Now let's check the total angle after those zero and pole have been added into the system based on the angle condition to confirm our selection.

Based on the illustration shown in Fig. 5.13, all angles can be calculated as follows:

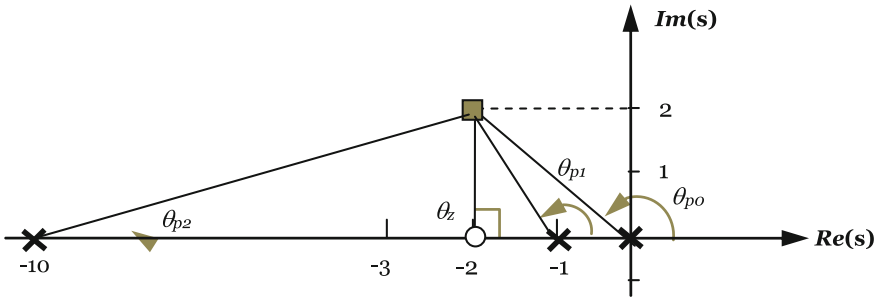


Fig. 5.13 The total angles on the compensated system

$$\theta_{p0} = 180^\circ - \tan^{-1}\left(\frac{2}{2}\right);\ \theta_{p1} = 180^\circ - \tan^{-1}\left(\frac{2}{1}\right);\ \theta_z = 90^\circ;\ \theta_{p2} = \tan^{-1}\left(\frac{2}{8}\right)$$

Based on the angle condition, which is shown in Eqs. (5.12) and (5.14),

$$\sum_{i=1}^{m} \angle(s+z_i) - \sum_{i=1}^{n} \angle(s+p_i) = \pi + 2k\pi$$

and

$$\sum_{i=1}^{m} \angle(s+z_i) - \sum_{i=1}^{n} \angle(s+p_i) = \pi \text{ when } k = 0$$

We have $\theta_z - (\theta_{p0} + \theta_{p1} + \theta_{p2}) = 90^\circ - (135^\circ + 117^\circ + 14^\circ) = -176^\circ \neq -180^\circ = \pi$.

This means that our estimation is not correct and need to redo the calculation to get the accurate result. Suppose the zero is located at real axis with its absolute value $|z| > 2$, as shown in Fig. 5.14 (only positive conjugate pole is shown). The θ_z can be calculated as

$$\theta_z = \tan^{-1}\left(\frac{2}{-2-z}\right)$$

Then applying the angle condition as above, we have

$$\theta_z - (\theta_{p0} + \theta_{p1} + \theta_{p2}) = \tan^{-1}\left(\frac{2}{-2-z}\right) - (135^\circ + 117^\circ + 14^\circ) = 180^\circ$$

Solving this equation, we can get $z = -2.14$ and $\theta_z = 86^\circ$. Using this value and putting it into the angle condition, we have

$$\begin{aligned}\theta_z - (\theta_{p0} + \theta_{p1} + \theta_{p2}) &= 86^\circ - (135^\circ + 117^\circ + 14^\circ) \\ &= 86^\circ - 266^\circ = -180^\circ = -\pi\end{aligned}$$

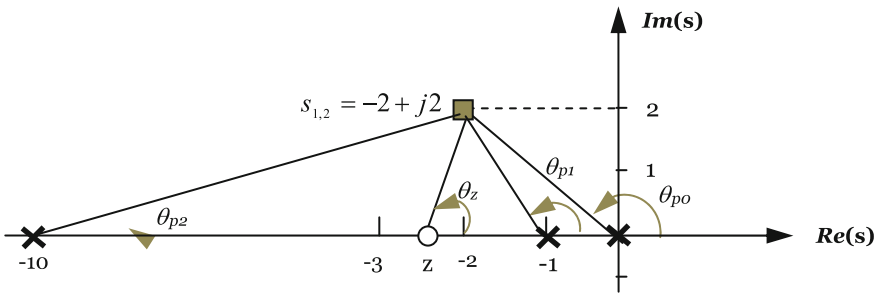


Fig. 5.14 The actual locations of zero and pole for the compensated system

The actual compensator is

$$D(s) = K\frac{s+z}{s+p} = K\frac{s+2.14}{s+10} \tag{5.25}$$

Finally, let's decide the value for the gain K. We need to use the magnitude condition to derive this parameter's value. The desired roots are $s_{1,2} = \sigma \pm j\omega_d = -2 \pm j2$. Refer to Fig. 5.14 and Eq. (5.13), the magnitudes of the characteristic equation become

$$1 + D(s)G(s) = 1 + K\frac{|(s+2.14)|}{|s||(s+1)||s+10|} = 0; \text{ or } K\frac{|s+2.14|}{|s||s+1||s+10|} = 1 \tag{5.26}$$

Using the positive desired pole $s_1 = -2 + j2$ to replace the s in (5.26), we get the magnitude condition as

$$K = \frac{|s||s+1||s+10|}{|s+2.14|} = \frac{|-2+j2||-2+j2+1||-2+j2+10|}{|-2+j2+2.14|}$$
$$= \frac{\sqrt{2^2+2^2} * \sqrt{2^2+1^2} * \sqrt{2^2+8^2}}{\sqrt{2^2+0.14^2}} = \sqrt{\frac{2720}{4.0196}} = 26$$

The root locus and the step response for the compensated closed-loop system are shown in Fig. 5.15a, b. The overshoot is small (about 15%) and the settling time is about 2.5 s.

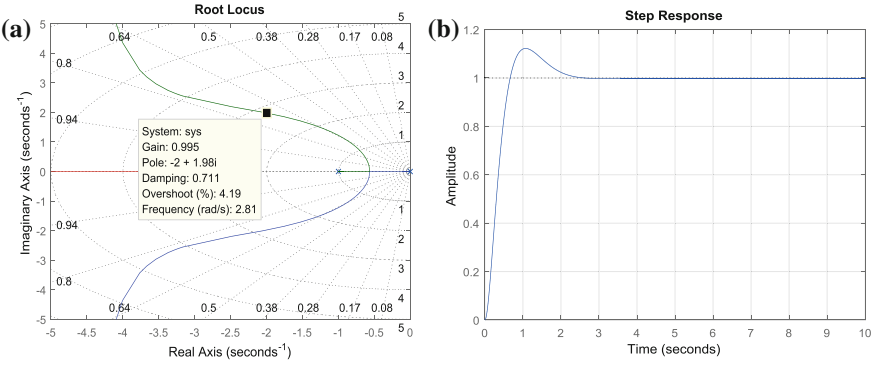


Fig. 5.15 Step response of the phase lead compensated system

5.2.5 *Design of a Phase Lag Compensator with Root Locus Method*

As we can see from Fig. 5.10, the only difference between a phase lead and a phase lag compensator is the location of the zero and the pole on the real axis. If a zero is located more closely to the origin than that of a pole, a phase lead compensator is obtained. Otherwise, if a pole is located more closely to the origin than that of a zero, a phase lag compensator can be achieved.

The design method and procedure for a phase lag compensator is similar to those steps used for a phase lead compensator. Now let's use another example to illustrate how to use root locus method to design a phase lag compensator.

Suppose we have a plant with the following transfer function:

$$G(s) = \frac{1000}{(s+5)(s+100)(s+200)} \tag{5.27}$$

The design specifications including (1) the desired dominant closed-loop poles are located at $s_{1,2} = -10 \pm j10$ and (2) a zero steady-state error to a step input.

First of all, we can plot the root locus of the original uncompensated system, which is shown in Fig. 5.16.

It can be found from Fig. 5.16 that in order to make the desired dominant closed-loop poles $s_{1,2} = -10 \pm j10$ on the locus, the original locus should be shifted to the right. This means that a pole should be added and it should be closer to the origin. As for the zero, its purpose is to reduce the settling time and rise time, and its location could be a little far away from the origin.

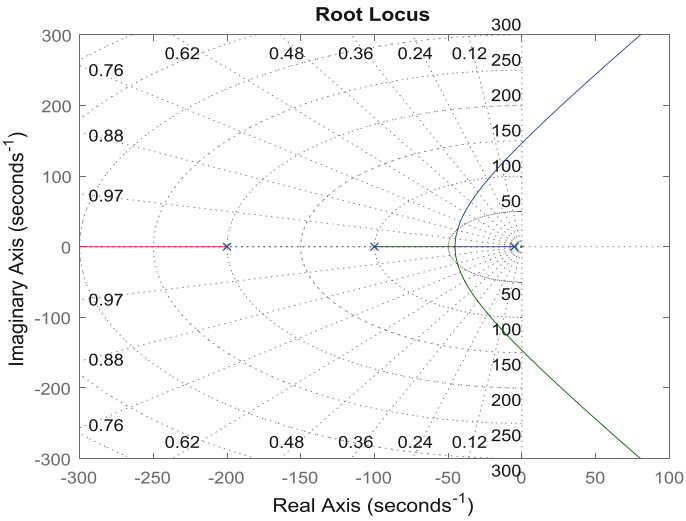


Fig. 5.16 The root locus for the original system

The first specification indicates that the damping ratio ξ is 0.707 or $\xi = \sin^{-1} 45^\circ$ with $\sigma = 10$ and $\omega_d = 10$. Assuming this, we try to use a phase lag compensator with the transfer function as

$$D(s) = K\frac{s+z}{s+p}$$

Based on the analysis above, we can select to add a pole at $s = -0.1$ and determine the zero's location based on the angle condition. Then, we can determine the gain K based on magnitude condition. The complete distributions for all zeros and poles are shown in Fig. 5.17.

The angle at the new added pole $s = -0.1$ (θ_p) and the angle at the new added zero (θ_z) are

$$\theta_p = 180^\circ - \tan^{-1}\left(\frac{10}{9.9}\right) = 180^\circ - 45^\circ = 135^\circ \text{ and } \theta_z = \tan^{-1}\left(\frac{10}{-10-z}\right) \tag{5.28}$$

All other angles are

$$\theta_{p0} = 180^\circ - \tan^{-1}\left(\frac{10}{5}\right) = 180^\circ - 63.43^\circ = 116.57^\circ;$$

$$\theta_{p1} = \tan^{-1}\left(\frac{10}{90}\right) = 6.34^\circ;\ \theta_{p2} = \tan^{-1}\left(\frac{10}{190}\right) = 3^\circ$$

Applying the angle condition, we get

$$\sum_{i=1}^{m} \angle(s+z_i) - \sum_{i=1}^{n} \angle(s+p_i) = \pi$$

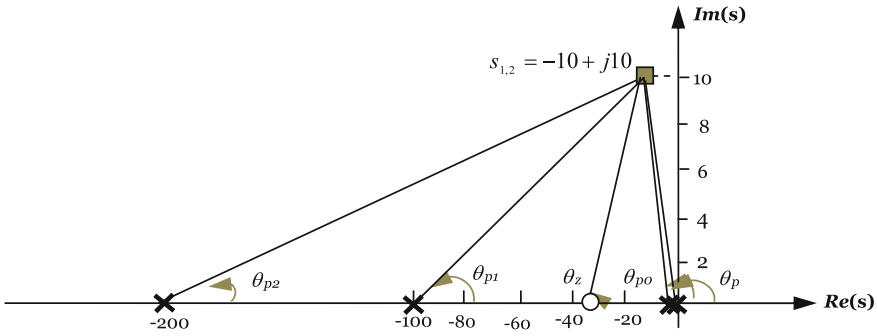


Fig. 5.17 The desired locations of zero and pole for the compensated system

or

$$(\theta_{p0}+\theta_{p1}+\theta_{p2}+\theta_p)-\theta_z=(116.57^\circ+6.34^\circ+3^\circ+135^\circ)-\tan^{-1}\left(\frac{10}{-10-z}\right)$$
$$=260.91^\circ-\tan^{-1}\left(\frac{10}{-10-z}\right)=180^\circ$$

Solving this equation, we have $z = -11.6$. The phase lag compensator has the transfer function as

$$D(s)=K\frac{s+z}{s+p}=K\frac{s+11.6}{s+0.1}$$

Now applying the magnitude condition for the characteristic equation, we can obtain

$$1+D(s)G(s)=1+K\frac{1000|s+11.6|}{|s+0.1||s+5||s+100||s+200|}$$
$$=0 \text{ or } K\frac{1000|s+11.6|}{|s+0.1||s+5||s+100||s+200|}=1 \tag{5.29}$$

Using the positive desired pole $s_1 = -10+j10$ to replace the s in (5.29), we get the magnitude condition as

$$K=\frac{|-10+j10+0.1||-10+j10+5||-10+j10+100||-10+j10+200|}{1000|-10+j10+11.6|}$$
$$=\frac{\sqrt{10^2+9.9^2}*\sqrt{10^2+5^2}*\sqrt{10^2+90^2}*\sqrt{10^2+190^2}}{1000\sqrt{10^2+1.6^2}}$$
$$K=\frac{\sqrt{198\times125\times8200\times36200}}{1000\sqrt{102.56}}=\frac{253414}{1000}=267.6\approx268$$

Finally, we get the phase lag compensator as

$$D(s)=K\frac{s+z}{s+p}=K\frac{s+11.6}{s+0.1}=268\frac{s+11.6}{s+0.1}$$

The root locus for the phase lag compensated system and the step response of this compensated closed-loop control system are shown in Fig. 5.18a, b.

It can be found from Fig. 5.18a that the desired dominant poles $s_{1,2} = -10\pm j10$ are indeed located at the root locus, as shown by two diamonds in the figure.

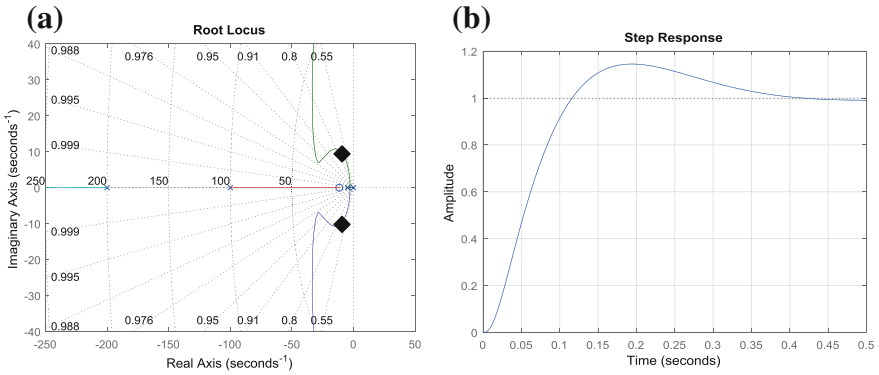


Fig. 5.18 Compensated root locus and step response for the phase lag controller

5.2.6 *Use Computer-Aided Methods to Design Compensators with Root Locus*

A rich set of computer-aided design and analysis functions, methods, and tools are provided by MATLAB to assist users to design, build, analyze, and simulate the closed-loop control systems in easier and faster ways. Summarily, these functions, methods, and tools are included in the Control System Toolbox™ and can be categorized into the following groups:

- A set of MATLAB functions can be directly called and executed by users to perform closed-loop system designs and analyses easily and quickly. These functions can be further divided into different groups based on the design method used:
  - Root locus-related functions,
  - Bode frequency-related functions, and
  - State-space-related functions.
- A set of methods and tools can be invoked from the MATLAB Command window to assist users to design, tune, and simulate closed-loop control systems. These methods and tools include
  - SISOTool(),
  - PIDTool(), and
  - Simulink.

Because some of them are similar with the similar functionalities, in this section, we only pay attention to root locus-related functions and `sisotool()`. As for other functions, `pidtool()` and Simulink, we will discuss them later in this chapter.

Now let's have a closer look at the root locus-related functions.

5.2.6.1 Root Locus Design Functions Used in MATLAB

In Sect. 4.4.5 in Chap. 4, we discussed some popular and important MATLAB functions used in system model developments. In fact, in addition to those root locus-related functions we discussed in that section, there are some other special root locus-related functions used to help users to design and build closed-loop control systems easily and quickly. These functions include

- `sgrid()`
- `rlocfind()`
- `minreal()`

The function `sgrid()` is used to mark regions of the complex plane corresponding to two input parameters, ξ (damping ratio) and ω_n (natural frequency).

The function `rlocfind()` can be used to find root locus gains for a given set of roots. Two syntaxes are most often used; `[K,POLES] = rlocfind(sys)` can be used for interactive gain selection from the root locus plot of a Single-Input-Single-Output (SISO) system `sys` generated by `rlocus()` function. When running, the `rlocfind()` will put up a crosshair cursor in the graphics window which is used to select a pole location on an existing root locus. The root locus gain associated with this point is returned in *K*, and all the system poles for this gain are returned in `POLES`.

`[K,POLES] = rlocfind(sys,P)` takes a vector `P` of desired root locations and computes a root locus gain for each of these locations.

The syntax of the function `minreal()` is `MSYS = minreal(sys)`, which means that this function can create a modified model `MSYS` that is equivalent to the original model `sys` with a reduced transfer function by eliminating all canceling pole/zero pairs.

Now consider a system with the transfer function as

$$G(s) = \frac{1}{s(s+5)} \tag{5.30}$$

We try to use some functions discussed above to design a compensator to get our desired response that are described as below:

(1) ξ = 0.707 and the dominant closed-loop poles are $s_{1,2} = -2 \pm j2$.
(2) The steady-state error is zero.

The damping ratio ξ = 0.707 is equivalent to having θ = 45° (refer to Fig. 5.2), and the dominant closed-loop $s_{1,2} = \sigma \pm j\omega_d = -2 \pm j2$ means that the $\omega_n = \sqrt{8} = 2.8$, and $\omega_d = 2$.

We try to design a PI compensator with the transfer function

$$D(s) = K\frac{1}{s+p} \tag{5.31}$$

Now use the following MATLAB scripts to plot the root locus for the original system:

```
s = tf('s');
sys = 1/(s*(s+5));
rlocus(sys)
grid;
axis([-15 3 -10 10]);
```

The root locus for the original system is shown in Fig. 5.19a.

Now call the function `sgrid()` to get the marked area for the given damping ratio ξ = 0.707 and natural frequency ω_n = 2.8, as shown in Fig. 5.19b.

```
zeta = 0.707;
wn = 2.8;
sgrid(zeta, wn)
```

In Fig. 5.19b, the two dotted lines at about a 45° angle indicate all poles with ξ = 0.7 if they are located on these two lines. If poles are in between these lines, they will have $\xi > 0.7$ and outside of these lines $\xi < 0.7$. The semicircle indicates pole locations with a natural frequency $\omega_n = 2.8$. Inside of the circle, $\omega_n < 2.8$ and outside of the circle $\omega_n > 2.8$.

It can also be found from Fig. 5.19b, in order to make desired poles $s_{1,2} = \sigma \pm j\omega_d = -2 \pm j2$ as closed-loop dominant poles, the root locus should be moved to the right a little bit. Therefore, we try to select our phase lag compensator to have a pole at $s = -10$ to make our compensator as

$$D(s) = K\frac{1}{s+p} = K\frac{1}{s+10} \tag{5.32}$$

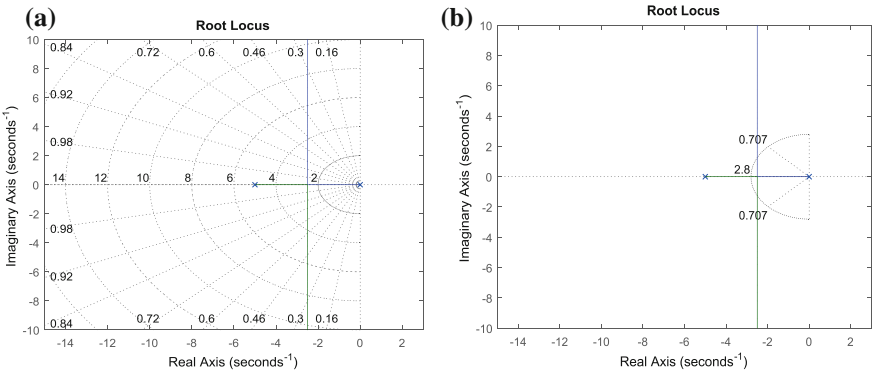


Fig. 5.19 The root locus for the original system and the marked regions for ξ and ω_n

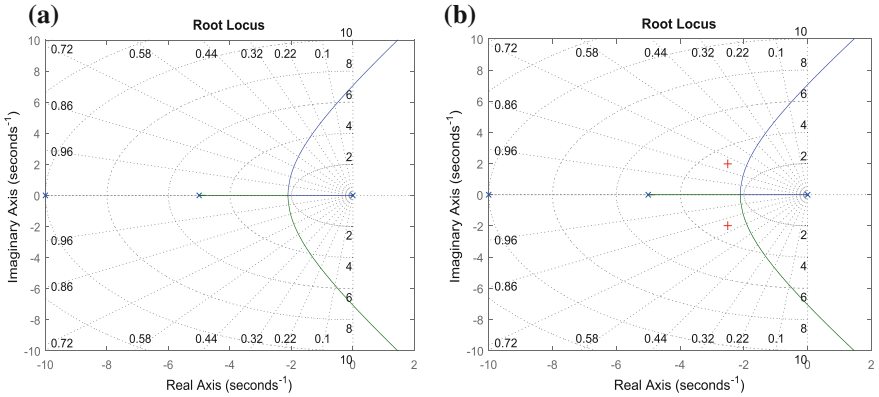


Fig. 5.20 The resulted root locus and the running status of the function rlocfind()

Then, use the scripts below to check resulted root locus, which is shown in Fig. 5.20a.

```
D = 1/(s+10);
rlocus(D*sys);
axis([-10 5 -10 10]);
```

Now use the following scripts to find the gain K on the desired root locations $s_{1,2} = -2 \pm j2$:

```
[k, poles] = rlocfind(sys)
```

When this function is running, you need to move your cursor to the desired pole's location $s_{1,2} = -2 \pm j2$ to click on there to enable the function to find the gain K and related poles. Two plus symbols are displayed on the locus, as shown in Fig. 5.20b. The resulted gain and poles are also displayed in the command window as below:

```
selected_point =  -1.9668 + 1.9814i
k =  10.1149
poles  = -2.5000 + 1.9659i
         -2.5000 - 1.9659i
```

Use the following scripts to plot the step response of this compensated closed-loop system, and it is shown in Fig. 5.21:

```
sysc = feedback(k*D*sys, 1)
step(sysc);
grid;
```

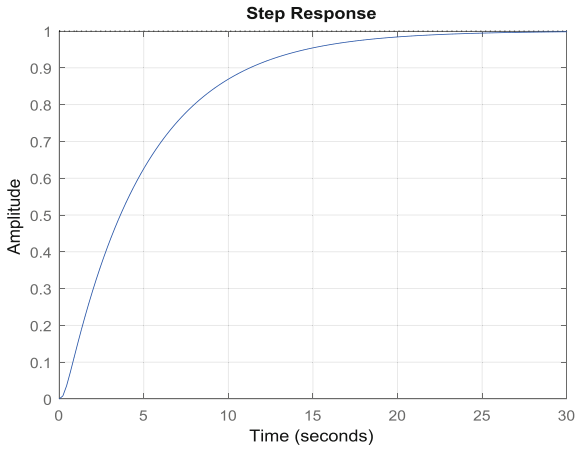


Fig. 5.21 The step response of the compensated closed-loop system

Next, let's take care of using the SISOTool to design compensators with root locus.

5.2.6.2 Use SISOTool to Design Compensators with Root Locus Method

MATLAB Control System Toolbox provides an additional tool called SISOTool to help users easily and quickly design, tune, and build all kinds of compensators based on the given dynamic model of the plants. A Graphic User Interface (GUI) is provided by this tool to enable users to enter and select desired parameters to design and build their desired control systems.

There are different syntaxes when using this tool, and the most popular syntaxes are

(1) `sisotool`
(2) `sisotool(plant)`
(3) `sisotool(plant,comp)`
(4) `sisotool(views)` or `sisotool(views,plant,comp)`
(5) `sisotool(views,plant,comp,sensor,prefilt)`

The first syntax `sisotool` will open a SISO design GUI for interactive compensator design. This GUI allows you to design a SISO compensator using root locus and Bode diagram techniques.

By default, when this SISO Design Tool is invoked in this syntax, the following components are displayed in the GUI window:

- An open-loop root locus and an open-loop Bode diagrams are shown up.
- A compensator, **C**, in the forward path in series with the plant, **G**, is displayed.

- The prefilter, **F**, and the feedback sensor, **H**, are assumed to be unity gains. Once you specify **G** and **H**, they are *fixed* in the feedback structure.

The second syntax `sisotool(plant)` opens the SISO Design Tool; then, users can import their `plant` from the workspace and initialize the plant model **G** to plant. The workspace variable `plant` can be any SISO LTI model created with either functions `ss()`, `tf()`, or `zpk()` as we discussed in Sect. 4.4.5 in Chap. 4.

The syntax `sisotool(plant,comp)` initializes the `plant` model **G** imported from the workspace to plant, and the compensator **C** to comp if that compensator has been built in the Command window or has been loaded into workspace.

The syntax `sisotool(views,plant,comp,sensor,prefilt)` initializes the plant **G** to `plant`, compensator **C** to `comp`, sensor **H** to `sensor`, and the prefilter **F** to `prefilt`. All arguments must be SISO LTI objects. The argument `views` can be any of the following strings or a combination of them:

- `'rlocus'`—Root Locus plot,
- `'bode'`—Bode diagrams of the open-loop response,
- `'nichols'`—Nichols plot, and
- `'filter'`—Bode diagrams of the prefilter **F** and the closed-loop response from the command into **F** to the output of the compensator **G**.

The syntax `sisotool(views)` or `sisotool(views,plant,comp)` specifies the initial configuration of the SISO Design Tool when it is opened.

Now let's use our phase lead compensator designed in Sect. 5.2.4 as an example to illustrate how to use this tool to help us to effectively design and build a closed-loop control system with root locus method. Our plant model and compensator are

$$G(s) = \frac{1}{s(s+1)} \quad \text{and} \quad D(s) = K\frac{s+z}{s+p} \tag{5.33}$$

(1) First of all, we need to create our plant model and enter the following scripts into the Command window:

```
G = tf(1, [1 1 0]);
Sisotool
```

(2) On the opened SISO GUI window, click on **Import** menu item under the **File** menu to open the **System Data** dialog box, as shown in Fig. 5.22a.
(3) Click on the **Browse** button to open the **Model Import** dialog box, as shown in Fig. 5.22b. Make sure that the **Workspace** radio button is selected, and click on the plant **G**, and click on the **Import** button.
(4) Then click on the **Close** button on the **Model Import** dialog and **OK** button on the **System Data** dialog box to close them.

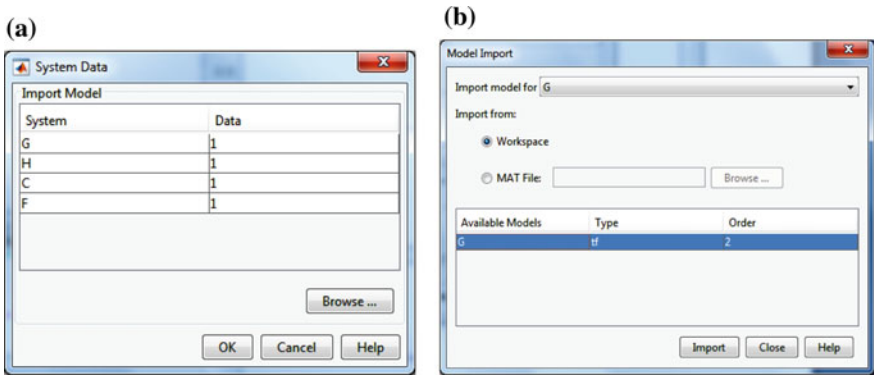


Fig. 5.22 The opened System Data and Model Import dialog boxes. Reprinted with the permission of MathWorks, Inc.

(5) Now the root locus and Bode plot for this original system are displayed on this GUI tool, as shown in Fig. 5.23.

(6) To place a pair of complex conjugate poles $s_{1,2} = \sigma \pm j\omega_d = -2 \pm j2$ to make the damping ratio $\xi = 0.707$, right-click on the root locus plot and select **Design Requirements|New** item to open **New Design Requirement** dialog box. Click the drop-down arrow on the **Design requirement type** combo box, and select **Damping ratio**. The default value is 0.7071 that is our desired value. Keep it and click on the **OK** button.

(7) In a similar way, set the natural frequency ω_n as 2.8. Now both new setup parameters are reflected on the root locus plot, as shown in Fig. 5.24.

(8) Now click on the **Design|Edit Compensator** menu item to open the **Control and Estimation Tools Manager** to design our compensator.

(9) Then right-click on the **Dynamics** box to select **Add Pole/Zero** item, and select **Real Zero** to add our estimated zero −2.14 into that box, as shown in Fig. 5.25.

(10) Return to the SISO Design GUI window, and click on the pole symbol **x** on the toolbar. Drag and place it to the most left location on the real axis in the root locus plot. Your finished window should look like one that is shown in Fig. 5.26.

(11) Now you can click on the Mouse Zoom In icon on the toolbar and drag plot a square area around the closed loops and zero to enlarge this area. Now click on the Hand icon on the toolbar and move the closed pole along the loci to try to place it on the desired locations $s_{1,2} = -2 \pm j2$. You also need to move the zero to make the loci to change its shape and then move poles to get them to the desired locations. Your finished root locus should match one that is shown in Fig. 5.27.

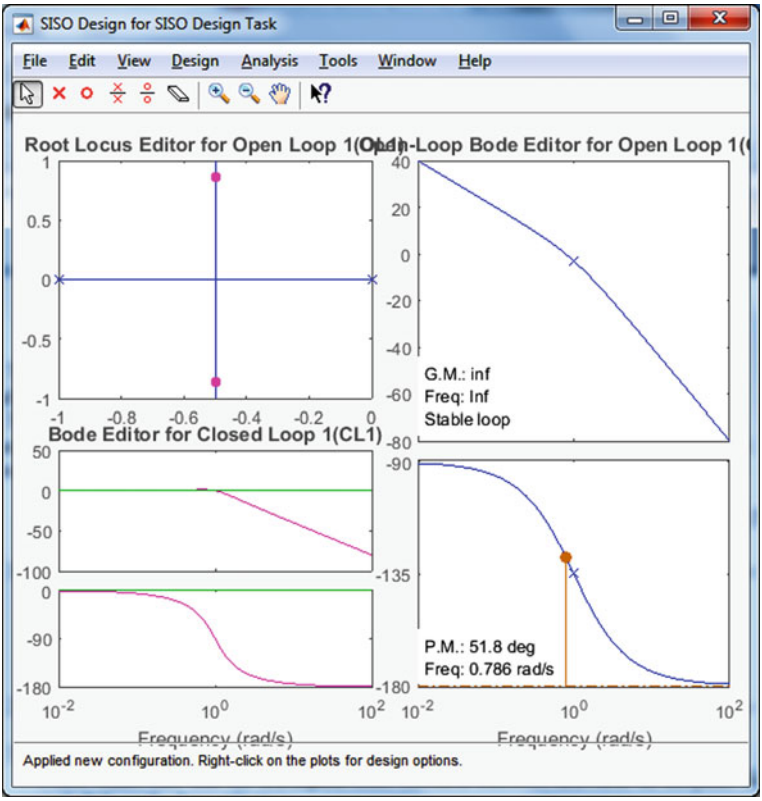


Fig. 5.23 The root locus and Bode response for the original system. Reprinted with the permission of MathWorks, Inc.

(12) To make sure that our closed-loop poles are located in the desired locations on the root locus, select on the **View|Closed-Loop Poles** menu item to open the **Closed-Loop Pole Viewer**, as shown in Fig. 5.28a. You can find that actual values for all parameters include the desired closed-loop poles' locations, compensated pole's location, damping ratio ξ, and natural frequency ω_n, immediately. All of these control parameters' values are equal or very close to our desired values $s_{1,2} = -1.99 \pm j2.02$, damping ratio $\xi = 0.7$, natural frequency $\omega_n = 2.84$, and the compensated pole is located at $s = -4.96$.

(13) Now we need to plot the step response for our closed-loop control system to check and confirm whether our design matches our design specifications. Select the menu item **Analysis|Other Loop Responses** to open the **Control and Estimation Tools Manager**, which is shown in Fig. 5.29. Only check the **Closed Loop r to y** checkbox since we only need this

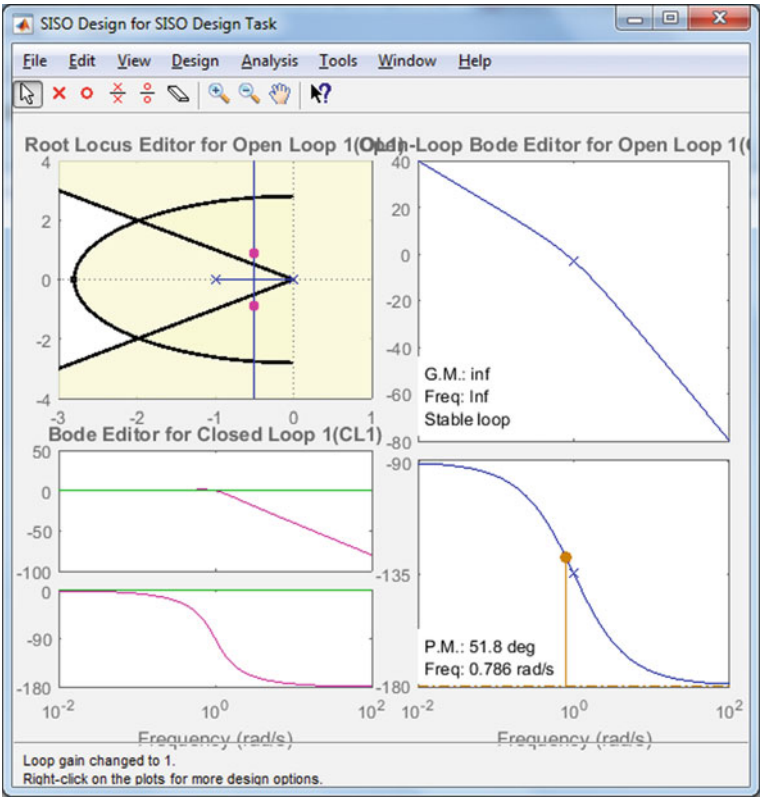


Fig. 5.24 Two setup parameters are reflected on the root locus plot. Reprinted with the permission of MathWorks, Inc.

response. The step response for this compensated closed-loop system is displayed, as shown in Fig. 5.28b.

Our final compensator or phase lead can be expressed as

$$D(s) = K\frac{s+z}{s+p} = 5.03\frac{s+2}{s+4.96}$$

The final gain K can be found from the bottom of the SISOTool GUI window. You can also check your entire design procedure with parameter values by clicking on the **View|Design History** item to open the **Design History** dialog box. Our final design parameters are

```
.....
Loop gain changed to 4.64.
Moved the selected real zero to s = -2.03
Loop gain changed to 5.16.
```

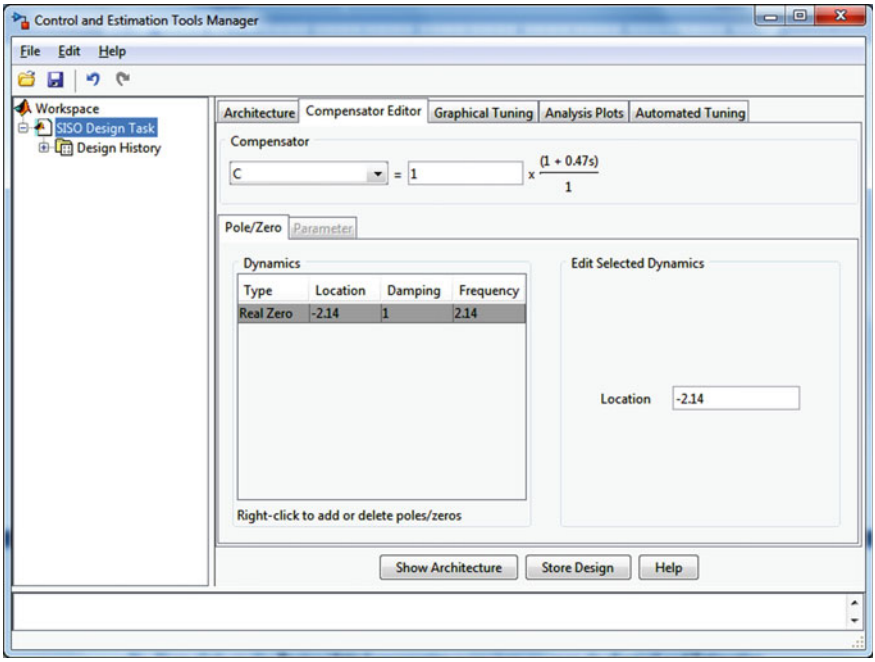


Fig. 5.25 Add a zero into the compensator. Reprinted with the permission of MathWorks, Inc.

```
Moved the selected real zero to s = −1.97
Loop gain changed to 4.87.
Moved the selected real zero to s = −2.01
Loop gain changed to 5.03.
```

You can also save this procedure as a text file by clicking on the **Save to Text File** button.

Finally, you can save your design into the workspace for the further usage. Go to **File|Export** menu item and select the **Compensator C**, and click on the **Export to Workspace** button.

5.3 Design PID Controllers with Frequency Method

Compare with the root locus method, the frequency response method may be more popular and more often used by design engineers to achieve their actual design purpose in most real applications. The main reason for this may be because that the frequency response method does not need too accurate or too many details about the control process or plant; therefore, it can "*tolerate*" some uncertainties existed in

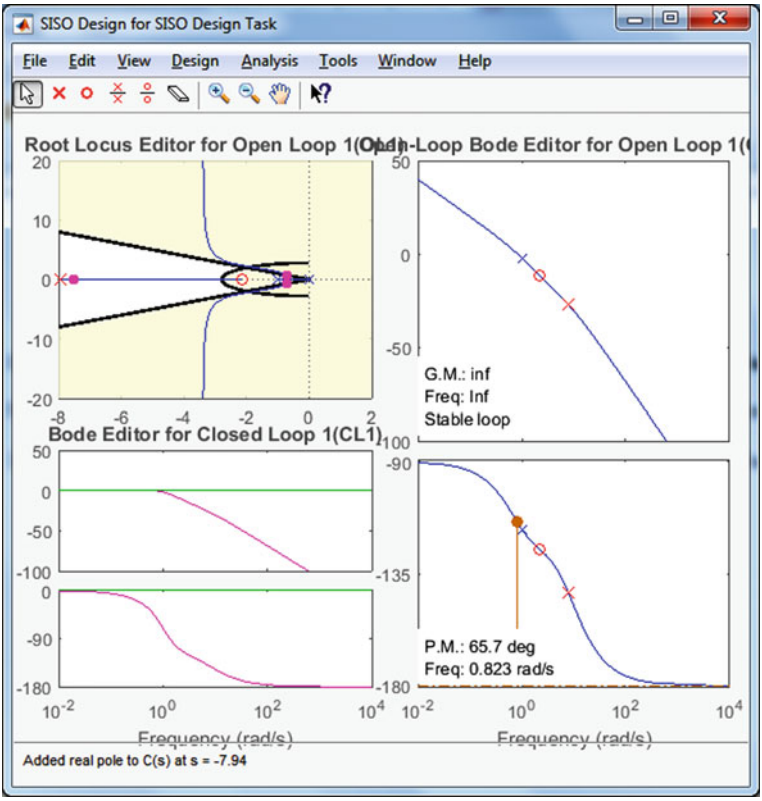


Fig. 5.26 Root locus for the compensator and the plant. Reprinted with the permission of MathWorks, Inc.

the control systems, such as inaccurate dynamic models or environmental disturbances.

Another possible advantage of using the frequency response method is that this method does not need to touch any intermediate processing of parameters. Instead, it only needs to make some measurements for the output amplitude and phase with a sinusoidal signal as the input, which is sufficient to design a controller or compensator for a closed-loop feedback control system. This means that the designers can consider the control target or process as a black box. The only information used for the controller design is the frequency response of the output when a sinusoidal input is applied to that black box.

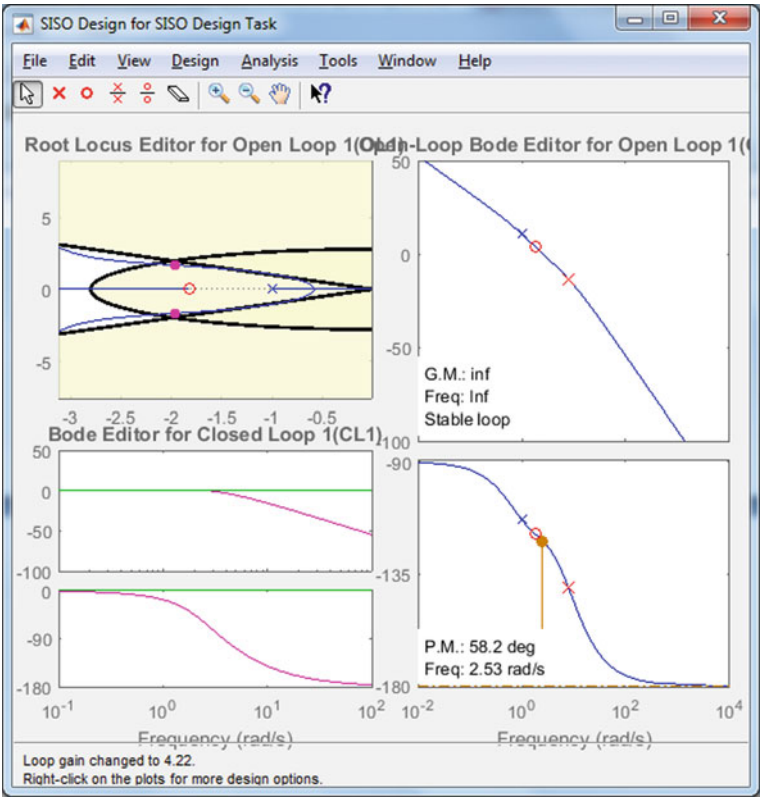


Fig. 5.27 The finished root locus. Reprinted with the permission of MathWorks, Inc.

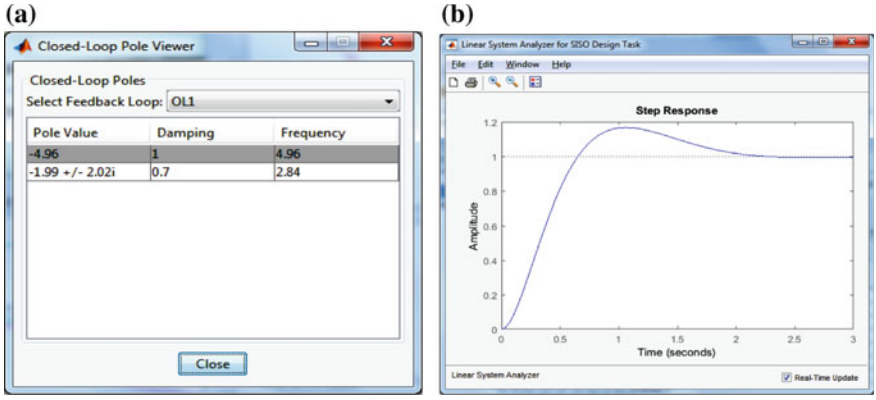


Fig. 5.28 The opened Closed-Loop Pole Viewer and the step response of the system. Reprinted with the permission of MathWorks, Inc.

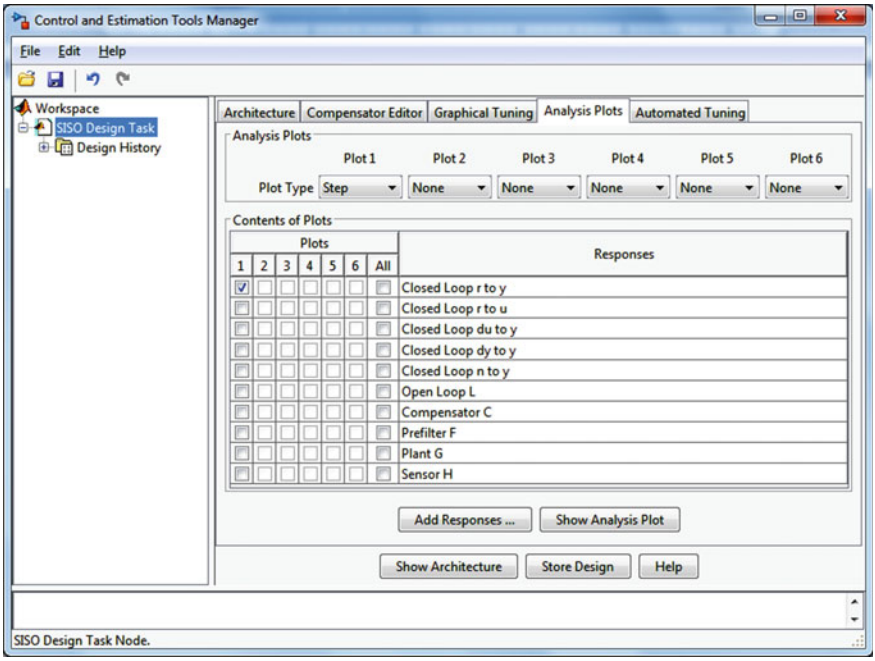


Fig. 5.29 The opened Control and Estimation Tools Manager. Reprinted with the permission of MathWorks, Inc.

5.3.1 *The Major Properties of the Frequency Response Method*

In order to get the frequency response of a control system or a plant $G(s)$, a sinusoidal signal $u(t) = \sin \omega t$ is needed to be applied to the input of that system. The output of a plant $y(t)$, including the amplitude and phase angle, can be written as below:

$$y(t) = A \sin(\omega t + \varphi)$$

This means that the output is also a sinusoidal signal with amplitude A shifted in phase with respect to the input by a phase angle φ.

In fact, the output $y(t)$ can be expressed by a convolution between the input $u(t)$ and the $g(t)$ that is the inverse Laplace transform of the plant $G(s)$ as

$$y(t) = \int_0^t g(\tau)u(t-\tau)\mathrm{d}\tau \tag{5.34}$$

This integration can be further divided into two parts as below:

$$y(t) = \int_0^\infty g(\tau)u(t-\tau)\mathrm{d}\tau - \int_t^\infty g(\tau)u(t-\tau)\mathrm{d}\tau \tag{5.35}$$

When using the sinusoidal signal to replace the input $u(t)$, the steady-state response of the output $y(t)$ can be written as

$$\begin{aligned}\lim_{t\to\infty} y(t) &= \int_0^\infty g(\tau)\sin[\omega(t-\tau)]\mathrm{d}\tau = \mathrm{Im}\left\{\left[\int_0^\infty g(\tau)\mathrm{e}^{-j\omega\tau}\mathrm{d}\tau\right]\mathrm{e}^{j\omega t}\right\} \\ &= \mathrm{Im}\left\{G(j\omega)\mathrm{e}^{j\omega t}\right\}\end{aligned} \tag{5.36}$$

where the second integration item vanishes in the limit $t \to \infty$ if $g(t)$ has certain properties.

In fact, the term in brackets is the definition of the Laplace transform of $g(t)$ at $s = j\omega$. Inserting the definition in the form $G(j\omega) = |G(j\omega)|\mathrm{e}^{j\arg(G(j\omega))}$, one obtains the output signal as

$$\lim_{t\to\infty} y(t) = |G(j\omega)|\sin(\omega t + \arg(G(j\omega))) = A\sin(\omega t + \varphi) \tag{5.37}$$

where $A = |G(j\omega)|$ is the magnitude of the transfer function of the plant $G(s)$ and the additional phase $\varphi = \arg(G(j\omega)) = \tan^{-1}\frac{\mathrm{Im}(G(j\omega))}{\mathrm{Re}(G(j\omega))} = \angle G(j\omega)$ is the phase angle of that transfer function.

It can be concluded that for a second-order LTI system, the steady-state output $y(t)$ is also a sinusoidal signal with the same frequency as the input, and the only differences between the input and the output are the output magnitude and an additional phase angle. These two items can be easily obtained by a product of the original input magnitude with the magnitude of the transfer function of the plant, and a division of the imaginary part by the real part of the transfer function of the plant.

5.3.1.1 Bode Plot Techniques

To describe and measure a system by using sinusoidal signal as input to get the frequency response of the output, a method was developed by H. W. Bode at Bell Laboratories between 1932 and 1942. This technique allows users to plot the frequency response for a system only based on the input and the output, which includes both magnitude and phase responses.

The main idea is to plot magnitude and phase curves using a logarithm scale. This allows the plotting of a high-order plant $G(s)$ by simple graphical addition of

the separate terms. An example Bode frequency plot for a typical second-order system with the transfer function shown below is shown in Fig. 5.30.

$$G(s) = \frac{\omega_n^2}{s^2 + 2\zeta\omega_n s + \omega_n^2} = \frac{1}{(s/\omega_n)^2 + 2\xi(s/\omega_n) + 1} \tag{5.38}$$

Figure 5.30a shows the magnitude frequency response with different damping ratio ξ values, and Fig. 5.30b shows the related phase frequency response via various damping ratios.

Some important specifications in these frequency responses include

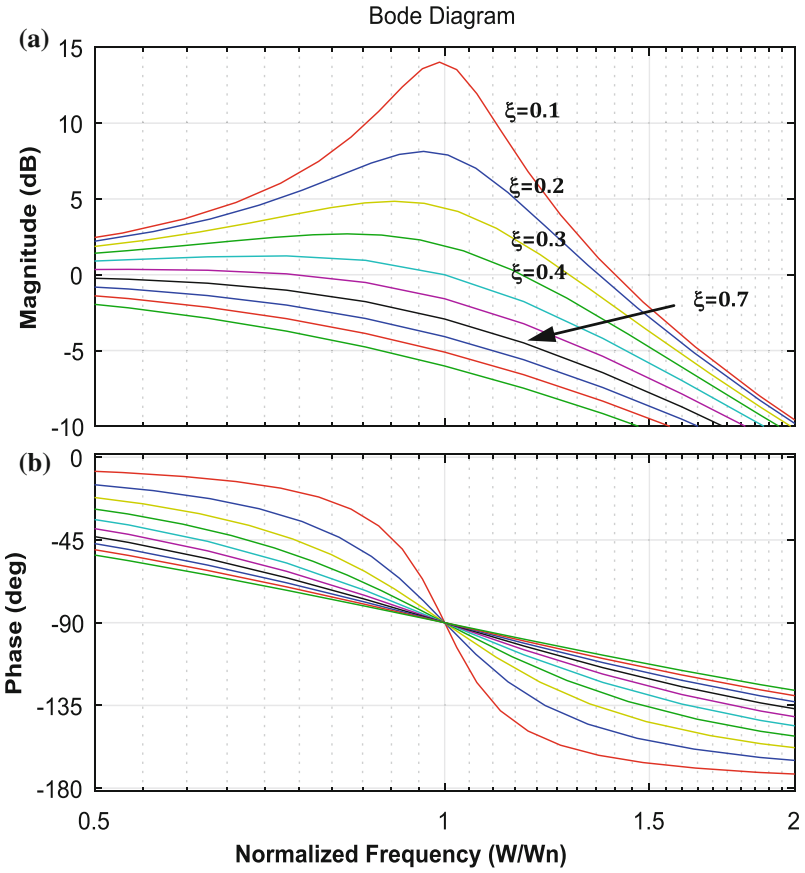


Fig. 5.30 Bode plot for magnitude and phase responses

- Bandwidth ω_{BW}, which is defined as a frequency at which the output y is attenuated to a factor of 0.707 times the input or down 3 dB at the horizontal axis.
- The resonant peak value M_r, which is a measure of the magnitude of the maximum magnitude of the magnitude frequency response.

These specifications are closely related to those of a step response in the time domain. The bandwidth ω_{BW} is exactly a measure of the speed of response and is similar to the time-domain measure of the rise time t_r or the s-plane measure of the natural frequency ω_n. It can be found from Fig. 5.30a that the bandwidth will be equal to the natural frequency at $\xi = 0.707$. For other damping ratios, the bandwidth is approximately equal to the natural frequency with an error typically less than a factor of 2.

The resonant peak M_r is a measure of the damping and its peak value occurred at the natural frequency, which is $1/(2\xi)$ for $\xi < 0.5$.

5.3.1.2 Stability and Margins

Recall that when we discussed the angle and magnitude conditions in Sect. 5.2.1.2 for a unit negative feedback closed-loop system, the closed-loop transfer function can be written as

$$T(s) = \frac{Y(s)}{R(s)} = \frac{KG(s)}{1 + KG(s)}$$

The characteristic equation is $1 + KG(s) = 0$ or $|KG(s)| = 1$ (Magnitude condition) and angle condition $\angle(KG(s)) = \pi = 180^\circ$. Now if we use $j\omega$ to replace s in above conditions, we have

$$|KG(j\omega)| = 1 \quad \text{and} \quad \angle(KG(j\omega)) = \pi = 180^\circ \tag{5.39}$$

In fact, when we do this replacement, the system becomes a neutral-stable system since all poles are now located at the imaginary axis. Thus, a Bode plot of this system is also neutrally stable with the gain value of K such that the closed-loop roots fall on the imaginary axis, and all roots on the root locus meet the conditions on (5.39). That means that the magnitude plot must equal to 1 at the same frequency where the phase plot equals 180°.

For root locus plot, we know that the system is stable if the gain K is less than 1 and becomes unstable if the gain K is greater than 1 when the root is located on the imaginary axis. Therefore, we have the following condition for stability:

$$|KG(j\omega)| < 1 \quad \text{at} \quad \angle(KG(j\omega)) = -180^\circ \tag{5.40}$$

Equation (5.40) is a major criterion to check whether a system is stable or not when using the Bode frequency method to represent frequency response for a system.

Two numeric quantities, Gain Margin (GM) and Phase Margin (PM), can be used to describe how stable for a control system described by the frequency response. In fact, the GM and PM can be used to measure the stability margin of a system based on the criterion in (5.40).

The GM is a factor by which the gain is less than the neutral stability value when the phase is equal to 180°. The PM is the amount by which the phase of $G(j\omega)$ exceeds −180° when the magnitude $|KG(j\omega)| = 1$. These two margins can be used alternately to measure the degree of the stability of a system based on Eq. (5.40).

To illustrate the implementations of these two margins, we use a system with a transfer function

$$G(s) = \frac{K}{s(s+1)^2} \tag{5.41}$$

to show users how to use these criteria in judging the stability degree for a system.

Three different gains, $K = 0.2$, $K = 0.5$, and $K = 5$, are applied to this transfer function of the system $G(s)$. It can be found from the magnitude frequency response shown in Fig. 5.31 that

- When $K = 0.2$, and the magnitude is 0 dB or 1, the crossover frequency of the magnitude response is about 0.2 rad/s. At that point, the $|0.2G(j0.2)| = 1$ or $20\log_{10}|0.2G(j0.2)| = 0$. In order to make this system stable, the phase should be equal to −180° at this $\omega = 0.2$ rad/s. However, it can be found from the phase frequency response shown in Fig. 5.31 that the corresponding phase at that frequency is about −110°. Therefore, the difference is −110° − (−180°) = 70°, which means that the phase margin is 70° since we still have 70° margins to go to get −180° that is the criterion frequency. The system is very stable.
- When $K = 0.5$, and the magnitude is 0 dB or 1, the crossover frequency of the magnitude response is about 0.4 rad/s. At that point, the $|0.5G(j0.4)| = 1$ or $20\log_{10}|0.5G(j0.4)| = 0$. In order to make this system stable, the phase should be equal to −180° at this $\omega = 0.4$ rad/s. However, it can be found from the phase frequency response shown in Fig. 5.31 that the corresponding phase at that frequency is about −135°. Therefore, the difference is −135° − (−180°) = 45°, which means that the phase margin is 45° since we still have 45° margins to go to get −180° that is the criterion frequency. The system is still stable.
- But if we still increase the gain to make $K = 5$, when the magnitude is 0 dB or 1, the crossover frequency of the magnitude response is about 1.5 rad/s. At that point, the $|5G(j1.5)| = 1$ or $20\log_{10}|5G(j1.5)| = 0$. In order to make this system stable, the phase should be equal to −180° at this $\omega = 1.5$ rad/s. However, it can be found from the phase frequency response shown in Fig. 5.31 that the

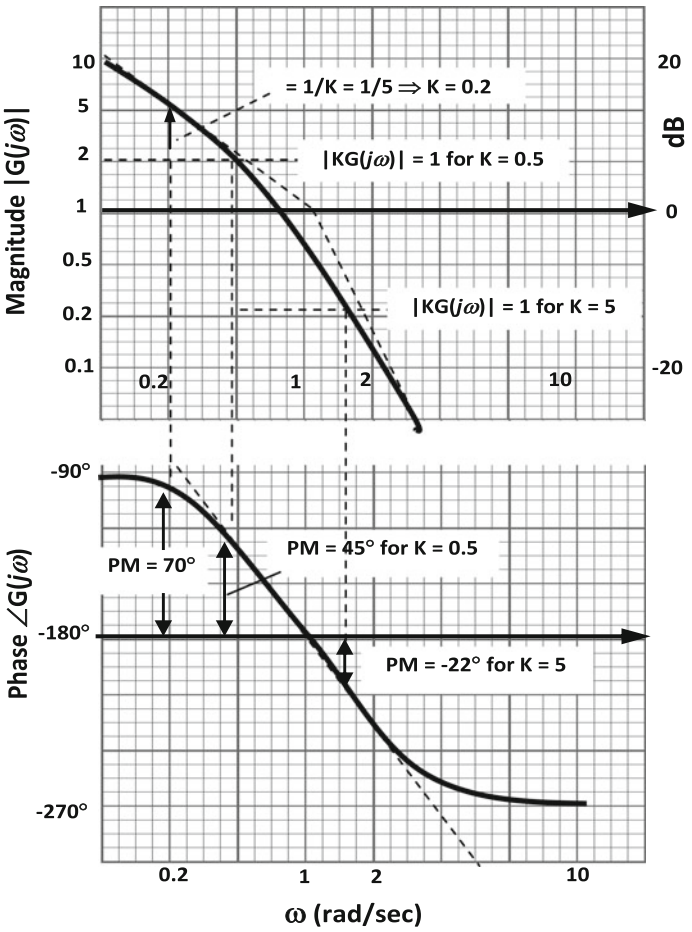


Fig. 5.31 An example of the gain margin and phase margin

corresponding phase at that frequency is about −202°. Therefore, the difference is −202° − (−180°) = −22°, which means that the phase margin is −22° because we go over and exceed −180° by 22°. The system now is not stable.

Generally, the phase margin is closely related to the damping ratio of a system. For a standard second-order system, this relationship can be described as

$$\xi \cong \frac{\text{Phase } M \arg \text{in}}{100} = \frac{\text{PM}}{100} \tag{5.42}$$

It can be found from (5.42) that the optimal PM is about 70°, which is equivalent to setting a damping ratio $\xi = 0.7$. Now let's take a look at how to sketch Bode frequency plot.

5.3.1.3 Sketching Bode Plot

In most applications and designs, it is still helpful to plot the frequency response of a system by hand in order to quickly design some simple systems without needing computer aids although the computer-aided methods can greatly improve the designing efficiency and provide more accurate results.

In the last section when we design compensators by using the root locus method, we always represent our system as the zero-pole-gain format as

$$KG(s) = K\frac{(s+z_1)(s+z_2)\ldots(s+z_m)}{(s+p_1)(s+p_2)\ldots(s+p_n)} \tag{5.43}$$

since this is a convenient way to determine the poles and zeros locations on the root locus to enable us to design our controllers. However, when working with Bode plot techniques to plot the frequency response, it is better to replace the variable s with $j\omega$ and rewrite the transfer function in the Bode format

$$KG(j\omega) = K_0(j\omega)^n\frac{(j\omega\tau_1+1)(j\omega\tau_2+1)\ldots(j\omega\tau_m+1)}{(j\omega\tau_a+1)(j\omega\tau_b+1)\ldots(j\omega\tau_n+1)} \tag{5.44}$$

Generally, there are four different situations or factors that occur in a transfer function to determine the frequency response plot:

1. A constant gain K.
2. Poles or zeros at the origin ($j\omega$).
3. Poles or zeros on the real axis ($j\omega\tau$ + 1).
4. A pair of complex conjugate poles or zeros.

Let's take a look at how to handle these factors to sketch asymptotes for magnitude and phase frequency responses one by one.

- For constant gain K, the logarithmic gain is $20\log K$ = constant in dB and the related phase angle is zero. The gain curve is simply a horizontal line on the Bode plot. However, if the gain is negative, $-K$, the logarithmic gain is still 20 log K, but the negative sign is assigned to the phase angle to make it as $-180°$.
- For poles at the origin ($j\omega$), a pole at the origin has a logarithmic magnitude.

$$20\log\left|\frac{1}{j\omega}\right| = -20\log\omega\,\text{dB},\ \text{where}\ (j\omega)^n = (j\omega)^{-1},\ \text{or}\ n=-1. \tag{5.45}$$

The phase angle $\phi(\omega) = n \times 90° = -1 \times 90° = -90°$.
The slope of the magnitude curve is −20 dB/decade for a pole. In a similar way, if multiple poles are at the origin, the magnitude gain will be

$$20\log\left|\frac{1}{(j\omega)^n}\right| = -20n\log\omega\,\text{dB},\ \text{where } n<0. \tag{5.46}$$

The phase angle is $\phi(\omega) = n \times 90°$. The slope of the magnitude curve is $-20n$ dB/decade.

- For zeros at the origin $(j\omega)$, a zero at the origin has a logarithmic magnitude

$$20\log|j\omega| = +20\log\omega\,\text{dB},\ \text{where } (j\omega)^n = (j\omega)^1,\ \text{or } n = +1. \tag{5.47}$$

The slope of the magnitude curve is +20 dB/decade for a pole. The phase angle $\phi(\omega) = n \times 90° = +1 \times 90° = 90°$.

- For poles on the real axis $(1+j\omega\tau)^{-1}$, a pole on the real axis has a logarithmic magnitude

$$20\log\left|\frac{1}{1+j\omega\tau}\right| = 20\frac{1}{\sqrt{1+(\omega\tau)^2}} = -10\log\left[1+(\omega\tau)^2\right]\text{dB} \tag{5.48}$$

The asymptotic curve for $\omega \ll 1/\tau$ is 20 log (1) = 0 dB, and the asymptotic curve for $\omega \gg 1/\tau$ is -20 log $(\omega\tau)$. The slope is -20 dB/decade. The intersection of the two asymptotes occurs when $\omega = 1/\tau$, which is called **break frequency** or **break point** in the Bode plot. The actual logarithmic gain at this break frequency is -3 dB.

The phase angle at this break point is $\phi(\omega) = -\tan^{-1}(\omega\tau)$.

- For zeros on the real axis $(1+j\omega\tau)$, a zero on the real axis has a logarithmic magnitude

$$20\log|1+j\omega\tau| = 20\log\sqrt{1+(\omega\tau)^2} = +10\log\left[1+(\omega\tau)^2\right]\text{dB} \tag{5.49}$$

The asymptotic curve for $\omega \ll 1/\tau$ is 20 log (1) = 0 dB, and the asymptotic curve for $\omega \gg 1/\tau$ is +20 log $(\omega\tau)$. The slope is +20 dB/decade. The intersection of the two asymptotes occurs when $\omega = 1/\tau$. The actual logarithmic gain at this break frequency is +3 dB. The phase angle at this break point is $\phi(\omega) = +\tan^{-1}(\omega\tau)$.

- For a pair of complex conjugate poles or zeros, $\left[1+(2\xi/\omega_n)j\omega+(j\omega/\omega_n)^2\right]$. This form can be written as $[1+j2\xi u-u^2]^{-1}$ when $u = \omega/\omega_n$. Thus, the logarithmic magnitude is

$$20\log|G(u)| = -10\log\left[\left(1-u^2\right)^2+4\xi^2u^2\right] \tag{5.50}$$

and the phase angle is

$$\phi(u) = -\tan^{-1}\left(\frac{2\xi u}{1-u^2}\right) \tag{5.51}$$

When $u \ll 1$ or $\omega/\omega_n \ll 1$ ($\omega \ll \omega_n$), the magnitude gain is 20 log |G| = −10 log (1) = 0 and the phase angle is about 0°. When $u \gg 1$ or $\omega \gg \omega_n$, the logarithmic magnitude is

$$20\log|G(u)| = -10\log u^4 = -40\log u \tag{5.52}$$

This results in a magnitude curve with a slope of −40 dB/per decade. The phase angle is about −180° when $u \gg 1$ ($\omega \gg \omega_n$). The magnitude asymptotes meet at the 0 dB line when $u = 1$ or $\omega = \omega_n$.

The following sketching steps are developed by Bode in the 1930s but they are still used by most design engineers today because of their popularity. First, let's take a look at the sketching of the magnitude frequency response.

(1) Manipulate the system transfer function into the Bode format as shown in (5.44).
(2) Find the value of the order n for the $K_0(j\omega)^n$ item. Depending on the value of n, start to plot the low-frequency magnitude asymptote through the point K_0 at $\omega = 1$ rad/s with a slope of n or $n \times 20$ dB per decade.
(3) Determine the break points where $\omega = 1/\tau_i$. Continue the magnitude asymptotes by extending the low-frequency asymptote until hitting the first frequency break point, then stepping the slope by ±1 (±20 dB/per decade) or ±2 (±40 dB/per decade), depending on whether the break point is from a first- or second-order term in the numerator or the denominator, and continue going through all break points in ascending order. This means that if a break point belongs to the numerator, for a first-order zero, the slope will be 20 dB/per decade. For a second-order zero, the slope will be 40 dB/per decade. However, if a break point belongs to the denominator, for a first-order pole, the slope will be −20 dB/per decade. For a second-order pole, the slope will be −40 dB/per decade.
(4) Sketch and connect multiple asymptotes by the approximated magnitude curve by increasing from the asymptote with a factor of 3 dB at the first-order numerator break and decreasing it by a factor of −3 dB at the first-order denominator breaks. At the second-order break points, sketch in the resonant peak or valley according to Fig. 5.30a using the equation $|G(j\omega)| = 1/(2\xi)$ at the break point.

Then, let's take care of the sketching of the phase frequency response.

(1) If K_0 in (5.44) is positive, start to plot line (with zero slope) at 0°. If K_0 in (5.44) is negative, start to plot line (with zero slope) at −180°.
(2) Plot the low-frequency asymptote of the phase curve $\phi = n \times 90°$. If $n = -1$, $\phi = -90°$, and if $n = -2$, $\phi = -180°$, and so on.

(3) Sketch for the approximate phase curve by changing the phase gradually over two decades by ±90° or ±180° at each break point in ascending order. For the first-order items in the numerator, the gradual phase change is +90°; for the first-order terms in the denominator, the change is −90°. For the second-order terms, the change is ±180° for the numerator and denominator.
(4) Determine the asymptotes for each phase curve to make their phase changes corresponding to steps in the phase from the approximate curve listed by last step.
(5) Add each phase curve together to get the final complete phase response.

Now let's use an example transfer function to illustrate how to plot a Bode plot with techniques we discussed above.

A system transfer function is given as

$$G(s) = \frac{143232s}{(s+2)(s+20)(s+40)^2} \tag{5.53}$$

(1) Convert this system function to the Bode format as

$$G(j\omega) = \frac{143232}{2 \times 20 \times 40^2} \frac{j\omega}{\left(1+\frac{j\omega}{2}\right)\left(1+\frac{j\omega}{20}\right)\left(1+\frac{j\omega}{40}\right)^2}$$
$$= 2.238 \frac{j\omega}{\left(1+\frac{j\omega}{2}\right)\left(1+\frac{j\omega}{20}\right)\left(1+\frac{j\omega}{40}\right)^2}$$

(2) Based on this equation, we can find that only one zero is on the origin, and four poles are located on the real axis, and they are pole 1 at $\omega = 2$, pole 2 at $\omega = 20$, and poles 3 and 4 are located at $\omega = 40$.
(3) Since the numerator $j\omega = (j\omega)^1$, so $n = 1$, which means that a +20 dB/decade slope should be used to plot a line starting from $\omega = 0$ and continue until hit $\omega = 2$, where the first pole is met and it is a first-order term. Therefore, $\omega = 2$ can be considered as the first break point. The gain at $\omega = 1$ is 20 log (2.238) = 7 dB.
(4) At the first break point $\omega = 2$, a first-order pole is met. Thus, a −20 dB/decade slope should be used to plot a horizontal line (the sum of +20 dB/decade and −20 dB/decade is zero in slope) until hit the second break point $\omega = 20$.
(5) At the second break point $\omega = 20$, another first-order pole is met. Thus, a −20 dB/decade slope is used to plot a −20 dB/decade line since the sum of 0 dB/decade and −20 dB/decade is equal to −20 dB/decade in slope. This plotting is stopped until hitting the third and fourth poles at the third break point $\omega = 40$.
(6) At the third break point $\omega = 40$, a second-order pole term is met. Therefore, a −60 dB/decade should be used to plot −60 dB/decade line until hitting another break point that is located at the infinitive location.

Next, let's take care of the phase angle plot.

(1) Since the gain $K = 2.238$ is positive, so we can plot the phase angle starting from 0°. But the value of $n = 1$, therefore a +90° should be plotted until hit the first break point $\omega = 2$.
(2) At the first break point $\omega = 2$, a first-order pole term is met. A −90° should be added with the previous phase angle +90° to get a 0° phase angle. This should be a gradual process from +90° to 0° until hit the second break point $\omega = 20$.
(3) At the second break point $\omega = 20$, the phase angle should be −90° since a first-order pole term is at that location. From the first break point $\omega = 2$ to the second break point $\omega = 20$, the phase change should be smoothly and gradually from 0° to −90°. This process will be continued until hit the third break point $\omega = 40$.
(4) At the third break point $\omega = 40$, the phase angle should be −270° since a second-order pole term is located at that frequency, which is equivalent to adding a −180° phase angle to the phase plot. Of course, this phase change is also smoothly and gradually.

The completed Bode plot for this system is shown in Fig. 5.32. The real Bode frequency plot by using MATLAB function `bode()` is shown in Fig. 5.33.

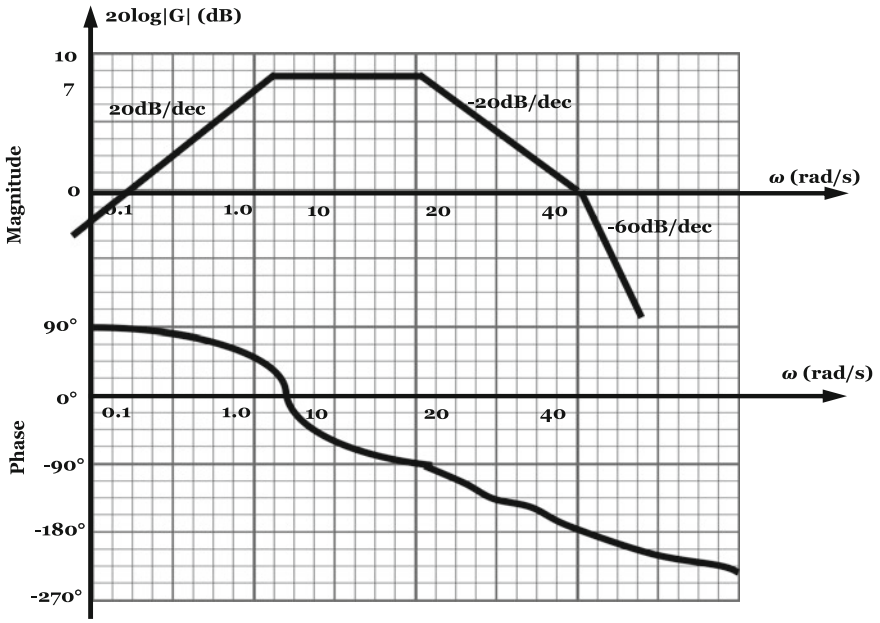


Fig. 5.32 The completed Bode plot for magnitude and phase responses

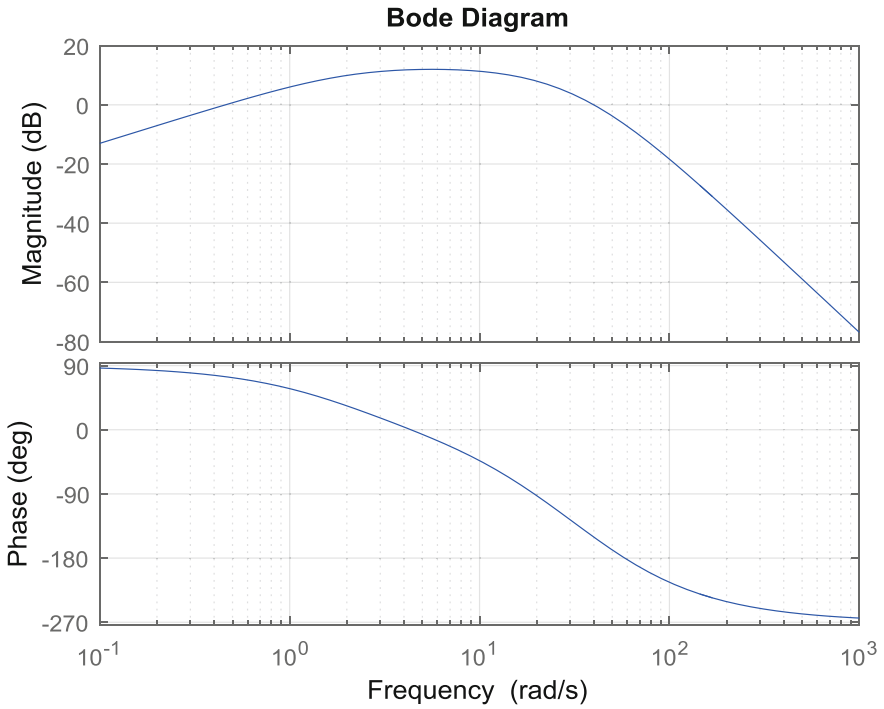


Fig. 5.33 The actual frequency response of the system

5.3.1.4 The Gain and Phase Frequency Relationship

It can be found from the Bode plots that the magnitude frequency response can be expressed as piecewise linear curves with straight lines. These piecewise linear curves allow us to perform some simple computations to be carried out without the need for precise plotting.

The key point when using this technique is that the slopes of the magnitude plot must be integer multiples of 20 dB/decade. In this way, we can find one point on a line from another point. No need to plot the phase curve to compute the phase at specific frequencies.

For example, we consider a segment of a magnitude frequency plot that is shown in Fig. 5.34. The slope of magnitude of this Bode plot must be an integer multiple of 20 dB/decade, as shown in Fig. 5.34, in which the slope is $N \times 20$ dB/decade (N is an integer).

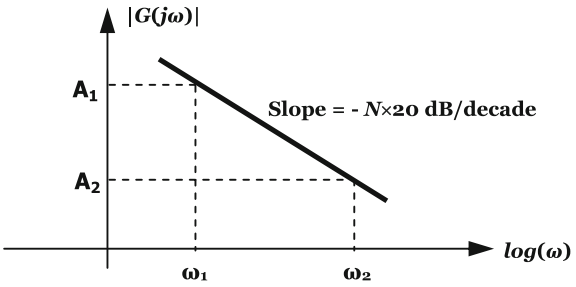


Fig. 5.34 A segment of magnitude frequency response

The following relationships exist based on this segment in Fig. 5.34:

$$\begin{aligned} 20\log(A_2) &= 20\log(A_1) - N * 20 * [\log\omega_2 - \log\omega_1] \\ \log\left(\frac{A_2}{A_1}\right) &= -N\log\left(\frac{\omega_2}{\omega_1}\right) = \log\left(\frac{\omega_1}{\omega_2}\right)^N \\ \left(\frac{A_2}{A_1}\right) &= \left(\frac{\omega_1}{\omega_2}\right)^N \end{aligned} \tag{5.54}$$

Based on (5.54), we can find the value for one point on a Bode plot line using some other points whose values are known. For example, if the values on points A_1, A_2, and ω_1 are known, we can derive the value for the point ω_2. Similarly, if we know the values on the points A_1, ω_1 and ω_2, we can calculate the value on the point A_2.

When using this technique to calculate some points' values, the following conditions must be kept in mind:

(1) The scale used in the Bode plot is logarithmic but the gains used in the formula are not in dB (just unit-less gains).
(2) The formula is only an approximation and it is valid as long as the straight line Bode plot approximation to the actual frequency response curve is valid.

Now let's use an example to illustrate how to use this technique to compute some points' value-based known points.

A magnitude frequency response (Bode plot) is shown in Fig. 5.35. Derive the transfer function of this system based on this Bode plot.

The following parameters can be determined from this plot:

- Two poles at $\omega = 0$ since the initial slope is −40 dB/decade.
- One zero at $\omega = 2$ that is the first break point since from this point the slope becomes −20 dB/decade.
- One pole at $\omega = 33$ that is the second break point since from this point the slope returns back to −40 dB/decade.

However, the value of the gain K at $\omega = 1$ is not given, which is marked as a letter y as an unknown. Therefore, we need to calculate the value on this point based on some other points whose values are known.

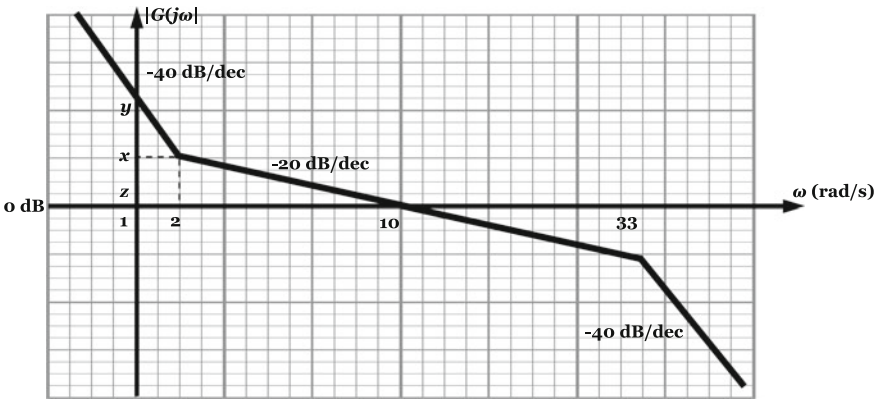


Fig. 5.35 An example magnitude frequency response

Based on these given or known parameters, we can derive the transfer function of the system (in the Bode format):

$$G(j\omega) = \frac{K\left(1+\frac{j\omega}{2}\right)}{(j\omega)^2\left(1+\frac{j\omega}{33}\right)}$$

To calculate the gain value, we need to use Eq. (5.54) as discussed above.

First, we need to get the value at the point x by using (5.54). Here, $A_1 = x$, $A_2 = z = 1$, $\omega_1 = 2$, and $\omega_2 = 10$. The point to be noted is that although the 20 log (A_2) value at z is 0 dB, its real value is 1 (just unit-less gains—condition 1 mentioned above). Based on (5.54), we have

$$\left(\frac{1}{x}\right) = \left(\frac{2}{10}\right)^1; \text{ so } x = 5.$$

The value of N used here is equal to 1 since a −20 dB/decade slope is plotted in this plot.

Now let's calculate the value on point y, which is the gain of this system. Still use (5.54), here $A_1 = y$, $A_2 = x$, $\omega_1 = 1$, and $\omega_2 = 2$, place these values into (5.54), and we have

$$\left(\frac{x}{y}\right) = \left(\frac{1}{2}\right)^2 \text{ or } \left(\frac{5}{y}\right) = \left(\frac{1}{2}\right)^2; \text{ so } y = 20.$$

The final transfer function is ($s = j\omega$)

$$G(s) = \frac{20\left(1+\frac{s}{2}\right)}{(s)^2\left(1+\frac{s}{33}\right)} = 330\frac{(s+2)}{s^2(s+33)}$$

To check whether the system is stable or not, we need to calculate the phase margin at $\omega = 10$ where the gain is 0. The phase at $\omega = 10$ rad/s is

$$\begin{aligned}\angle G(j10) &= \tan^{-1}\left(\frac{10}{2}\right) - 180^\circ - \tan^{-1}\left(\frac{10}{33}\right) \\ &= 78.69^\circ - 180^\circ - 16.86^\circ = -118.17^\circ\end{aligned}$$

The phase margin is $-118.17^\circ - (-180^\circ) = 61.83^\circ$. So the system is stable.

Let's take a look at another example system whose Bode plot is shown in Fig. 5.36. Determine the non-minimum phase transfer function based on this plot.

Based on Fig. 5.36, the following parameters can be derived:

- Two poles at $\omega = 5$ since a −40 dB/decade slope is displayed.
- One zero at $\omega = 20$ since a −20 dB/decade slope is followed.
- The gain is 12 dB that is equivalent to 3.98.

The issue is that where the zero is located? Is it in the LHP or RHP? It can be found from the phase frequency response that this zero must be located at the RHP;

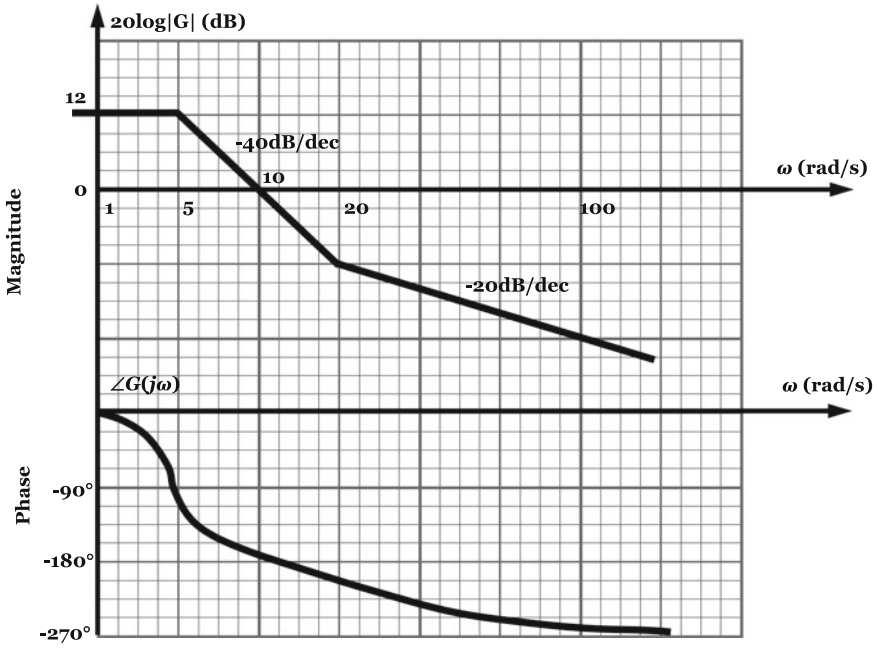


Fig. 5.36 Another example system with Bode plot

otherwise, the phase angle at $\omega = 20$ rad/s should go up. But it is not in Fig. 5.36; therefore, it should be located at RHP.

Based on these parameters and analysis, we have the transfer function as

$$G(s) = \frac{3.98\left(1 - \frac{s}{20}\right)}{\left(1 + \frac{s}{5}\right)^2} = 4.975\frac{(20 - s)}{(s + 5)^2}$$

One of the most important contributions made by Bode is his theorem: "*For any stable minimum phase system, the phase of $G(j\omega)$ is uniquely related to the magnitude of $G(j\omega)$*." The so-called stable minimum phase system means a system with no RHP zeros or poles at all. In other words, a stable system with all its zeros and poles are located in the LHP is called a minimum phase system.

When representing the slope of $|G(j\omega)|$ versus ω in a log–log scale, the following relationship exist:

$$\angle G(j\omega) \cong n \times 90^\circ$$

where n is the slope of $|G(j\omega)|$ in units of decade of magnitude per decade of frequency.

A statement of the Bode gain-phase theorem can be expressed as

$$\angle G(j\omega_0) = \frac{1}{\pi}\int_{-\infty}^{\infty}\left(\frac{\mathrm{d}M}{\mathrm{d}u}\right)W(u)\mathrm{d}u \text{ radians.}$$

M	log magnitude of $\|G(j\omega)\|$,
U	log normalized frequency log (ω/ω_0),
$\mathrm{d}M/\mathrm{d}u$	slope n, and
$W(u)$	weighting function = log $(\coth\|u\|/2)$.

Next, let's take care of the design of compensators by using the Bode frequency response method.

5.3.2 General Considerations of Designing Compensator with Bode Plot

As we discussed in Sect. 5.3.1.2, the relationship between the stability and margin for a system is a key to design any kind of compensator for that system. Regularly, the original system may have no or less phase margin to make system unstable or less stable. The design purpose is to add some compensators to make the closed-loop system not only more stable but also meet the specification requirement. To make this target become to true, the general way is to change or modify

the slope of the magnitude curve around the crossover frequency where the 20 log $G(j\omega) = 0$ to make it equal to −20 dB/decade to get more phase margin.

In fact, the compensation process is the manipulation of gain curve and/or poles positions to improve phase margin to make the system more stable and meet the specifications. The magnitude frequency response is more often used compared to the phase frequency response, and the possible reason may because that it is easier to compute phase margin from the magnitude frequency response plot, which is highly friendly. But the gain margin needs to be found from the phase plot, which is less convenient. Also for any stable loop, the crossover frequency at which the gain reaches 0 dB is lower than frequency at which phase reaches −180°.

The following two criteria must be kept in mind when using the Bode plot method:

(1) Always crossover at a slope of −20 dB/decade.
(2) No other slope such as 0, +20, −40, −60 dB/decade, etc. can yield any decent phase margin.

Before we can design real compensators, let's first take a look at different basic building blocks to familiarize us with their purposes and features. Four typical building blocks are

(1) Simple pole (phase lag compensator),
(2) Simple zero (phase lead compensator),
(3) Integrator, and
(4) Differentiator.

The transfer functions for simple pole and simple zero are

$$G_1(s) = \frac{1}{1+\frac{s}{p}} \quad \text{and} \quad G_2(s) = 1 + \frac{s}{z}$$

The related Bode plots are shown in Fig. 5.37a, b.

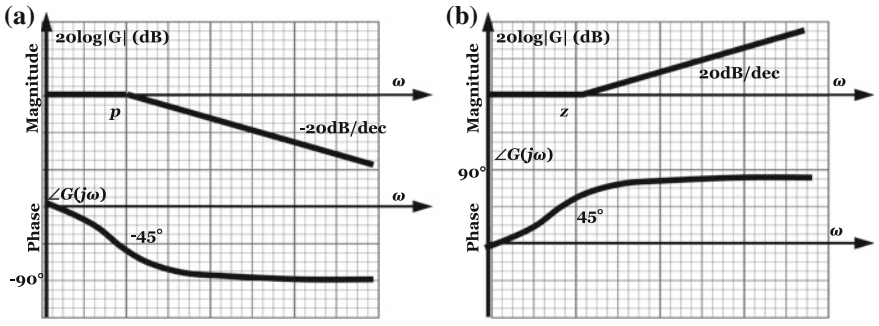


Fig. 5.37 Bode plots for simple pole and simple zero blocks

It can be found from Fig. 5.37 that the crossover frequency for simple pole is $\omega = p$, but this value is $\omega = z$ for the simple zero block. Starting from this frequency, the magnitude frequency response is going down by −20 dB/decade for simple pole block, but it is going up by 20 dB/decade for the simple zero block. Also at this frequency, the phase angle is −45° (simple pole) and 45° (simple zero) blocks.

For the integrator and differentiator block, their Bode plots are shown in Fig. 5.38.

It can be found from Fig. 5.38a that the integrator has a −20 dB/decade slope line and −90° phase angle for all frequencies. But the differentiator has a slope of a 20 dB/decade with 90° phase angle for all frequencies as shown in Fig. 5.38b.

The complete Bode plot for any other complicated system can be built by simply adding these basic blocks together.

Have these basic techniques ready, now let's start to design and build some compensators to improve the performances of desired control systems. First, let's take care of the phase lead compensators.

5.3.3 *Design of a Phase Lead Compensator with Bode Plot Method*

As we discussed in Sect. 5.2.4, a typical phase lead compensator can be expressed as

$$D(s) = K\frac{s+z}{s+p} \tag{5.55}$$

For the phase lead compensator, $|z| < |p|$, but for the phase lag compensator, $|z| > |p|$ should be held. Generally, the phase lead compensation adds a derivative term and increases the bandwidth as well as the speed of the response but decreases the overshoot.

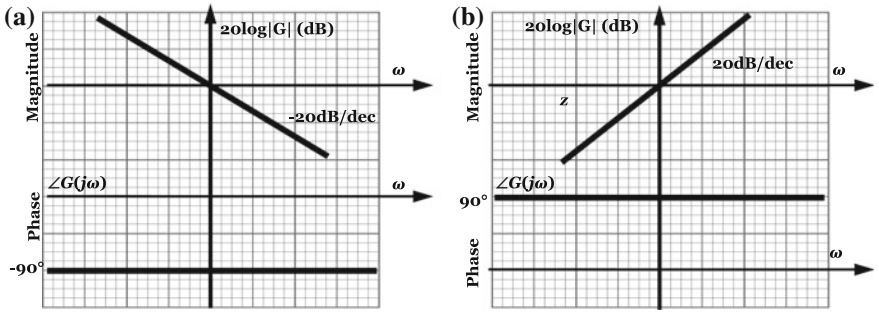


Fig. 5.38 Bode plots for integrator and differentiator blocks

The design of phase lead compensator generally places a zero z at a frequency that is lower than the crossover frequency where the magnitude is 1 and a pole p higher than the crossover frequency. The purpose of adding a zero is to increase the magnitude slope around the crossover frequency to make it going down slower to therefore increase the phase margin. Exactly, the phase lead compensator increases the phase in the interval between these two break points, zero z and pole p. The maximum phase increment and the maximum crossover frequency ω_{cmax} can be determined by

$$\varphi_{\max} = \sin^{-1}\left(\frac{1-\alpha}{1+\alpha}\right) \text{ where } \alpha = \frac{z}{p} \text{ and } \omega_{\max} = \sqrt{|z| * |p|} \tag{5.56}$$

The relationship between the maximum phase angle that can be increased by using the phase lead compensator and the parameter $1/\alpha$ is shown in Fig. 5.39.

Now let's use an example to illustrate how to design a phase lead compensator for a control system by using the Bode plot method.

The system plant has a transfer function as

$$G(s) = \frac{43}{s\left(1+\frac{s}{2}\right)}\bigg|_{s=j\omega} = G(j\omega) = \frac{43}{j\omega\left(1+\frac{j\omega}{2}\right)} \tag{5.57}$$

The specifications include (1) need to track inputs $r(t)$ with less than 1% error up to input frequencies of 1 rad/s, and (2) phase margin should be between 55° and 65°, (3) ω_c = 20 rad/s.

First, let's plot the Bode plot for the original system, which is shown in Fig. 5.40. The phase angle is about −165° at the crossover frequency ω = 6.5 rad/s and the phase margin is about −165° − (−180°) = 15°.

In order to meet the specifications, the steady-state error when an input is a unit ramp or $R(s) = 1/s^2$ should be less than 1%; this can be calculated as

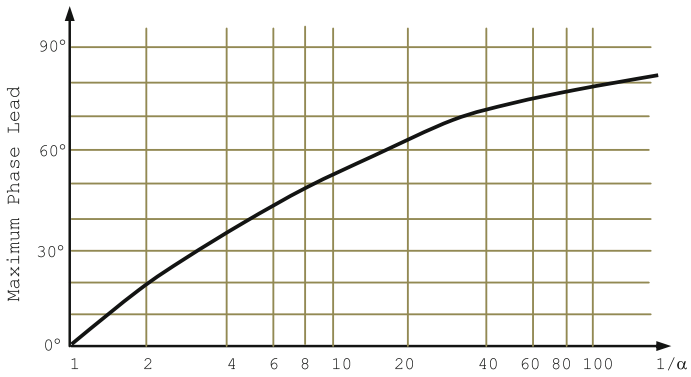


Fig. 5.39 Relationship between the maximum phase angle and 1/α

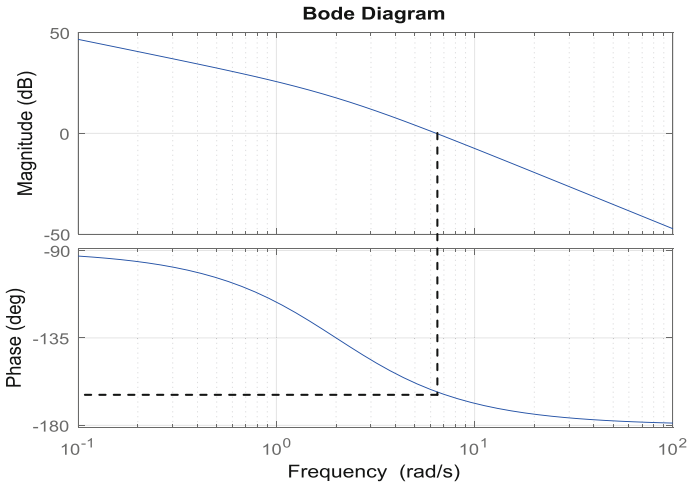


Fig. 5.40 The Bode plot for the original system

$$e_{ss} = \lim_{s\to 0} s\left(\frac{1}{1+D(s)G(s)}\right)R(s) = \lim_{s\to 0} s\left(\frac{1}{1+D(s)G(s)}\right)\frac{1}{s^2}$$
$$= \lim_{s\to 0}\left(\frac{(s+2)}{s(s+2)+86D(s)}\right) = \frac{2}{86D(0)}$$

This means

$$\frac{2}{86D(0)} \le 0.01 \quad \text{or} \quad D(0) \ge 2.325$$

So we can select $K = 2.33$ in Eq. (5.55) as the gain for our phase lead compensator. According to the specifications, the phase margin should be between 55° and 65°; therefore, we can select the desired phase margin as 55°. The original system has already provided a 15° phase margin, and thus we just need an additional 40° phase margin at the crossover frequency $\omega = 6.5$ rad/s. However, the crossover frequency would be increased if we maintain the same low-frequency gain but add a compensator zero, and this may result in a phase margin that more than 40° is obtained. Therefore, we should design our phase lead compensator to limit the maximum phase lead to 60° with a new crossover frequency $\omega_c = 20$ rad/s. It can be found from Fig. 5.39 that $1/\alpha = 12$ will meet this requirement and this means $z = 0.08p$.

Also from Eq. (5.56), the required new crossover frequency $\omega_c = 20$ rad/s; based on that equation, we have $\omega_c^2 = z * p = 400$, and solve this equation with $z = 0.08p$, $z = 5.7$, and $p = 70$. Placing these values into the compensator, we have

$$D(s) = 2.33\frac{1+(s/5.7)}{1+(s/70)}$$

The Bode plot for the phase lead compensated system is shown in Fig. 5.41a, and the step response of this closed-loop system is shown in Fig. 5.41b. It can be found from Fig. 5.41a that the phase in the new crossover frequency ω = 25 rad/s is about −120°; thus, the phase margin is about 60°. From the step response, it can be found that the overshoot is about 18% with a steady-state error less than 1%.

Summarily, the design procedure for a phase lead compensator can be performed by the following two steps:

(1) Identify the low-frequency gain K to meet the requirement of steady-state error.
(2) Select the combination of lead ratio $1/\alpha$ and zero location based on Eq. (5.56) to get an acceptable PM at the desired new crossover frequency.

Next, let's take a look at the design for the phase lag compensator.

5.3.4 *Design of a Phase Lag Compensator with Bode Plot Method*

Basically, by using a phase lag compensator, the bandwidth of the magnitude response in the Bode plot will be decreased since a pole is applied at very low frequency. Because the crossover frequency is reduced, the phase margin is relatively increased. The advantage of using a phase lag compensator is that the steady error can be reduced, but the cost is that the phase is decreased below the breakpoint.

Let's use the same example we used in the last section to illustrate how to use Bode plot method to design a phase lag compensator with the same specifications.

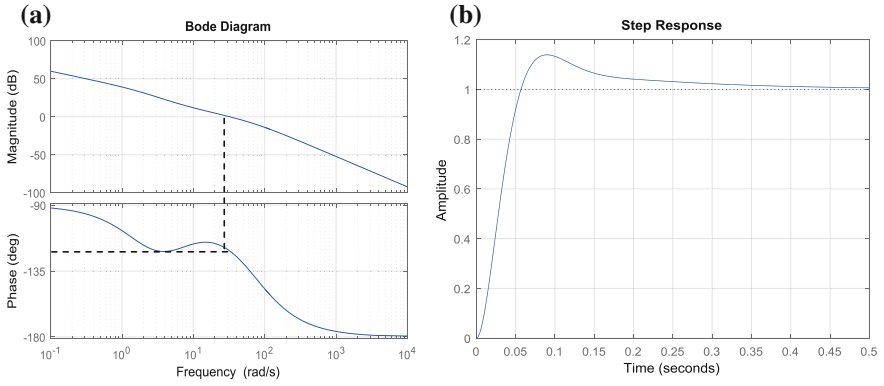


Fig. 5.41 The compensated Bode plot and step response

The low-frequency gain is still $K = 2.33$ as we did in the last section. To prevent detrimental effects from the compensation phase lag, the pole and zero of the compensator need to be much lower than the new crossover frequency. In order to get a 65° phase margin, $1/\alpha$ equals to 20 based on Fig. 5.39 ($\alpha = p/z$ for phase lag). However, because this is a phase lag compensator that may reduce the phase, we prefer to select a little big phase margin such as 60 for $1/\alpha$. If the pole is selected at very low frequency such as $\omega = 0.001$ rad/s, then the zero should be at $\omega = 0.06$ rad/s. This results in the phase lag compensator with the following transfer function:

$$D(s) = 2.33\frac{1 + (s/0.06)}{1 + (s/0.001)}$$

The compensated Bode plot frequency response is shown in Fig. 5.42. It can be found from this figure that the phase margin at the new crossover frequency is about 65°.

A special feature in design a phase lag compensator is that both compensated pole and zero must be located on the left of the zero and pole of the original system. Otherwise, the designed compensator cannot play its role effectively.

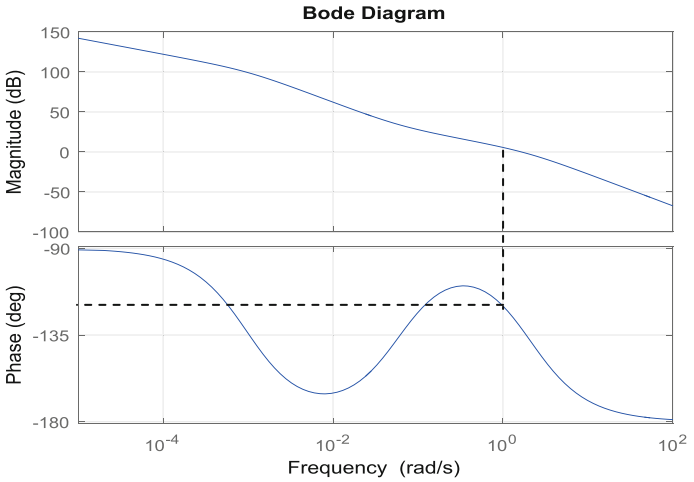


Fig. 5.42 The compensated Bode plot with a phase lag

5.3.5 Design of a PID Controller with Bode Plot Method

By combining a phase lead and a phase lag compensator, a PID controller can be built. But regularly we prefer to design a PID controller with a direct way. A standard PID controller has the following transfer function:

$$D(s) = K_P + K_D s + K_I \frac{1}{s} = \frac{K}{s}\left[(1 + T_D s)\left(s + \frac{1}{T_I}\right)\right] = \frac{K}{s}\left[\left(1 + \frac{s}{\omega_D}\right)(s + \omega_I)\right] \tag{5.58}$$

where $\omega_D = 1/T_D$ and $\omega_I = 1/T_I$; they are compensated zero and pole's locations on the Bode plot. The frequency response of the PID controller or compensator is shown in Fig. 5.43. Both break frequencies or break points, ω_D and ω_I, are also shown in this figure. The T_D is called the derivative time constant and the T_I is called the integration time constant, respectively.

The advantages of using a PID compensator over other compensators are

- It can meet multiple design requirements, such as overshoot, steady-state error, and rise and settling times, simultaneously in one compensator.
- It is more powerful compared with any other compensators since it provides three control gains to enable users to have more choices and more controllability to the control target.
- It provides more stable and noise-rejection ability because of its wide range on the phase margin and integration factor.

Depending on the different control objectives, sometimes the position of pole and zero shown in Fig. 5.43 may be exchanged to provide more phase lead function to improve the phase margin requirement.

Let's use an example to illustrate how to use the Bode plot or frequency response method to design a PID controller for a target plant. The transfer function for a plant is

$$G(s) = \frac{s + 10}{s^2}$$

The design specifications including the steady-state error are less than 1%, and the new approximate crossover frequency is about 0.35 rad/s with good damping ratio.

To find an appropriate PID controller, regularly we need to select gain K to meet the steady-state error requirement. Since a −40 dB/decade slope is shown up because of the pole $s = 0$ at the plant transfer function, we need to place a zero that is closed to the new crossover frequency $\omega = 0.35$ rad/s to make this slope to −20 dB/decade to increase the phase margin. Generally, we can pick ω_D about a factor of 4 lower than the new crossover frequency to make it. Also, we can pick ω_I

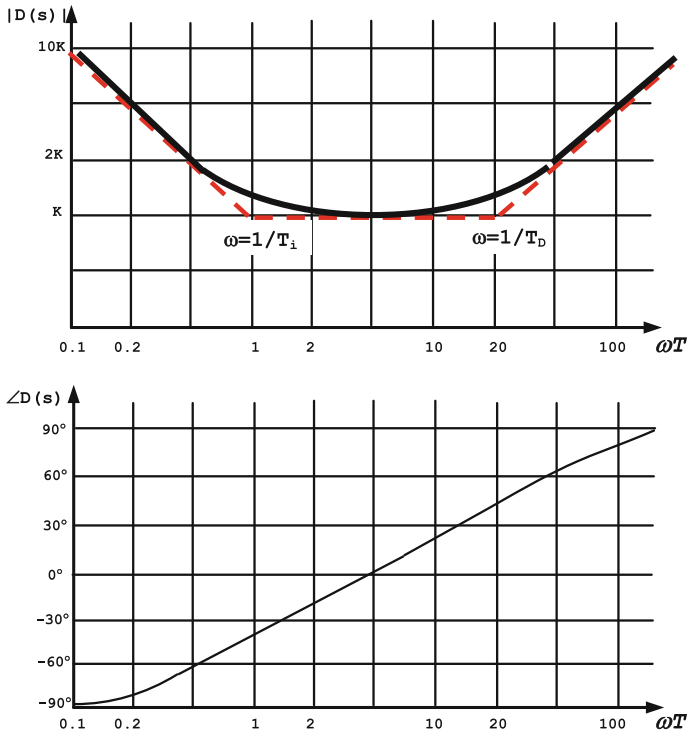


Fig. 5.43 The frequency response of a PID compensator

about a factor of 4 higher than the new crossover frequency to make steady-state error small.

Based on these design ideas, we have the following calculations:

(1) Select $\omega_D = \frac{\omega_C}{4} = \frac{0.35}{4} = 0.0875$ rad/s.
(2) Select $\omega_I = 4 \times \omega_C = 4 \times 0.35 = 1.4$ rad/s.
(3) Select gain $K = \frac{1}{|D(j\omega)G(j\omega)|}\Big|_{\omega=0.2} = \frac{1}{100} = 0.01$.

The PID controller should have a transfer function as

$$D(s) = \frac{0.01}{s}\left[\left(1 + \frac{s}{0.0875}\right)(s + 1.4)\right] \tag{5.59}$$

The compensated frequency response is shown in Fig. 5.44. It can be found that the new crossover frequency is about 1.6 rad/s with a phase margin around 54°.

5.3.6 *Use Computer-Aided Methods to Design Compensators with Bode Plot*

Although the performances of a control system can be significantly improved by designing of different compensators by hands, it is more powerful and convenient to use computer-aided method to design various compensators to do the similar design jobs.

As we discussed in Sect. 5.2.6, MATLAB provides a set of tools to support users to design and build various compensators to improve the control performances for different control systems. In this section, we will concentrate on the following tools:

- SISOTool(),
- PIDTool(), and
- Simulink.

First, let's pay attention to the SISOTool and see how the Bode plot can be used to help us to design and build compensators to improve the control performances.

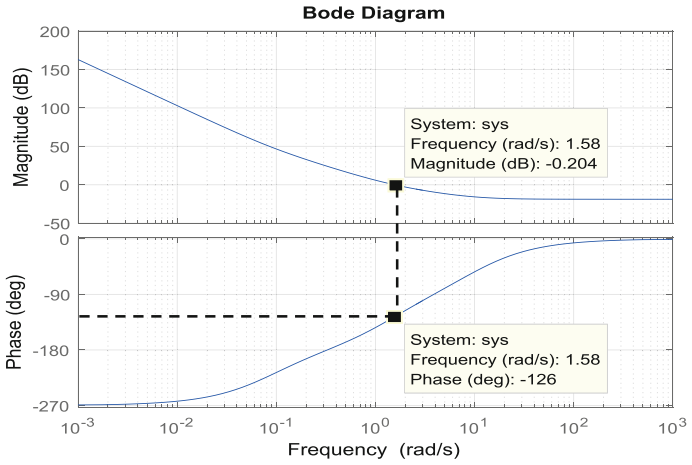


Fig. 5.44 The compensated frequency response with a PID controller

5.3.6.1 Use SISOTool to Design Compensators with Bode Plot

As we discussed in Sect. 5.2.6, SISOTool provides two major tools, root locus and Bode plot, to help users to design all different kinds of compensators for the selected plant or target. A detailed introduction and discussion about this tool has been provided in Sect. 5.2.6.2. Refer to that section to get more details about this tool.

Now let's use our phase lead compensator designed in Sect. 5.2.4 as an example to illustrate how to use this tool to help us effectively design and build a closed-loop control system with Bode plot method. Our plant model and compensator are

$$G(s) = \frac{1}{s(s+1)} \quad \text{and} \quad D(s) = K\frac{s+z}{s+p} \tag{5.60}$$

(1) First of all, we need to create our plant model and enter the following scripts into the Command window:

```
G = tf(1, [1 1 0]);
sisotool
```

(2) On the opened SISO GUI window, click on **Import** menu item under the **File** menu to open the **System Data** dialog box.
(3) Click on the **Browse** button to open the **Model Import** dialog box. Make sure that the **Workspace** radio button is selected, and click on the plant **G**, and click on the **Import** button.
(4) Then click on the **Close** button on the **Model Import** dialog and **OK** button on the **System Data** dialog box to close them.
(5) Now the root locus and Bode plot for this original system are displayed on this GUI tool, as shown in Fig. 5.45. The crossover frequency is 0.785 rad/s, and the PM is 51.8°.
(6) To place a pair of complex conjugate poles $s_{1,2} = \sigma \pm j\omega_d = -2 \pm j2$ to make the damping ratio $\xi = 0.707$, right-click on the Bode plot and select **Design Requirements|New** item to open **New Design Requirement** dialog box. Click the drop-down arrow on the **Design requirement type** combo box, and select **Gain and Phase margins**. Uncheck the **Gain margin** checkbox since we do not worry about it, enter **70** into the **Phase margin** box, and click **OK**.

(7) Now click on the **Design|Edit Compensator** menu item to open the **Control and Estimation Tools Manager** to design our compensator.
(8) Then right-click on the **Dynamics** box to select **Add Pole/Zero** item, then select and click on **Real Zero** to add our estimated zero −2.14 into that box, as shown in Fig. 5.46.
(9) Return to the SISO Design GUI window, and click on the pole symbol **x** on the toolbar. Drag and place it to the most right location on the magnitude

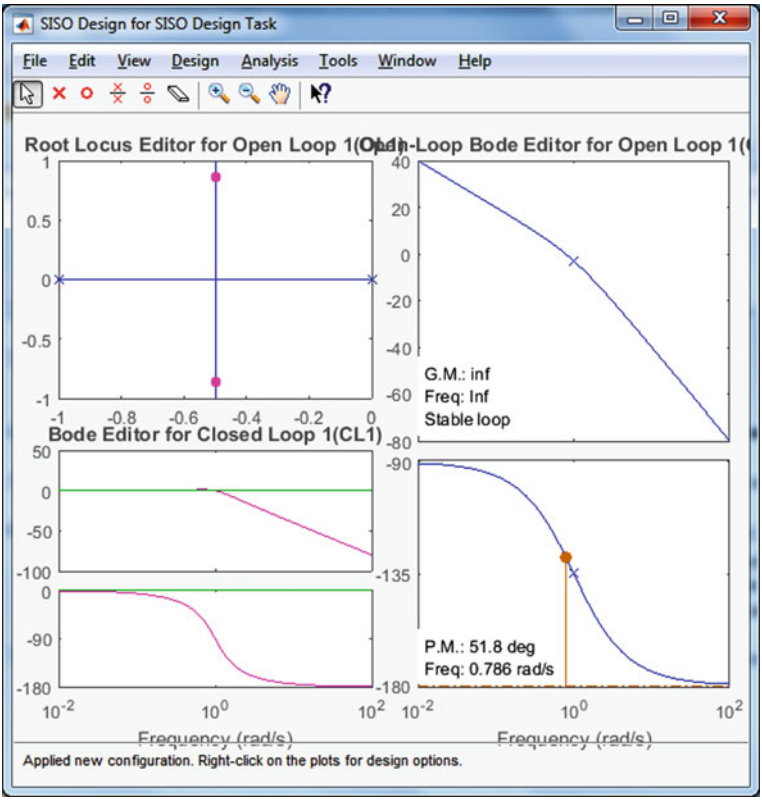


Fig. 5.45 The Bode plot for the original plant. Reprinted with the permission of MathWorks, Inc.

response trajectory in the Bode plot. Your finished window should look like one that is shown in Fig. 5.47.

(10) Now you can find from Fig. 5.47 that the PM is now increased to 70.7° and the crossover frequency is also increased from 0.785 to 0.826 rad/s, which is a very small increment and can be acceptable.

(11) To check whether our closed-loop poles are located in the desired locations on the Bode plot, select on the **View|Closed-Loop Poles** menu item to open the **Closed-Loop Pole Viewer**, as shown in Fig. 5.48. You can find that actual values for all parameters, including the desired closed-loop poles' locations, compensated pole's location, damping ratio ξ and frequencies, immediately. These control parameters' values are equal or very close to our desired values such as damping ratio $\xi = 0.728$.

(12) Now we need to plot the step and closed-loop Bode response for our closed-loop control system to check and confirm whether our design matches our design specifications. Select the menu item **Analysis|Other Loop Responses** to open the **Control and Estimation Tools Manager**, which

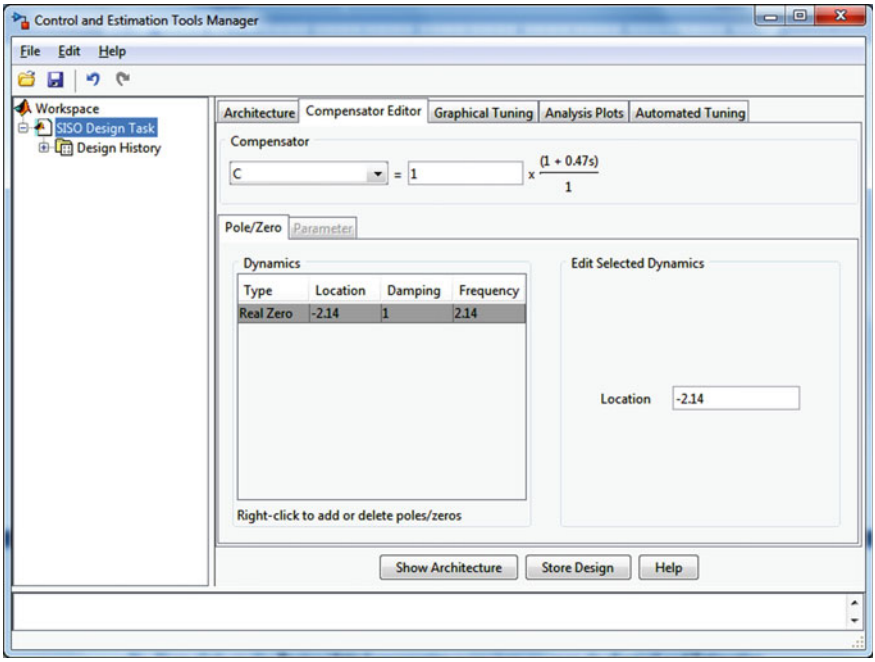


Fig. 5.46 Add a zero to the compensator. Reprinted with the permission of MathWorks, Inc.

is shown in Fig. 5.49. Only select the **Closed Loop r to y** line and check 1 and 2 checkboxes since we need both step response and the closed-loop Bode plot response. The step and closed-loop Bode responses for this compensated system are displayed, as shown in Fig. 5.50.

Our final compensator or phase lead can be expressed as

$$D(s) = K\frac{s+z}{s+p} = \frac{s+2.14}{s+55}$$

The final gain K (=1) can be found from the **Compensator Editor** tab in the **Control and Estimation Tools Manager** dialog box. You can also check your entire design procedure with parameter values by clicking on the **View|Design History** item to open the **Design History** dialog box. Our final design parameters are

```
22-Nov-2017: Starting SISO Tool for system: SISO Design Task
Changed control system configuration.
Added Poles/Zeros.
Edited zero.
Added real pole to C(s) at s = -55
```

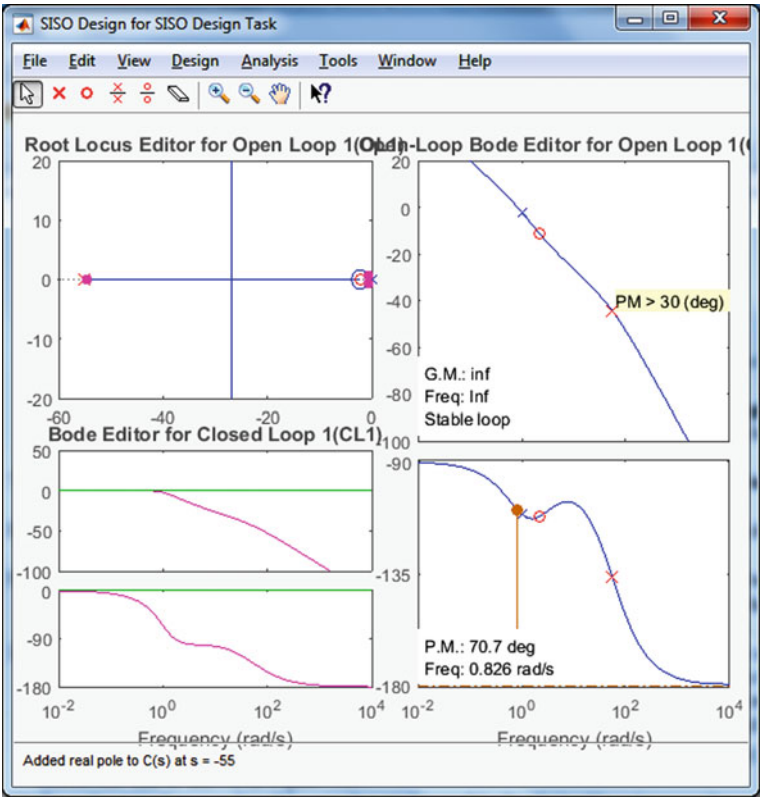


Fig. 5.47 Bode plot for the compensator and the plant. Reprinted with the permission of MathWorks, Inc.

You can save this procedure as a text file by clicking on the **Save to Text File** button.

Finally, you can save your design into the workspace for the further usage. Go to **File|Export** menu item and select the **Compensator C**, and click on the **Export to Workspace** button.

5.3.6.2 Use PIDTool or PIDTuner to Design and Tune Compensators

The PIDTool is a powerful tool provided by MATLAB Control System Toolbox to help users to design, build and tune desired PID controllers for the selected plant system. The PIDTool is also called PIDTuner in the new MATLAB version with similar control functions.

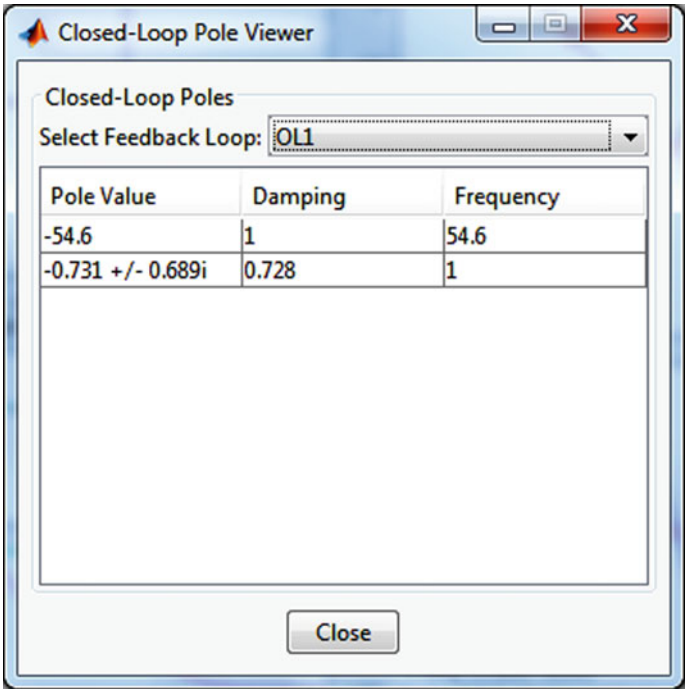


Fig. 5.48 The opened Closed-Loop Pole Viewer. Reprinted with the permission of MathWorks, Inc.

Three functions are involved in this tool, and they are

(1) `pidtune()`
(2) `pidtuneOptions()`
(3) `pidTuner()`

The first two functions can be called from the command line and perform related functions, but the third function is exactly a GUI tool and provides a graphical interface to allow users to design and tune their PID controllers in an easy way.

For the first function, `pidtune()`, it tunes the parameters of the PID controller `C` to balance performance (response time) and robustness (stability margins). Different syntaxes are available:

```
C = pidtune(sys,type)
C = pidtune(sys,C0)
C = pidtune(sys,type,wc)
C = pidtune(sys,C0,wc)
C = pidtune(sys,...,opts)
[C,info] = pidtune(...)
```

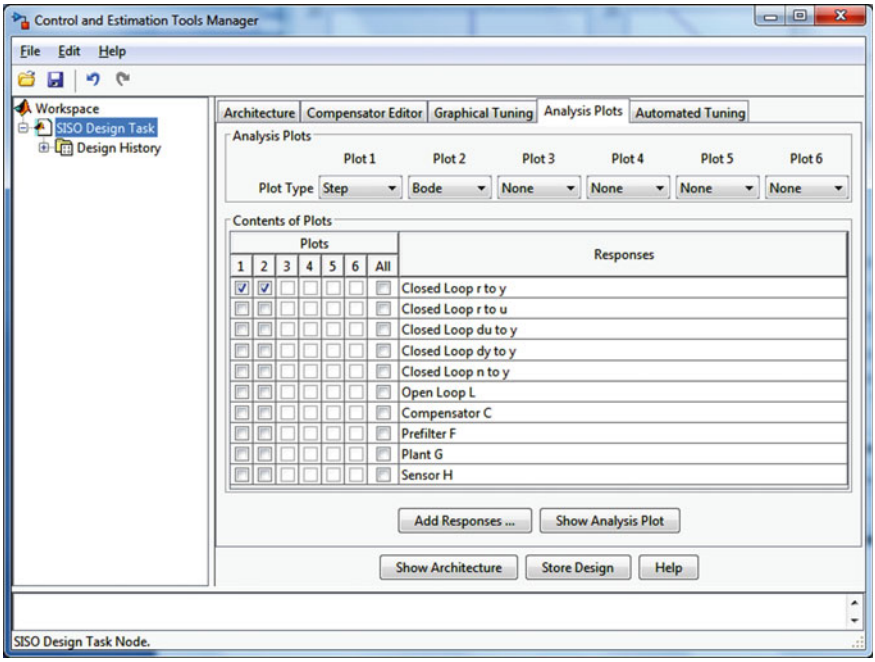


Fig. 5.49 The opened Control and Estimation Tools Manager. Reprinted with the permission of MathWorks, Inc.

The first syntax `C = pidtune(sys,type)` designs a PID controller of type `type` for the plant `sys`. If `type` specifies a one-degree-of-freedom (1-DOF) PID controller, then the controller is designed for the unit negative feedback loop.

The function `C = pidtune(sys,C0)` designs a controller of the same type and form as the controller `C0`. If `sys` and `C0` are discrete-time models, `C` has the same discrete integrator formulas as `C0`. The functions `C = pidtune(sys,type,wc)` and `C = pidtune(sys,C0,wc)` specify a target value `wc` for the first 0 dB gain crossover frequency of the open-loop response.

The function `C = pidtune(sys,...,opts)` uses additional tuning options, such as the target phase margin and noise-rejection level. One should use the function `pidtuneOptions()` to specify those options before calling this function.

The final syntax `[C,info] = pidtune(...)` returns the data structure `info`, which contains information about closed-loop stability, the selected open-loop gain crossover frequency, and the actual phase margin.

The second function `pidtuneOptions()` is used to define options for `pidtune()` function. Two syntaxes are provided

```
opt = pidtuneOptions()
opt = pidtuneOptions(Name,Value)
```

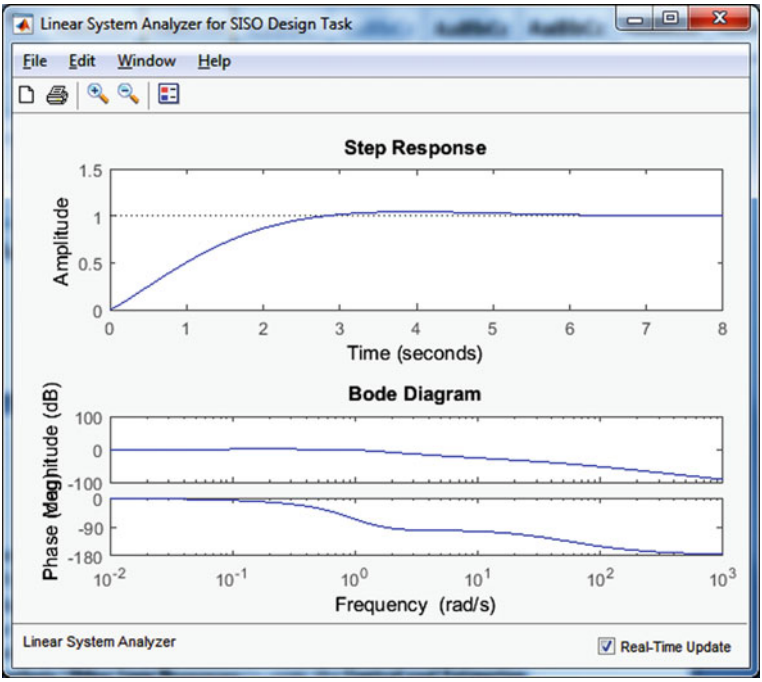


Fig. 5.50 Step and closed-loop Bode responses of the compensated system. Reprinted with the permission of MathWorks, Inc.

The first syntax returns the default option set for the `pidtune()` command, and the second syntax creates an option set with the options specified by one or more `Name-Value` pair arguments. Three typical `Name-Value` pairs are available:

- `Name: 'PhaseMargin', Value:` Numerical number for phase margin (default 60°).
- `Name: 'DesignFocus', Value:`
  - `'balanced'` (default): For a given robustness, tune the controller to balance reference tracking and disturbance rejection.
  - `'reference-tracking'`: Tune the controller to favor reference tracking if possible.
  - `'disturbance-rejection'`: Tune the controller to favor disturbance rejection.
- `Name: 'NumUnstablePoles', Value`: Numerical number of unstable poles (default 0).

An example of using these two functions is

```
sys = tf(1,[1 3 3 1]);
```

```
opts = pidtuneOptions
('PhaseMargin',60,'DesignFocus','disturbance-rejection');
[C,info] = pidtune(sys,'pid',opts);
```

Now let's use our previous example to illustrate how to use these pidtune-related functions to design a desired PID controller. The system plant is the same as one that is shown in Eq. (5.60)

$$G(s) = \frac{1}{s(s+1)}$$

To design a PID controller for this system, use the following MATLAB scripts to generate a Script file:

```
s = tf('s');
G = 1/(s*(s+1));
opts = pidtuneOptions('PhaseMargin',70,'DesignFocus','disturbance- rejection');
[C,info] = pidtune(G,'pid',opts)
```

The design results are shown in the Command Windows as below:

```
C =
             1
   Kp + Ki * --- + Kd * s
             s
  with Kp = 2.76, Ki = 1.13, Kd = 1.69
Continuous-time PID controller in parallel form.
info =
          Stable: 1
          CrossoverFrequency: 1.7810
          PhaseMargin: 70
```

By using the following MATLAB scripts, one can plot the step response for this closed-loop compensated system, which is shown in Fig. 5.51.

```
% use pidtune() and pidtuneOptions() functions to design PID controller
s = tf('s');
G = 1/(s*(s+1));
opts = pidtuneOptions('PhaseMargin',70,'DesignFocus','disturbance-rejection');
[C,info] = pidtune(G,'pid',opts);
P = C.Kp;
I = C.Ki;
D = C.Kd;
CC = (P + (I/s) + D*s);
H = CC*G;
sys = feedback(H, 1);
step(sys);
grid;
```

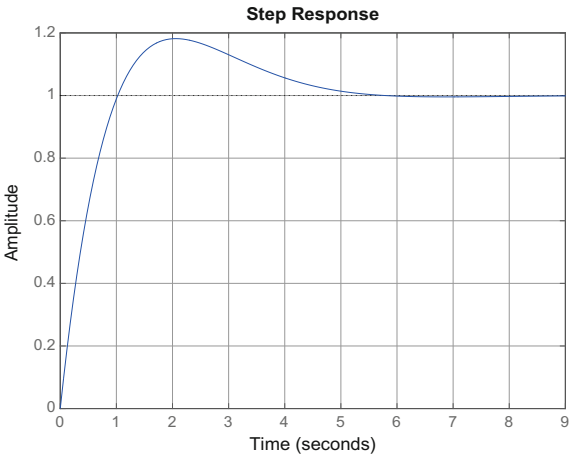


Fig. 5.51 Step response of the compensated closed-loop system

Now let's concentrate on the third function, `pidTuner()`, which is exactly a GUI tool that provides a more easy and convenient way to design and tune a PID controller.

This tool provides a Graphical User Interface (GUI) to facilitate the designing and tuning process for a PID controller. The PIDTuner opens the PID Tuner for designing a 1-DOF or 2-DOF PID controller. For a 1-DOF PID controller, a unit negative feedback path is provided by this tool to form a closed-loop control system block.

Four syntaxes are available to this tool:

(1) `pidTuner()`
(2) `pidTuner(sys)`
(3) `pidTuner(sys, type)`
(4) `pidTuner(sys, C)`

The first syntax just provides and opens a blank PID Tuner GUI window.

The second syntax provides and opens a PID Tuner GUI window with the `sys` as the default plant or process with a closed-loop step response plot. Then, users can make real-time tuning for this controller by adjusting some related parameters to get the optimal or ideal output of the system. These tunings can be achieved by just moving some progress bars on the GUI window to get the immediate response from the step response. The tuning parameters and results, such as `P`, `I`, and `D` gains, overshoot, rise time, settling time, and steady-state error, can be seen on the screen and step response immediately.

The third syntax allows users to select a controller type, such as `P`, `PI`, `PD`, or `PID` for the selected plant `sys`.

The fourth syntax enables users to select a base controller as the starting point to begin this tune process for the selected plant `sys`.

Now let's use an example plant to illustrate how to use this GUI Tool to design and tune our desired PID controller to obtain an optimal output for our system plant.

The plant model is one we identified in Sect. 4.7 at Chap. 4, a DC motor **Mitsumi 448 PPR Motor**. The identified transfer function for this motor is

$$G(s) = \frac{6.74}{(s + 1.79)}$$

Type the following commands on the MATLAB Command Window to invoke this tool:

```
>> s = tf('s');
>> sys = 6.74/(s+1.79);
>> pidTuner(sys)
```

The PID Tuner GUI window is opened with a step response, as shown in Fig. 5.52.

On the upper left corner, a `sys` is shown in the **Plant** box. If you opened a blank GUI, you can click the drop-down arrow in this box to import your desired plant. You can also select the different controller types from the **Type** box, but the default type is PI.

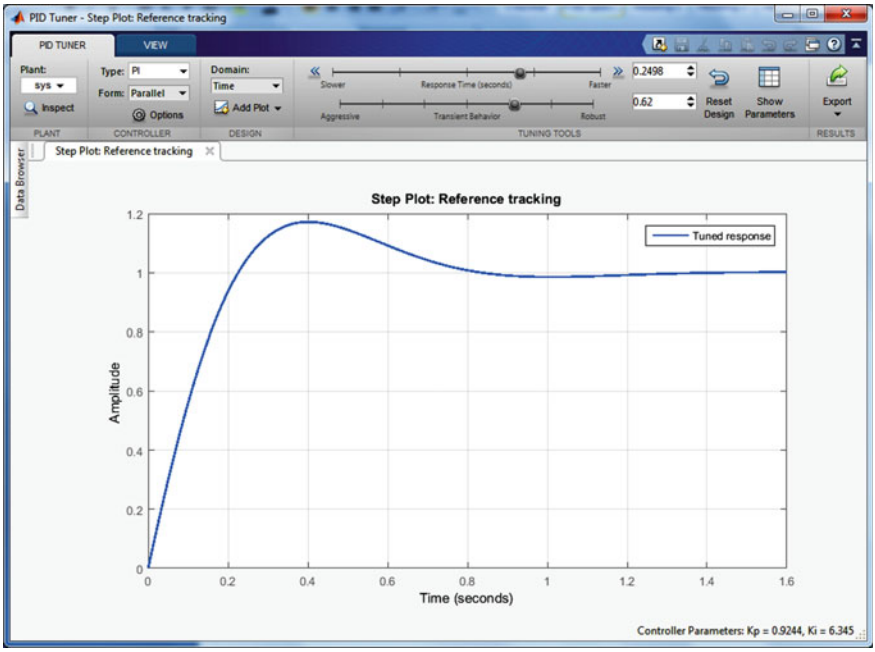


Fig. 5.52 The opened PID Tuner GUI Window

During the tuning process, you can check the current control parameters by clicking on the **Show Parameters** icon located on the upper right corner on this GUI window.

Now let's begin our tuning process. Two tuning tools, exactly two progress bars, are located on the top of this GUI window:

- Response Time (seconds)
- Transient Behavior

By moving the cursor on these two bars, we can change and tune control gains or parameters until we get the desired or optimal output by checking the step response.

To get faster and more robust response, move point on the bar to right; otherwise, move it to the left. You may need to make trade-off between the faster robust response and the small overshoot.

Our final tuning results are shown in Fig. 5.53. All control gains can be found from the lower right corner on the GUI window. You can also find these parameters by clicking on the **Show Parameters** icon, from which you can get all parameters related to the step response.

Next let's discuss another powerful tool provided by the MATLAB to assist users to design and test their control systems, which is MATLAB® Simulink®.

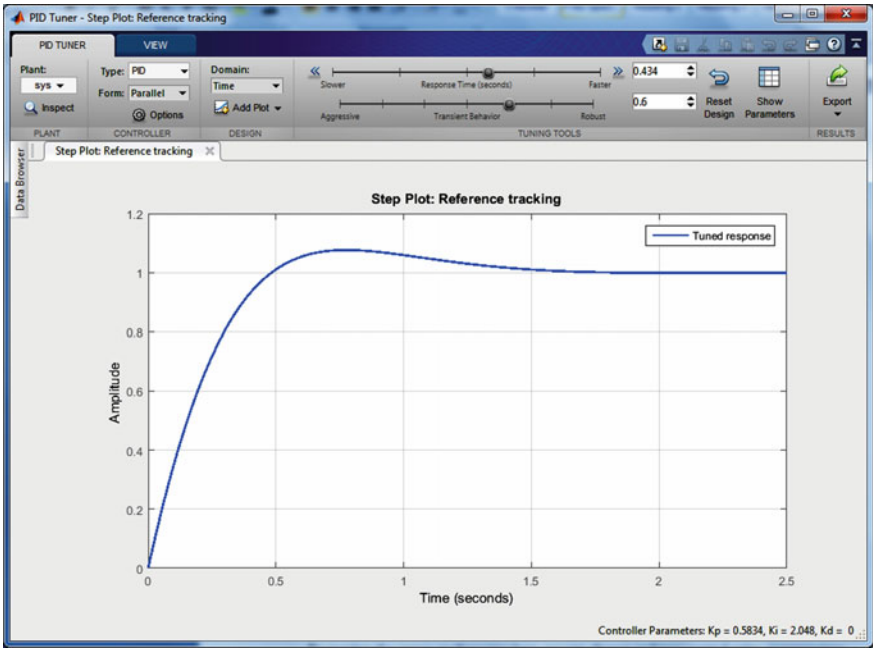


Fig. 5.53 The tuning result for our identified motor model

5.3.6.3 Use Simulink® to Assist Design and Analysis of Control Systems

MATLAB® Simulink® is a block diagram environment for multi-domain simulation and model-based design. It supports system-level design, simulation, automatic code generation, and continuous test and verification of embedded systems. Simulink provides a graphical editor, customizable block libraries, and solvers for modeling and simulating dynamic systems. It is integrated with MATLAB®, enabling you to incorporate MATLAB algorithms into models and export simulation results to MATLAB for further analysis.

Now let's still use our motor model as an example to illustrate how to use MATLAB® Simulink to help us to build our closed-loop block and perform desired simulation process to obtain the optimal output for our system.

Open the MATLAB® Simulink® by typing **Simulink** after the cursor ***fx***≫ in the MATLAB® command window, and the opened Simulink® Browser is shown in Fig. 5.54.

Create a new Simulink® model by going to **New Model** icon to open the Simulink® workspace. Perform the following operations to build this Simulink function block:

1. On the left **Simulink Browser** panel, click on the **Sources** item to open the **Sources** library. Drag and place the **Step** block into the Simulink workspace.
2. Click on the **Sinks** item to open the **Sinks** library. Drag and place the **Scope** block into the Simulink® workspace.

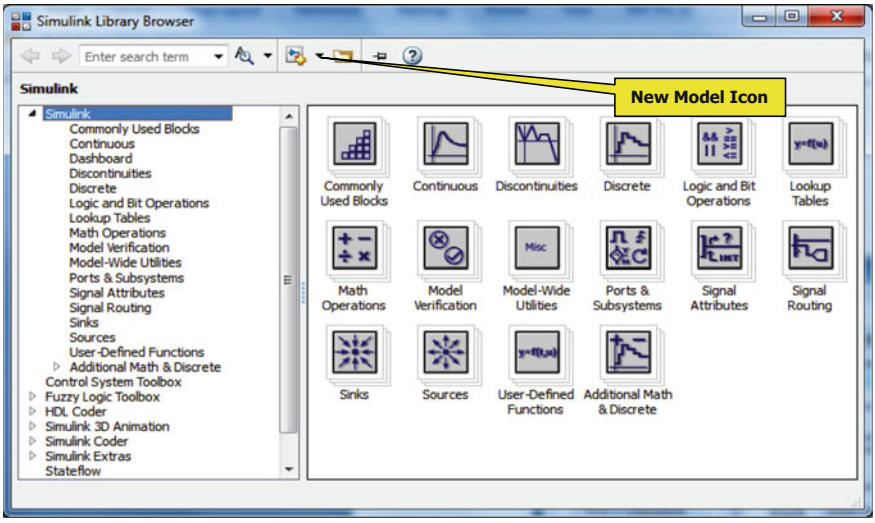


Fig. 5.54 The opened Simulink window. Reprinted with the permission of MathWorks, Inc.

3. Click on the **Commonly Used Blocks** item to open this library. Drag and place the **Sum** and the **Gain** blocks into the Simulink® workspace. Rotate the **Gain** block with 180° by going to **Format|Flip Block** menu item. Double-click the Sum block to open its property wizard. Then replace the second + sign with a minus − sign in the **List of signs** box. Click on the **OK** button to close this wizard.
4. Click on the **Continuous** item to open the **Continuous** library. Drag the **PID Controller** and **Transfer Fcn** blocks and place them into the Simulink® workspace.
5. Now make connections for each block one by one by dragging the output terminal of each block to the following block input terminal. One point to be noted is that when connecting between the output terminal of the **Transfer Fcn** and the input terminal of the **Scope** and the **Gain** blocks, make sure to do the connection between the **Transfer Fcn** and the **Gain** block first. Then drag the input terminal of the **Scope** block back to the **Transfer Fcn** block to do this connection.

Your finished Simulink® block function connection should match one that is shown in Fig. 5.55. Save this Simulink® model as **DCMotor.mdl** to a desired location in your host computer by going to **File|Save** menu item.

Now let's modify each block's parameters to meet our real system requirements.

1. Double-click on the **Step** block and enter **0** to the **Step time** and **0.005** to the **Sample time** text boxes, respectively.
2. Double-click on the **PID Controller** block and enter the tuned PID parameters as we did in the last section into the corresponding box for **Kp**, **Ki** and **Kd**, which is **Kp** = **0.5834**, **Ki** = **2.0484**, and **Kd** = **0**.

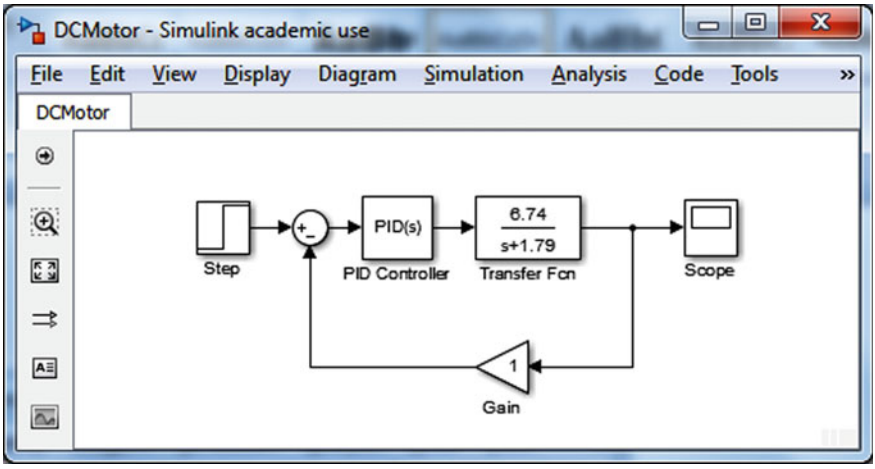


Fig. 5.55 The finished Simulink block connections

3. Double-click on the **Transfer Fcn** block to enter our identified dynamic model parameters for the DC motor, which is [**6.74**] for the numerator and [**1 1.79**] for the denominator.

Now double-click on the **Scope** block to open this scope to monitor our simulation result.

One can tune the PID parameters to try to get the optimal step response for this motor control system. After tuning the PID parameters, the optimal PID parameters are

- `Kp = 0.7500`
- `Ki = 2.0484`
- `Kd = 0.0000`

Start the simulation by going to **Simulation|Run** menu item. The final optimal simulated result is shown in Fig. 5.56.

We need to use these simulated control parameters into our real microcontroller control program to complete the closed-loop control to our motor system, and we will do this job in the next chapter.

5.3.7 The Nyquist Stability Criterion

As we discussed in the system stability and stability margins with Bode plot, regularly an increasing gain would make the system become unstable. However, in

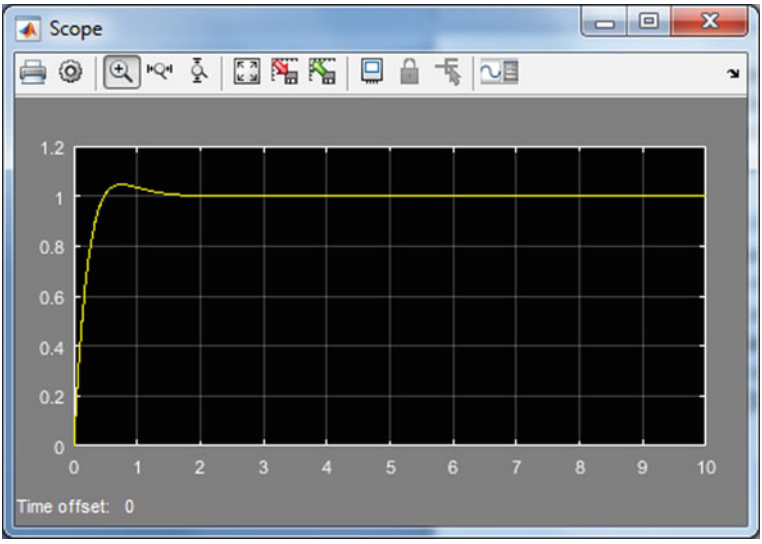


Fig. 5.56 The simulated step response result

some cases, this conclusion is untrue. Some unstable open-loop systems can become stable after they are made of closed-loop systems. In order to extend the stability study, an electrical engineer named Harry Nyquist at Bell Telephone Laboratories reported a new method to determine how to judge if a closed-loop system is stable or not. This is called Nyquist stability criterion.

Basically, the major advantage of using the Nyquist stability criterion is to use an open-loop system to study and check the stability of a closed-loop system that is made of that open-loop system. In fact, the Nyquist criterion relates the stability of a closed system to the open-loop frequency response and open-loop pole location.

Before we can continue discussing this criterion, let's first familiar us with some terminologies applied in this criterion.

Mapping—If we want to map a complex number in the s-plane into a function $F(s)$, we need to use a mapping technique. For example, we want to map variable $s = \sigma + j\omega$ to a function $F(s) = 2s + 1$, but the $F(s) = u + jv$ is also a complex plane. Thus, we have the following mapping:

$$F(s) = u + jv = 2s + 1 = 2(\sigma + j\omega) + 1 = (2\sigma + 1) + j2\omega$$
$$u = 2\sigma + 1 \text{ and } v = 2\omega$$

Contour—Consider a collection of points, which is called a contour A made of a square and is shown in Fig. 5.57a. Contour A can be mapped into Contour B in the $F(s)$ plane via the map function $F(s)$, as shown in Fig. 5.57b. The direction of traversal of the s-plane contour A is identical with that of the contour B in the $F(s)$ plane. By convention, the area within a contour to the right of the traversal of the contour is thought as the area enclosed by the contour. Thus, we consider the clockwise traversal of the contour as the positive.

In applying the Nyquist stability criterion to check the stability of a closed-loop system, generally, we need to plot a contour, such as contour A in Fig. 5.57, and let the contour A in the s-plane encircle the entire RHP. The resulting evaluation of the system $G(s)$ will only encircle the origin if the $G(s)$ has an RHP pole or zero.

As we mentioned, the advantage of using this criterion to check the stability of a system $G(s)$ is that a contour evaluation of an open-loop $KG(s)$ can be used to

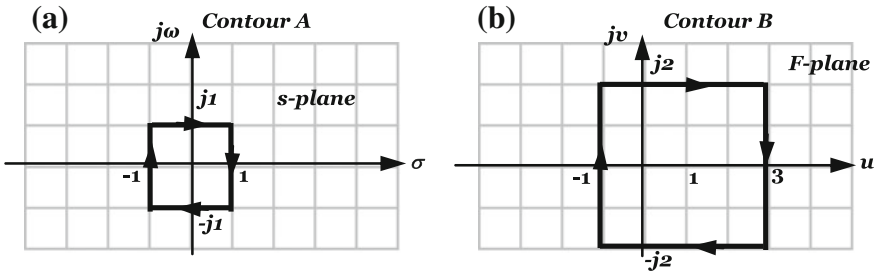


Fig. 5.57 The mapping between s-plane and $F(s)$-plane

determine stability of the closed-loop system. For example, for a unit negative feedback closed-loop system, its transfer function can be expressed as

$$T(s) = \frac{Y(s)}{R(s)} = \frac{KG(s)}{1 + KG(s)}$$

The characteristic equation or the denominator of this transfer function is $1 + KG(s) = 0$, which is equivalent to $KG(s) = -1$. If we map this $KG(s)$ to $F(s)$, then we can get $F(s) = KG(s)$ as we did in Fig. 5.57. If an evaluation contour of s enclosing the RHP contains a zero or pole of $1 + KG(s)$, then the evaluated contour of $1 + KG(s)$ will encircle the origin. Equivalently, the mapped $F(s) = KG(s)$ will encircle -1 and we then know that $1 + KG(s)$ contains a pole or zero in the RHP. Graphical presentation of this evaluation of $KG(s)$ or $F(s)$ in this way is called a Nyquist plot.

If a clockwise contour A encloses a zero of $1 + KG(s)$, which is the root for that closed-loop system, it is equivalent to a clockwise $F(s) = KG(s)$ encirclement of -1 with contour B. Thus, if A encloses a pole of $1 + KG(s)$, that is, if there is an unstable open-loop pole, there will be a counterclockwise $F(s) = KG(s)$ encirclement of -1. The net number of clockwise encirclements, N, equals the number of zeros in the RHP, Z, minus the number of poles in the RHP, P, which is

$$N = Z - P$$

In most real applications, $KG(s)$ will be a system who has zero response at infinite frequency, which means that the big arc of contour A at infinity results in $KG(s)$ being an infinitesimal point at the origin for that portion of A. So a complete evaluation of a physical system $KG(s)$ is achieved by allowing s to traverse the imaginary axis from $-j\infty$ to $+j\infty$. Similarly, the evaluation of $KG(s)$ is from $s = 0$ to $+j\infty$. Because the $KG(s)$ is a real function, the evaluation of $KG(s)$ from $s = -j\infty$ to 0 can be obtained by reflecting the $s = 0$ to $+j\infty$ portion about the real axis.

For example, a system whose plant can be expressed as a transfer function is

$$G(j\omega) = \frac{1}{1 + j\omega T} = \frac{1 - j\omega T}{1 + (\omega T)^2} = \frac{1}{1 + (\omega T)^2} - j\frac{\omega T}{1 + (\omega T)^2} = A + jB \tag{5.61}$$

$$A^2 + B^2 = \frac{1 + (\omega T)^2}{\left(1 + (\omega T)^2\right)^2} = \frac{1}{1 + (\omega T)^2} = A, \text{ which means } \left(A - \frac{1}{2}\right)^2 + B^2 = \left(\frac{1}{2}\right)^2 \tag{5.62}$$

The plot of this transfer function is shown in Fig. 5.58.

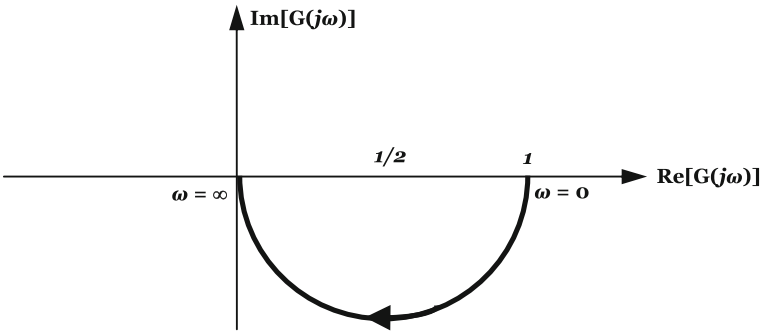


Fig. 5.58 An example of Nyquist plot

Generally, the following operation procedure can be used to check the stability of a system by using the Nyquist stability criterion:

(1) First make a mapping plot for $F(s) = KG(s)$ from $s = -j\infty$ to $+j\infty$. One can do the positive part from $s = 0$ to $+j\infty$ first, and then reflect the image about the real axis and add it to the preceding image.
(2) Check the number of clockwise encirclements of -1, and this is the number N. One can draw a straight line in any direction from -1 to ∞, and count the net number of clockwise crossing of that line by $G(s)$. If encirclements are in the counterclockwise direction, the number N will be negative.
(3) Find the number of unstable (RHP) poles of $G(s)$ and this is the number P.
(4) The number of unstable closed-loop roots, called Z, can be obtained as

$$Z = N + P \tag{5.63}$$

Although it is convenient to plot the Nyquist frequency response by hands, MATLAB provides more powerful tools to help users to build and draw all kinds of Nyquist plot for most systems, even some complicated systems. Now let's take a look at these tools and functions.

5.3.7.1 Tools and Functions Provided by MATLAB for Nyquist Analysis

Three functions related to Nyquist analysis are included in the MATLAB Control Toolbox, and they are

(1) `nyquist()`
(2) `nyquistplot()`
(3) `nyquistoptions()`

Basically, the first and the second functions are similar and both of them provide the plotting of a Nyquist frequency response in the *s*-plane.

Some popular syntaxes for the first and the second function include the following:

(1) `nyquist(sys)` plots the Nyquist plot of the dynamic system `sys`. The frequency range and number of points are chosen automatically.
(2) `nyquist(sys,{wmin,wmax})` draws the Nyquist plot for frequencies between `wmin` and `wmax` in Radians/Time Unit.
(3) `nyquist(sys,w)` uses the vector `w` of frequencies (in Radians/Time Unit) to evaluate the frequency response.
(4) `nyquist(sys1,sys2,...,w)` plots the Nyquist response of several systems `sys1`, `sys2`, … on a single plot. The frequency vector `w` is optional. You can specify a color, line style, and marker for each model, such as `nyquist (sys1,'r',sys2,'y—',sys3,'gx')`.
(5) `[RE,IM] = nyquist(sys,w)` and `[RE,IM,W] = nyquist(sys)` return the real parts RE and imaginary parts IM of the frequency response along with the frequency vector `w` if unspecified. No plot is drawn on the screen.

Two syntaxes exist for the third function and they are as follows:

(1) `P = nyquistoptions()` returns the default options for Nyquist plots. You can use these options to customize the Nyquist plot appearance using the command line later.
(2) `P = nyquistoptions('cstprefs')` initializes the plot options with the options you selected in the Control System and System Identification Toolbox Preferences Editor.

Now let's illustrate how to use these functions to help us to design and analyze some systems with some example systems.

5.3.7.2 Some Example of System Analyses by Using Nyquist Functions

Here, we have three systems expressed by their related transfer functions. We need to apply the Nyquist criterion to determine the stability of these unit feedback systems.

(1) $G(s) = \frac{s+3}{(s+2)(s^2+2s+25)}$
(2) $G(s) = \frac{s+10}{(s+2)(s+7)(s+50)}$
(3) $G(s) = \frac{500(s-2)}{(s+2)(s+8)(s+50)}$

For the first system, all closed-loop poles, $s = -2$, $s = -3$, and $s = -1 \pm j4.9$, and zeros, $z = -3$, are located at the LHP; therefore, $Z = P = N = 0$. This means that the Nyquist plot does not enclose $(-1, j0)$ point.

For the second system, similar conclusion can be obtained since all its closed-loop poles, $s = -2$, $s = -7$, $s = -50$, and $s = -10$ and zero $z = -10$, are located at the LHP. This makes $Z = P = N = 0$. The Nyquist plot also does not enclose $(-1, j0)$ point.

For the third system, it looks like a little complicated and we need to use some help from MATLAB functions to determine whether it is a stable system. Use the following MATLAB scripts to plot this Nyquist plot for this system:

```
k = 500;
s = tf('s');
G = (s-2)/((s+2)*(s+8)*(s+50));
sys = k*G;
nyquist(sys);
grid;
```

The running result is shown in Fig. 5.59. It can be found from this figure that the Nyquist plot encircled -1 point at one time, so $N = Z - P = 1 - 0 = 1$. The closed-loop system is unstable with one closed-loop pole in the right half plane.

5.4 Design PID Controllers with State-Space Method

The state-space representation is a mathematical model of a physical system represented by a set of input, output, and state variables related by first-order differential equations. The so-called state space refers to the Euclidean space in which the variables on the axes are the state variables. The state of the system can be represented as a vector within that space. In this model, dynamic systems can be

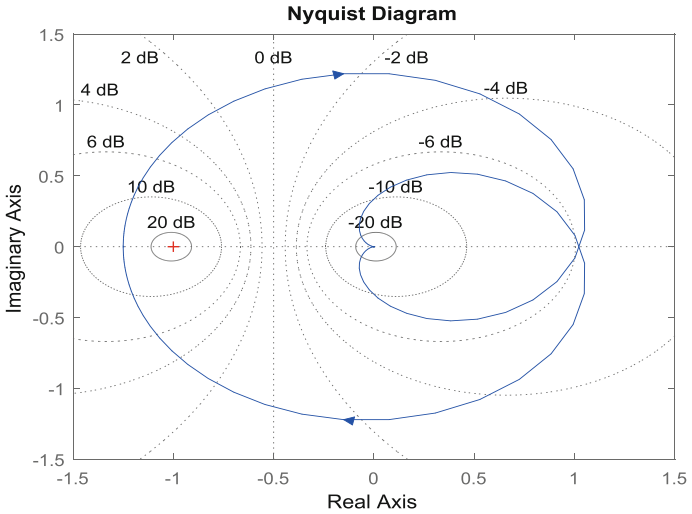


Fig. 5.59 The Nyquist plot for the third system

described by a set of first-order differential equations in variables called state, and the solution may be viewed as a trajectory in the space.

The state-space representation is also known as the time-domain approach and it provides a convenient and compact way to model and analyze systems with multiple-input, multiple-output (MIMO). With inputs and outputs, we would otherwise have to write down Laplace transforms to encode all the information about a system. Unlike the frequency domain and root locus approaches, the use of the state-space representation is not limited to systems with linear components and zero initial conditions. Because of these features, state-space model can be used in many different areas.

Compared with frequency and root locus methods, we discussed in previous sections, which have been called *classical control design* methods, the state-space method has often been referred to as *modern control design* method even this method was introduced in 1950s. One of possible reasons for this may because this method can handle MIMO systems. Although the state-space design provides this advantage over other design methods, we still concentrate our discussions on single-input-single-output (SISO) systems in this section.

5.4.1 State-Space Variable

A *state variable* is one of the sets of variables that are used to describe the mathematical *state* of a dynamical system. Intuitively, the state of a system describes enough about the system to determine its future behavior in the absence of any external forces affecting the system. Models that consist of coupled first-order differential equations are said to be in state-variable form.

The internal state variables are the smallest possible subset of system variables that can represent the entire state of the system at any given time. The minimum number of state variables required to represent a given system, $\boldsymbol{n}$, is usually equal to the order of the system's differential equation. It is important to understand that converting a state-space realization to a transfer function form may lose some internal information about the system, and may provide a description of a system which is stable, when the state-space realization is unstable at certain points.

For example, a standard second-order LTI system with input u and output y can be described by a differential equation as

$$\frac{\mathrm{d}^2 y}{\mathrm{d}t^2} + 2\xi\omega_\mathrm{n}\frac{\mathrm{d}y}{\mathrm{d}t} + \omega_\mathrm{n}^2 y = Ku \tag{5.64}$$

If we define two state variables as $x_1 = y$ and $x_2 = \mathrm{d}y/\mathrm{d}t$, these two state variables can be written as a matrix as

$$\boldsymbol{x} = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} y \\ \dfrac{\mathrm{d}y}{\mathrm{d}t} \end{bmatrix} = \begin{bmatrix} y \\ \dot{y} \end{bmatrix} \tag{5.65}$$

In Eq. (5.65), two state variables x_1 and x_2 construct a column vector $\boldsymbol{x}$. Furthermore, if we place coefficients of the state equation shown in (5.64) into a square matrix **A**, and coefficients of the input into a column matrix **B**, the Eq. (5.64) can be expressed as

$$\begin{bmatrix} \dot{x}_1 \\ \dot{x}_2 \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ -\omega_n^2 & -2\xi\omega_n \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} + \begin{bmatrix} 0 \\ K \end{bmatrix} u \quad \text{or} \quad \dot{\boldsymbol{x}} = \mathbf{A}x + \mathbf{B}u \tag{5.66}$$

where **A** is the system matrix and **B** is the input matrix, and both can be expressed as

$$\mathbf{A} = \begin{bmatrix} 0 & 1 \\ -\omega_{\mathrm{n}}^2 & -2\xi\omega_{\mathrm{n}} \end{bmatrix}; \quad \mathbf{B} = \begin{bmatrix} 0 \\ K \end{bmatrix} \tag{5.67}$$

If we take the output $y = x_1$, then we have the output equal to

$$y = \begin{bmatrix} 1 & 0 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \quad \text{or} \quad y = \mathbf{C}\boldsymbol{x} + \mathbf{D}u \tag{5.68}$$

where **C** is a row vector called the output matrix and the **D** is also a row vector representing a possible direct path from the input u to the output y, and it is referred as the direct transmission.

Now the differential equation described the input–output relationship in (5.64) has been converted or replaced by the state variable representation given by (5.66) and (5.68).

The state variables can be used to describe the internal behavior of a control plant or process. Generally, they represent physical quantities resulting from a mathematical modeling of the plant based on physical laws. In fact, the state variables are combined into the so-called state vector $\boldsymbol{x}$. As this vector is a function of time, $\boldsymbol{x}(t)$, moving in an n-dimensional vector space, the methods based on the corresponding system description are referred to as state-space methods. The vector $\boldsymbol{x}(t)$ at some certain time t is called the *state* of the system at time t.

5.4.2 *System Transfer Functions and Eigenvalues in State Space*

In order to get a clear picture between the state variable equations and frequency response as well as poles and zeros for a given system, let's take a closer look at

state-space equations. In fact, the transfer function of a continuous time-invariant linear state-space model can be derived in the following way.

First, taking the Laplace transform of $\dot{\boldsymbol{x}} = \mathbf{A}\boldsymbol{x} + \mathbf{B}u$, which yields

$$s\mathbf{X}(s) - x(0) = \mathbf{A}\mathbf{X}(s) + \mathbf{B}\mathbf{U}(s) \tag{5.69}$$

Next, we simplify for $\mathbf{X}(s)$, giving

$$(s\mathbf{I} - \mathbf{A})\mathbf{X}(s) = x(0) + \mathbf{B}\mathbf{U}(s) \tag{5.70}$$

and thus

$$\mathbf{X}(s) = (s\mathbf{I} - \mathbf{A})^{-1}x(0) + (s\mathbf{I} - \mathbf{A})^{-1}\mathbf{B}\mathbf{U}(s) \tag{5.71}$$

Substituting for $\mathbf{X}(s)$ in the output equation

$$\begin{aligned} &\mathbf{Y}(s) = \mathbf{C}\mathbf{X}(s) + \mathbf{D}\mathbf{U}(s), \text{ which giving} \\ &\mathbf{Y}(s) = \mathbf{C}\Big[(s\mathbf{I} - \mathbf{A})^{-1}x(0) + (s\mathbf{I} - \mathbf{A})^{-1}\mathbf{B}\mathbf{U}(s)\Big] + \mathbf{D}\mathbf{U}(s) \end{aligned} \tag{5.72}$$

The transfer function $G(s)$ is defined as the ratio of the output to the input of a system considering its initial conditions to be zero ($x(0) = 0$). However, the ratio of a vector to a vector does not exist, so we consider the following condition satisfied by the transfer function

$$\mathbf{G}(s) \times \mathbf{U}(s) = \mathbf{Y}(s) \tag{5.73}$$

comparing with the equation for $\mathbf{Y}(s)$ above gives

$$\mathbf{G}(s) = \mathbf{C}(s\mathbf{I} - \mathbf{A})^{-1}\mathbf{B} + \mathbf{D} \tag{5.74}$$

Clearly, $\mathbf{G}(s)$ must have m by n dimensionality, and thus has a total of $m \times n$ elements. So for every input there are m transfer functions with one for each output. This is why the state-space representation can easily be the preferred choice for multiple-input, multiple-output (MIMO) systems.

For example, we have a system with the following state variable description

$$\mathbf{A} = \begin{bmatrix} -5 & -4 \\ 1 & 0 \end{bmatrix}; \quad \mathbf{B} = \begin{bmatrix} 1 \\ 0 \end{bmatrix}; \quad \mathbf{C} = [\,0 \quad 1\,]; \quad \mathbf{D} = 0.$$

Translating these to the transfer function form, we have

$$s\mathbf{I}-\mathbf{A}=\begin{bmatrix} s+5 & 4 \\ -1 & s \end{bmatrix}$$

$$(s\mathbf{I}-\mathbf{A})^{-1}=\frac{\begin{bmatrix} s & -4 \\ 1 & s+5 \end{bmatrix}}{s(s+5)+4}$$

$$G(s)=\frac{\begin{bmatrix}0 & 1\end{bmatrix}\begin{bmatrix} s & -4 \\ 1 & s+5 \end{bmatrix}\begin{bmatrix}1\\0\end{bmatrix}}{s(s+5)+4}=\frac{\begin{bmatrix}1 & s+5\end{bmatrix}\begin{bmatrix}1\\0\end{bmatrix}}{s(s+5)+4}=\frac{1}{s(s+5)+4}=\frac{1}{(s+1)(s+4)}$$

As we know, a pole at a transfer function $G(s)$ is exactly a value of frequency $s = j\omega$, and if its value is $s = \lambda_i$, then the system response in the time domain would be $K_i e^{\lambda_i t}$, with no external input u ($u = 0$) or no forcing function. The λ_i can be considered as a natural frequency of the system. If we map this situation into the state space, we have the state variable representation as

$$\dot{\boldsymbol{x}} = \mathbf{A}\boldsymbol{x} \tag{5.75}$$

If we have initial value $\boldsymbol{x}(0) = \boldsymbol{x}_0$, which is a constant initial vector, the solution should be $\boldsymbol{x}(t) = e^{\lambda_i t}\boldsymbol{x}_0$. Equation (5.75) becomes

$$\dot{\boldsymbol{x}}(t) = \lambda_i e^{\lambda_i t}\boldsymbol{x}_0 = \mathbf{A}\boldsymbol{x} = \mathbf{A}e^{\lambda_i t}\boldsymbol{x}_0 \quad \text{or} \quad \mathbf{A}\boldsymbol{x}_0 = \lambda_i \boldsymbol{x}_0 \tag{5.76}$$

Equation (5.76) can be written as

$$(\lambda_i\mathbf{I}-\mathbf{A})\boldsymbol{x}_0 = 0 \tag{5.77}$$

Equations (5.76) and (5.77) are referred to as equations of eigenvalues λ_i and eigenvectors $\boldsymbol{x}_0$ of the matrix $\mathbf{A}$. For a nonzero $\boldsymbol{x}_0$, the above equation is satisfied if and only if

$$\det[\lambda_i\mathbf{I}-\mathbf{A}] = 0 \tag{5.78}$$

These equations indicate that the poles of the transfer function are the eigenvalues of the system matrix $\mathbf{A}$. A zero of a system transfer function is the value of frequency $s = j\omega$ such that if the input is an exponential at the zero frequency, or $u(t) = u_0 e^{st}$, then the output is zero, $y(t) = 0$.

This means that the system has a nonzero input but nothing comes out.

We discussed how to translate or convert the state variable representation forms to the transfer function forms in this section; now we need to do the job in an opposite way, or convert from the transfer functions to the state variables forms. This can be done by using a block diagram representation. The key of this representation is to use an integrator as the central element and it is appropriate for the first-order state-space variables and equations.

Generally, a transfer function can be represented as

$$G(s) = \frac{Y(s)}{U(s)} = \frac{s^m + b_{m-1}s^{m-1} + \cdots + b_1 s + b_0}{s^n + a_{n-1}s^{n-1} + \cdots + a_1 s + a_0}; \quad (n \geq m)$$

and all coefficients are real positive numbers.

Now we dividing both numerator and denominator above by s^n, we get

$$G(s) = \frac{Y(s)}{U(s)} = \frac{s^{-(n-m)} + b_{m-1}s^{-(n-m+1)} + \cdots + b_1 s^{-(n-1)} + b_0 s^{-n}}{1 + a_{n-1}s^{-1} + \cdots + a_1 s^{-(n-1)} + a_0 s^{-n}} \tag{5.79}$$

Equation (5.79) can be expressed as a block diagram with a series of integrators, $1/s$, as shown in Fig. 5.60.

Based on the coefficients, a_i and b_i, we can derive the related matrix for **A**, **B** and **C**. The values for each matrix are

$$\begin{aligned} \dot{x} = Ax + Bu &= \begin{bmatrix} 0 & 1 & 0 & \dots & 0 & 0 \\ 0 & 0 & 1 & \dots & 0 & 0 \\ 0 & 0 & 0 & \dots & 0 & 0 \\ & \dots & & & & \\ 0 & 0 & 0 & \dots & 0 & 1 \\ -a_0 & -a_1 & -a_2 & \dots & -a_{n-1} & -a_n \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ \cdot \\ \cdot \\ x_n \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ m \end{bmatrix} u; \\ y = Cx &= \begin{bmatrix} b_0 & b_1 & b_2 & \dots & b_{n-1} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ \cdot \\ \cdot \\ x_n \end{bmatrix} \end{aligned} \tag{5.80}$$

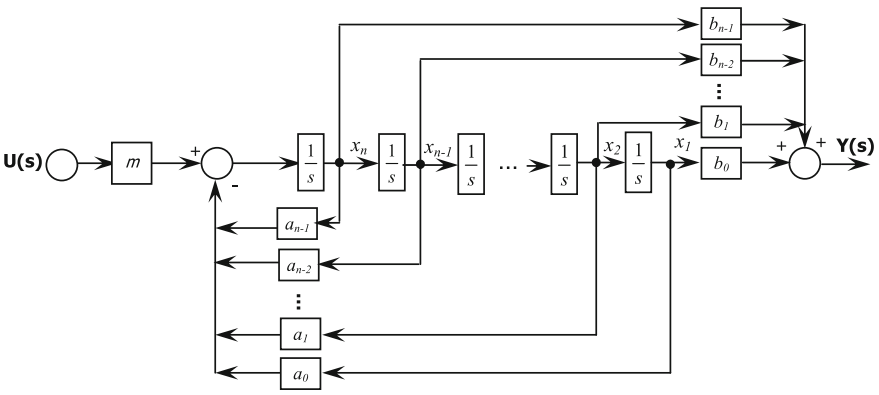


Fig. 5.60 The block diagram representation of the state variable equations

Now let's use an example to illustrate how to use this method to convert from a transfer function to the related state-space variable matrices and associated equations.

For example, we have a system plant with a transfer function $G(s) = \frac{2s^3 + 5s^2 + 6s + 8}{s^4 + 4s^3 + 8s^2 + 16s + 6}$. First, divide both numerator and denominator by s^4, so we have

$$G(s) = \frac{2s^{-1} + 5s^{-2} + 6s^{-3} + 8s^{-4}}{1 + 4s^{-1} + 8s^{-2} + 16s^{-3} + 6s^{-4}}$$

The related block diagram is shown in Fig. 5.61. Compared with Eqs. (5.79) and (5.80), we can find

$$A = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ -6 & -16 & -8 & -4 \end{bmatrix}; \quad B = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix}; \quad C = [8 \quad 6 \quad 5 \quad 2]$$

To extend this representation, we can obtain standard state variable matrix and state-space equations as

$$\dot{x}_1 = x_2, \dot{x}_2 = x_3, \dot{x}_3 = x_4, \dot{x}_4 = -6x_1 - 16x_2 - 8x_3 - 4x_4 + u$$
$$y = 8x_1 + 6x_2 + 5x_3 + 2x_4$$

Comparing these results with the block diagram shown in Fig. 5.61, it can be found that each state variable x_i is followed by an integrator $1/s$. Inversely, each

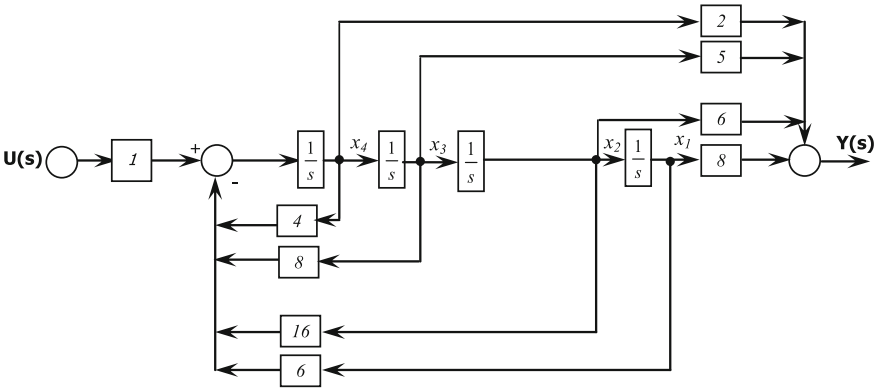


Fig. 5.61 The block diagram for the example transfer function

derivative of a state x_i is the previous state x_{i-1}. This is making sense that the following state is an integration of the previous state.

A key point to be noted is that the combined operation or the summary operation for all coefficients a_i is a minus operation.

Another example is $T(s) = \frac{2s^2+8s+6}{s^3+8s^2+16s+6}$. Dividing numerator and denominator by s^3, we get $T(s) = \frac{2s^{-1}+8s^{-2}+6s^{-3}}{1+8s^{-1}+16s^{-2}+6s^{-3}}$. The state-space matrix representation for this transfer function is $\dot{x} = Ax + Bu = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ -6 & -16 & -8 \end{bmatrix} x + \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} u$. The output can be expressed as $y = \begin{bmatrix} 6 & 8 & 2 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}$.

5.4.3 Block Diagram and State-Space Design

As we can see from Eq. (5.64), the input–output relationship of a system can be described by a set of differential equations. However, this differential equation can be replaced by the state variable representations given by (5.66) and (5.68). The state-space design method utilizes this way of describing the plant to directly obtain the feedback controllers or compensators without the need to find transforms. Basically, the state-space design method can be considered as a sequence of independent steps. Totally, four steps are widely implemented in the state-space design for a closed-loop control system.

The **first step** is to find the control or control law as feedback of a linear combination of all state variables. The objective of this step is to design the block controller such that it generates a control input signal $u(t)$ from a reference signal $r(t)$ and the state variables $(x_1, \ldots, x_n)$ with the following properties:

- The effects of initial values of the state variables decay as $t \rightarrow \infty$.
- The control output $y(t)$ should be able to track the reference signal $r(t)$ as well as possible.
- Both of these processes will run with desired dynamics and this will be achieved by specifying poles at the space domain.

In other words, the purpose of the control law design is to allow us to determine or design a set of pole locations for the closed-loop target system based on the given specifications, such as rise time, overshoot, settling time, or steady-state error, to get the desired dynamic outputs.

The **second step** is to design an estimator or observer to compute an estimation of the entire state vector when provided with the measurements of the system as shown in Eq. (5.68). In fact, the technical realization of pure state feedback requires that all of the state variables are accessible, or they can be continuously measured. However, this is not always the case if the measurement is expensive, difficult, or impossible for principal reasons. Thus, it is desirable to reduce the number of variables fed back. As an extreme, only one single variable, typically the control output variable y, is measured and fed back while the other design objectives remain unchanged.

The resulting controller is called output-feedback controller. In order to preserve the advantages of state feedback, it is a good idea to estimate the inaccessible state variables from the accessible ones and to run the state feedback controller with the estimated variables. Such a state estimator is also called a state observer. By introducing the estimated state variables $\hat{x}_1, \ldots, \hat{x}_n$ into the state feedback controlled system, the resulted structure is shown in Fig. 5.62.

The **third step** is made of combining the control law obtained in step 1 and the estimator obtained from step 2 together to get the control input u, as shown in Fig. 5.62.

The **fourth step** is to add the input reference into the system to make the plant output to follow or track external commands with given specifications.

Two important terminologies are needed to be emphasized when designing the state-space controllers: Observability and Controllability.

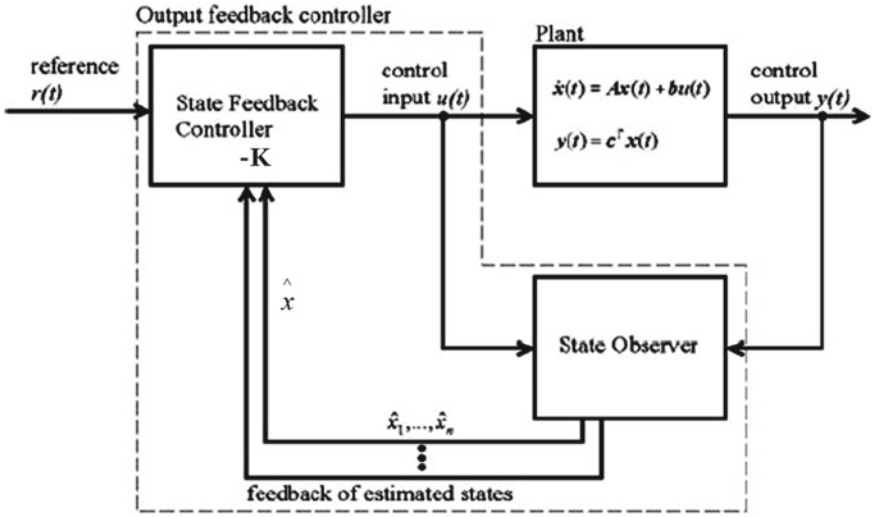


Fig. 5.62 Output feedback by combining state observer and state feedback controller

State observation is not possible for all systems in all situations, for example, it is not for any plant and any choice of control output variable. If some state variable does neither directly nor indirectly take effect on the control output, then it is impossible to estimate this state variable from measuring the output; the system is not completely observable.

Furthermore, if some part system is neither directly nor indirectly affected by the control input, then it is impossible to specifically influence its dynamics; the system is not completely controllable. Therefore, the notions of controllability and observability are key concepts in the analysis and design of state-space control systems.

5.4.4 Control Law Design for Pole Placement

The first step to use state-space method to design closed-loop control system is to determine the control law as feedback of a linear combination of all state variables, which means

$$u = -\mathbf{K}\boldsymbol{x} = -\begin{bmatrix} K_1 & K_2 & \dots & K_n \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \cdot \\ \cdot \\ x_n \end{bmatrix} \tag{5.81}$$

Generally, for an nth-order system, we have n feedback gains, such as K_1, K_2, …, K_n, with n roots or poles of the system, as shown in Fig. 5.63. This is good enough for us to select any desired root location by choosing the different values of K_i.

Substituting Eq. (5.81) into (5.66), we have

$$\dot{\boldsymbol{x}} = \mathbf{A}\boldsymbol{x} - \mathbf{B}\mathbf{K}\mathbf{x} \tag{5.82}$$

The related characteristic equation of this closed-loop system is

$$\det[s\mathbf{I} - (\mathbf{A} - \mathbf{B}\mathbf{K})] = 0 \tag{5.83}$$

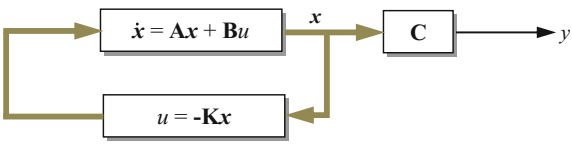


Fig. 5.63 Assumed system for the control law design

When solving this equation, it creates an nth-order polynomial in s containing all gains, such as K_1, K_2, … K_n. Exactly, the control law design is to select or pick up all gains in the gain vector **K** to meet Eq. (5.83). In other words, the roots in Eq. (5.83) are in the desired locations. Also, the prerequisite for the calculation of **K** is that the system must be controllable, which means that the output will be determined or affected by any inputs or input changes.

Theoretically, this selection process of **K** is executable, but in practice it may be complicated and needs some iteration by designers. Thus, to select or calculate the gains **K** by hand is almost never done. Fortunately, MATLAB provides a set of functions to help users to complete these complicated calculations. Since we divide our discussions about the state-space design and analysis into four steps, therefore, we will introduce these functions in each separate section.

For control law design, six functions are popular and widely implemented:

(1) `acker()`
(2) `place()`
(3) `lqr()`
(4) `lqry()`
(5) `lqi()`
(6) `lqgtrack()`

The first function is built based on Ackermann's formula to perform this calculation. However, this function is only good for the design of SISO systems with small (≤ 10) number of state variables. For more complex systems, a more reliable function, `place()`, should be applied. But an additional limitation of using this function is that none of the desired closed-loop poles can be repeated, which means that all poles must be different or distinct. The good thing is that this limitation is not applied to the `acker()` function.

Both of the first and the second functions require inputs consisting of the system description matrices, **A** and **B**, and a vector **p**, which is the locations of the n desired poles. The output of these functions is the feedback gain **K**.

The syntaxes for these functions are

$$\texttt{K} = \texttt{acker}(\texttt{A}, \texttt{B}, \texttt{p}); \quad \text{or} \quad \texttt{K} = \texttt{place}(\texttt{A}, \texttt{B}, \texttt{P});$$

The basic requirement and consideration for control law design is to try to reduce the control effort. This effort is directly proportional to the moving distance between the open-loop poles and the closed-loop poles since these moving actions are performed by the feedback gains. Also if a zero is closed to a pole, the system may be nearly uncontrollable and moving such pole may need large control effort.

The third function `lqr()` is called Linear Quadratic Regulator (LQR) function and it is used to calculate the optimal gain matrix **K** by minimizing a cost function

$$J(u) = \int_0^\infty \left[\mathbf{x}^{\mathrm{T}}\mathbf{Q}\mathbf{x} + \mathbf{u}^{\mathrm{T}}\mathbf{R}\mathbf{u}\right]\mathrm{d}t$$

where **Q** is an $n \times n$ state weighting matrix, **R** is an $m \times m$ control weighting matrix, and m is the number of control inputs in a multiple-input system. For a SISO system, $m = 1$ and **R** is a scalar value R. The weights **Q** and R can be selected by designers by trial-and-error to get the desired balance between state errors $\mathbf{x}^{\mathrm{T}}\mathbf{x}$ and control usage u^2. Generally, **Q** is a diagonal matrix with a weighting factor of one or more of the state vector elements when $R = 1$.

The syntaxes of this function are

```
[K,S,e] = lqr(SYS,Q,R,N)
[K,S,e] = LQR(A,B,Q,R,N)
```

The matrix `N` is set to zero when omitted. Also, the returned values are the solution `S` of the associated algebraic Riccati equation and the closed-loop eigenvalues e = eig(**A** − **B** * **K**).

Similarly, the function `lqry()` is also used to calculate the optimal gain matrix **K** by minimizing a cost function

$$J(u) = \int_0^\infty \left[\mathbf{y}^{\mathrm{T}}\mathbf{Q}\mathbf{y} + \mathbf{u}^{\mathrm{T}}\mathbf{R}\mathbf{u}\right]\mathrm{d}t$$

The syntax for this function is `[K,S,e] = lqry(sys,Q,R,N)`. This function returns the optimal gain matrix **K**, the Riccati solution `S`, and the closed-loop eigenvalues e = eig(**A** − **B** * **K**). The state-space model `sys` specifies the continuous- or discrete-time plant data (A, B, C, D). The default value $N = 0$ is assumed when `N` is omitted.

The function `lqi()` is used to compute an optimal state feedback control law for the tracking loop shown in Fig. 5.64.

For a plant `sys` with state-space equations `dx/dt = Ax + Bu`, `y = Cx + Du`, or their discrete-time counterpart, the state feedback control is of the form `u = −K [x; xi]` where `xi` is the integrator output. This control law ensures that the output

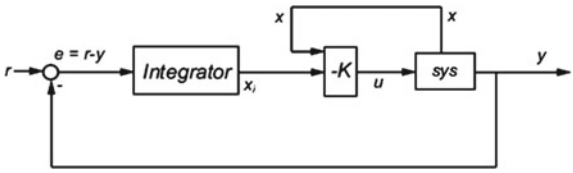


Fig. 5.64 Control law design for an integrator closed-loop control system

y tracks the reference command r. For MIMO systems, the number of integrators is equal to the dimension of the output y.

The syntax of this function is `[K,S,E] = lqi(sys,Q,R,N)`. The matrix `N` is set to zero when omitted. This function returns the solution `S` of the associated algebraic Riccati equation and the closed-loop eigenvalues `E`.

The last function `lqgtrack()` forms a Linear Quadratic Gaussian (LQG) servo controller with integral action for the loop shown in Fig. 5.65. This compensator ensures that the output y tracks the reference command r and rejects process disturbances w and measurement noise v. The function assumes that r and y have the same length.

Now let's use an example to illustrate how to use these functions to help us to derive the optimal gains based on the given conditions as below.

The system plant is

$$G(s) = \frac{1}{(s+1)^2} = \frac{1}{s^2+2s+1} \tag{5.84}$$

The specifications are closed-loop roots with ω_n = 3 rad/s and ξ = 0.8.

Referring to Eqs. (5.67) and (5.68), we have

$\mathbf{A} = \begin{bmatrix} 0 & 1 \\ -1 & -2 \end{bmatrix}$; $\mathbf{B} = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$; $\mathbf{C} = [1 \quad 0]$ and $\mathbf{D} = 0$, where $2\xi\omega_0 = 2$ and $\omega_0 = 1$.

The desired characteristic equation is

$$s^2 + 2\xi\omega_n s + \omega_n^2 = 0$$

Use the following MATLAB scripts to obtain the feedback gain matrix **K**:

```
% test acker() function to get gain matrix K
A = [0 1; -1 -2];
B = [0; 1];
Wn = 3;
Ze = 0.8;
p = roots([1  2*Wn*Ze   Wn^2]);
K = acker(A, B, p)
```

The running result is `K = 8.0000  2.8000`.

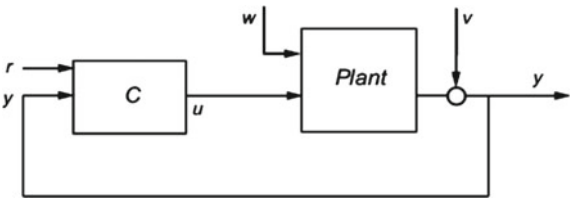


Fig. 5.65 Linear Quadratic Gaussian (LQG) servo controller

Next let's investigate the roots obtained by using `lqr()` function with three **Q** weighting matrices shown below:

$$\mathbf{Q}_1 = \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix}; \quad \mathbf{Q}_2 = \begin{bmatrix} 100 & 0 \\ 0 & 0 \end{bmatrix}; \quad \mathbf{Q}_3 = \begin{bmatrix} 100 & 0 \\ 0 & 5 \end{bmatrix}$$

Use the following MATLAB scripts to test and obtain different design parameters:

```
% test lqr() function to get different design parameters
A = [0 1; -1 -2];
B = [0; 1];
Q1 = [1   0; 0   0];
Q2 = [100   0; 0   0];
Q3 = [100   0; 0   5];
R = 1;
K = lqr(A, B, Q3, R)
p = eig(A - B*K);
[Wn, Ze] = damp(p)
```

Run this piece of script codes three times with different **Q**, such as $\mathbf{Q}_1$ in the first time, $\mathbf{Q}_2$ for the second time, and $\mathbf{Q}_3$ for the third time by modifying the code line `K = lqr(A, B, Q3, R)`. The running results are

$\mathbf{K} = [0.4142 \;\; 0.1974]$,	$\mathbf{K} = [9.0499 \;\; 2.7010]$,	$\mathbf{K} = [9.0499 \;\; 3.2057]$
$\omega_n = 1.1892$,	$\omega_n = 3.1702$,	$\omega_n = 3.1702$
$\xi = 0.9239$,	$\xi = 0.7415$,	$\xi = 0.8211$

It can be found that for this simple system, the function `acker()` is good enough to find the gain matrix **K**. However, for more complex systems with higher order roots, the function `lqr()` is more appropriate to be used to obtain the optimal value for all roots.

5.4.5 *Estimator Design*

The control law designed in the last section assumed that all state variables can be measured and available for the feedback design. However, in most situations, this assumption is untrue and some of the variables are not measured or cannot be measured. In order to solve this problem, it is necessary to design an estimator to estimate those "hidden" or unavailable variables by reconstructing all of those missed state variables of a system from measurement of some of them. Those estimated variables should be denoted by $\hat{\mathbf{x}}$. In this way, we can use these estimated variables to replace all unavailable variables to perform and fulfill our control designs and implementations.

In order to get correct estimations, we need to feed back the difference between the measured and estimated outputs and correct the estimated model continuously

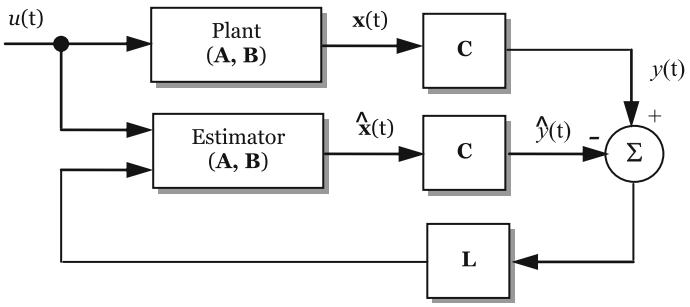


Fig. 5.66 Closed-loop estimator

until to get the optimal results. This process can be illustrated by block diagram shown in Fig. 5.66.

The estimated state variables can be expressed as

$$\dot{\hat{\mathbf{x}}} = \mathbf{A}\hat{\mathbf{x}} + \mathbf{B}\mathrm{u} + \mathbf{L}(y - \mathbf{C}\hat{\mathbf{x}}) \tag{5.85}$$

where $\mathbf{L}$ is a proportional gain defined as

$$\mathbf{L} = [l_1, l_2, \ldots, l_n]^{\mathrm{T}}$$

This gain vector is chosen to obtain satisfactory error limitations. The error in the estimate can be defined as

$$\tilde{\mathbf{x}} = \mathbf{x} - \hat{\mathbf{x}} \tag{5.86}$$

The dynamics of the error is obtained by subtracting the $\hat{\mathbf{x}}$ in Eq. (5.85) from the state equation in (5.66), and we can get

$$\begin{aligned}\dot{\tilde{\mathbf{x}}} &= \dot{\mathbf{x}} - \dot{\hat{\mathbf{x}}} \\ &= \mathbf{A}\mathbf{x} + \mathbf{B}\mathrm{u} - \mathbf{A}\hat{\mathbf{x}} - \mathbf{B}\mathrm{u} - \mathbf{L}(y - \mathbf{C}\hat{\mathbf{x}}) = \mathbf{A}(\mathbf{x} - \hat{\mathbf{x}}) - \mathbf{L}y + \mathbf{L}\mathbf{C}\hat{\mathbf{x}} = \mathbf{A}(\mathbf{x} - \hat{\mathbf{x}}) - \mathbf{L}\mathbf{C}(\mathbf{x} - \hat{\mathbf{x}})\end{aligned}$$

Using Eq. (5.86) and putting it into this above equation, we have

$$\dot{\tilde{\mathbf{x}}} = (\mathbf{A} - \mathbf{L}\mathbf{C})\tilde{\mathbf{x}} \tag{5.87}$$

The characteristic equation of the error is

$$\det[s\mathbf{I} - (\mathbf{A} - \mathbf{L}\mathbf{C})] = 0 \tag{5.88}$$

The vector $\mathbf{L}$ can be selected to make $\mathbf{A} - \mathbf{L}\mathbf{C}$ stable and reasonably quick eigenvalues to make $\tilde{\mathbf{x}}$ to be decayed to zero without the control input $\boldsymbol{u}$ and initial conditions. This means that $\hat{\mathbf{x}}(t)$ will definitely converge to $\mathbf{x}(t)$.

Errors in the above model of the plant (**A**, **B**, **C**) may introduce additional errors to the state estimation in Eq. (5.87). To solve this issue, the vector **L** can be selected to make those errors acceptably small. In fact, most estimators are usually electrical or electronic unit calculating the estimated state based on Eq. (5.87).

Similarly, the selection process of **L** is identical with those steps used in selecting gains **K** in the control law design. Suppose we select the desired location of the estimator error poles as $s_i = \beta_1, \beta_2, \ldots, \beta_n$, then the desired estimator characteristic equation is

$$\alpha_e(s) = (s - \beta_1)(s - \beta_2)\ldots(s - \beta_n) \tag{5.89}$$

In this way, we can solve for **L** by comparing coefficients in Eqs. (5.88) and (5.89).

A point to be noted is that this selection process is almost never done by hand; instead, we need to use some MATLAB functions discussed in the last section, such as `acker()` and `place()`, with some modifications. The transpose of Eq. (5.88) is

$$\det\left[s\mathbf{I} - \left(\mathbf{A}^{\mathbf{T}} - \mathbf{C}^{\mathbf{T}}\mathbf{L}^{\mathbf{T}}\right)\right] = 0 \tag{5.90}$$

Now we can compute **L** to get estimator poles at the desired location **p** by using MATLAB function `acker()` or `place()`, such as

`L = acker(A', C', p)'` or `L = place(A', C', p)'`.

For a SISO system, the solution for **L** will be unique if the system is observable, which means that we can get or deduce all modes of the system by monitoring the sensor outputs. For an unobservable system, some modes or subsystems have no effect on the output.

Regularly, the selection of the estimator poles that determine **L** is selected to be a factor of 2 or 6 times faster than the controller poles to ensure a faster decay of the estimator errors compared with the desired response. This will enable the controller poles to dominate the system response. However, if the sensor noise is too large, a slower estimator pole should be selected to reduce the noise's disturbance to the system. The cost for this is that the system bandwidth will be also reduced.

To get the optimal estimator roots, a trade-off between the fast and slow roots should be made by using the optimal estimation theory. In fact, a balance between the process noise intensity R_w and the sensor noise intensity R_v can be made, and both of them are scalars for a SISO system. We can use the ratio of both noises to evaluate the effects of the noises. If we fix R_v to 1, then we can vary R_w only to change this ratio. A MATLAB function `kalman()` is a good way to help us to do this kind of optimal estimation process.

The syntax of this function is

```
L = kalman(sys, Rw, Rv)
```

For example, we still use the plant shown in Eq. (5.84) with related state variables and matrixes to calculate or estimate the optimal estimator by using this function. Use the following scripts to perform this estimation:

```
A = [0 1; -1 -2];
B = [0; 1];
C = [1 0];
sys = ss(A, B, C, 0);
Q = 10; R = 1;
[kest,L,P] = kalman(sys, Q, R)
%est = estim(sys,L)
```

The running result is (we assume that R_w is 10 times larger than that of R_v)

```
L =  0.9382     0.4401
P =  0.9382     0.4401
     0.4401     2.2315
```

5.4.6 Compensation—Combined Control and Estimation

Now it is the time for us to combine the control law (5.81) with the estimator (5.85) together to complete our design. At this moment, we ignore the command input r and will discuss this in the next section. By implementing the control law with the estimated state elements, our design equations look like

$$\begin{aligned} \dot{\hat{\mathbf{x}}} &= (\mathbf{A} - \mathbf{BK} - \mathbf{LC})\hat{\mathbf{x}} + \mathbf{L}y \\ u &= -\mathbf{K}\hat{\mathbf{x}} \end{aligned} \tag{5.91}$$

These equations define what we did previously, steps 1 and 2, in order to get the control output u based on the given output y. The actual connection and configuration for this combination are shown in Fig. 5.67. The roots for this resulted closed-loop system are combination of roots of controller and roots of the estimator. The poles and zeros of the compensator can be obtained from Eq. (5.91). An interesting thing is that both controller and the estimator can be designed separately;

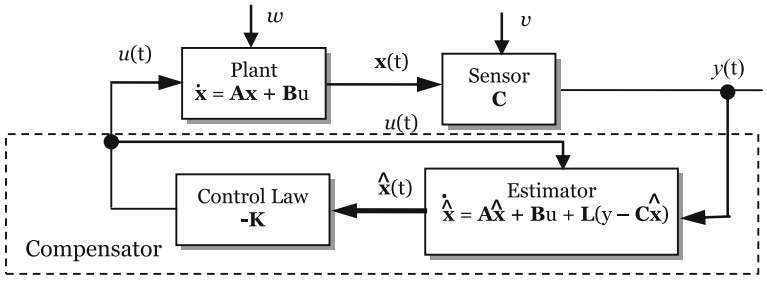


Fig. 5.67 Combination of the estimator and the controller

however, when both are combined together, the resulted roots are just kept unchanged.

If we want to see the effect of using the estimated state variables on the system dynamics, we need to use estimated $\hat{\mathbf{x}}$ in place of $\mathbf{x}$ in the system state equation as

$$\dot{\mathbf{x}} = \mathbf{A}\mathbf{x} - \mathbf{B}\mathbf{K}\hat{\mathbf{x}} \quad \text{or} \quad \dot{\mathbf{x}} = \mathbf{A}\mathbf{x} - \mathbf{B}\mathbf{K}(\mathbf{x} - \tilde{\mathbf{x}}) \tag{5.92}$$

The entire system dynamics can be obtained by combining Eqs. (5.92) and (5.87), as

$$\begin{bmatrix} \dot{\mathbf{x}} \\ \dot{\tilde{\mathbf{x}}} \end{bmatrix} = \begin{bmatrix} \mathbf{A} - \mathbf{B}\mathbf{K} & \mathbf{B}\mathbf{K} \\ 0 & \mathbf{A} - \mathbf{L}\mathbf{C} \end{bmatrix} \begin{bmatrix} \mathbf{x} \\ \tilde{\mathbf{x}} \end{bmatrix}$$

The characteristic equation of this closed-loop system is

$$\det \begin{bmatrix} s\mathbf{I} - \mathbf{A} + \mathbf{B}\mathbf{K} & -\mathbf{B}\mathbf{K} \\ 0 & s\mathbf{I} - \mathbf{A} + \mathbf{L}\mathbf{C} \end{bmatrix} = 0 \tag{5.93}$$

This results in

$$\det[s\mathbf{I} - \mathbf{A} + \mathbf{B}\mathbf{K}] \det[s\mathbf{I} - \mathbf{A} + \mathbf{L}\mathbf{C}] = \alpha_c(s)\alpha_e(s) = 0 \tag{5.94}$$

The transfer function of the compensator can be obtained by inspecting Eq. (5.74) and substituting in the similar matrices from Eq. (5.94). This produces

$$D(s) = \frac{U(s)}{Y(s)} = -\mathbf{K}[s\mathbf{I} - \mathbf{A} + \mathbf{B}\mathbf{K} + \mathbf{L}\mathbf{C}]^{-1}\mathbf{L} \tag{5.95}$$

For example, for the system plant shown in Eq. (5.84), with state variable equations

$\mathbf{A} = \begin{bmatrix} 0 & 1 \\ -1 & -2 \end{bmatrix}$; $\mathbf{B} = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$; $\mathbf{C} = [1 \quad 0]$ and $\mathbf{D} = 0$, where $2\xi\omega_0 = 2$ and $\omega_0 = 1$.

The specifications for the control law design are still the same as closed-loop roots with ω_n = 3 rad/s and ξ = 0.8. The control gains are $\mathbf{K}$ = `8.0000 and 2.8000`, which are same as we obtained before.

Now if we select the estimator roots are at ω_n = 5 rad/s and ξ = 0.5, the desired characteristic equation is

$$s^2 + 2\xi\omega_n s + \omega_n^2 = s^2 + 5s + 25$$

Use the following MATLAB scripts to obtain the estimator feedback gain matrix $\mathbf{L}$:

```
A = [0 1; −1 −2];
B = [0; 1];
C = [1 0];
Wn = 5;
Ze = 0.5;
p = roots([1 2*Wn*Ze Wn^2]);
L = acker(A', C', p)'
```

The results are $\mathbf{L}$ = 3.0 and 18.0.

Then, use the following MATLAB scripts to get the inverse of $[s\mathbf{I} - \mathbf{A} + \mathbf{BK} + \mathbf{LC}] = \mathbf{H}$.

```
A = [0 1; -1 -2];
B = [0; 1];
C = [1   0];
K = [8  2.8];
L = [3; 18];
I = eye(2);
s = tf('s');
H = inv(s*I - A + B*K + L*C)
```

The $\mathbf{H}$ is a 2 × 2 matrix. Using $\mathbf{K}$ and $\mathbf{L}$ matrix values and placing them into Eq. (5.95), we can get the transfer function of the controller as

$$G(s) = -74.4\frac{(s+2.5)}{s^2+7.8s+41.4} = -74.4\frac{(s+2.5)}{(s+3.9-j5.1)(s+3.9+j5.1)}$$

This looks like a phase lead compensator.

5.4.7 *Introduce the Reference Input*

In the last section, we are exactly following our design steps, step 3, by combining the control law obtained in step 1 and the estimator obtained from step 2 together to get the desired controller or compensator. This kind of compensator is exactly a regulator, which means that the characteristic equations of the control and estimator can be designed to obtain satisfactory transient response to effectively reject some disturbances such as white noise coming from the process or from the sensors. The shortcoming of this design is that we cannot make sure whether this regulator can follow an input command perfectly or smoothly to get our design objective. Fortunately, this can be done by proper introduction of the reference input to the system equation.

Two possible ways to introduce the command input r into the closed-loop system are shown in Fig. 5.68a, b, respectively. The only difference between these two configurations is the location of the compensator in the loop; either in the feedback or feedforward path.

One of the easy ways to introduce a command input is to subtract the output y from the input command r, which is very similar to closed-loop design for a given plant as we did before. This kind of introduction of input to the system is shown in

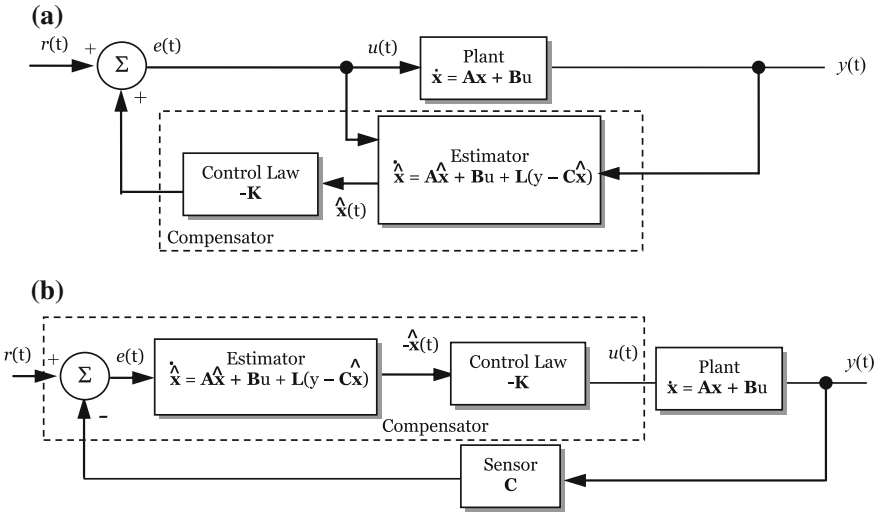


Fig. 5.68 Two possible locations of the introduction of command input

Fig. 5.68b. An advantage of using this approach is that the step command can directly enter into the estimator to cause an estimation error to decay with the estimator dynamic characteristics in addition to the response corresponding to the control poles.

Another possible way is to enter the command r directly into the system plant and estimator in an identical way as shown in Fig. 5.68a. Because a step command affects the plant and estimator in the same way, both respond identically and no estimator error is induced. The result is that there are no estimator error characteristics in the response and the total response is composed of controller characteristics only.

Now we can get a clear picture about the system plant and controller based on state variables and state-space equations shown below:

$$\left.\begin{aligned}\dot{\mathbf{x}} &= \mathbf{A}\mathbf{x} + \mathbf{B}u\\ y &= \mathbf{C}\mathbf{x}\end{aligned}\right\}\text{Plant}$$

$$\left.\begin{aligned}\dot{\hat{\mathbf{x}}} &= (\mathbf{A} - \mathbf{B}\mathbf{K} - \mathbf{L}\mathbf{C})\hat{\mathbf{x}} + \mathbf{L}y\\ u &= -\mathbf{K}\hat{\mathbf{x}}\end{aligned}\right\}\text{Controller}$$

These are the representations of the system plant (process) and the controller (compensator) in the state-space domain.

5.4.8 Integral Control

As we discussed in Sect. 5.4.6 with an example compensator design, the result of the compensator is a phase lead style. Generally, state-space design always generates a compensation composing of proportional and derivative feedback, but it is difficult to produce an integral control unless some special operations are taken in the design process.

The integral control can be generated by augmenting the state vector with the desired integral x_{I}, which obeys the differential equation

$$\dot{x}_{\mathrm{I}} = \mathbf{C}\mathbf{x}(= y) - r(= e)$$

thus we have

$$x_{\mathrm{I}} = \int e \, \mathrm{d}t$$

This equation is augmented to the state equation (5.66) and it becomes

$$\begin{bmatrix} \dot{x}_{\mathrm{I}} \\ \dot{\mathbf{x}} \end{bmatrix} = \begin{bmatrix} 0 & \mathbf{C} \\ 0 & \mathbf{A} \end{bmatrix} \begin{bmatrix} x_{\mathrm{I}} \\ \mathbf{x} \end{bmatrix} + \begin{bmatrix} 0 \\ \mathbf{B} \end{bmatrix} u - \begin{bmatrix} 1 \\ 0 \end{bmatrix} r$$

The feedback law becomes

$$u = -[\mathrm{K_I} \quad \mathbf{K}] \begin{bmatrix} x_{\mathrm{I}} \\ \mathbf{x} \end{bmatrix} \quad \text{or} \quad u = -\mathbf{K} \begin{bmatrix} x_{\mathrm{I}} \\ \mathbf{x} \end{bmatrix}$$

For example, suppose we have a system

$$\frac{Y(s)}{U(s)} = \frac{1}{s+2}$$

Using the following MATLAB scripts, we can get the related state variable matrices:

```
s = tf('s');
G = 1/(s+2);
sys = ss(G)
```

The running results are $\mathbf{A} = -2$, $\mathbf{B} = 1$, $\mathbf{C} = 1$, and $\mathbf{D} = 0$.

The specifications are closed-loop roots at $\omega_{\mathrm{n}} = 5$ and $\xi = 0.5$, which is equivalent to a desired characteristic equation as

$$s^2 + 5s + 25 = 0$$

The augmented system description looks like

$$\begin{bmatrix} \dot{x}_I \\ \dot{\mathbf{x}} \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 0 & -2 \end{bmatrix} \begin{bmatrix} x_I \\ \mathbf{x} \end{bmatrix} + \begin{bmatrix} 0 \\ 1 \end{bmatrix} u$$

Use the following MATLAB scripts to get the gain **K** matrix:

```
A = [0 1; 0 -2];
B = [0; 1];
Wn = 5;
Ze = 0.5;
p = roots([1 2*Wn*Ze Wn^2]);
K = acker(A, B, p)
```

The resulted gain matrix is $\mathbf{K} = [25 \quad 3]$.

The closed-loop control system is shown in Fig. 5.69. A point to be noted is that if a reference command $r(t)$ is introduced, the system error, $e = r - y$, should be integrated and not the output alone.

Finally, before we can finish this chapter, we try to design a PID controller to build a closed-loop control system for our example DC motor.

5.4.9 Controllability and Observability

In Sect. 5.4.3, we introduced the observability and controllability. We will provide more details about these two terminologies in this part.

The controllability is defined as follows: A system described by the state variable matrices (**A**, **B**) can be considered to be controllable if there exists an unconstrained control **u** that can transfer any initial state **x**(0) to any other desired location **x**(t). In other words, to determine whether a system $\dot{x} = Ax + Bu$ is controllable or not, we can check the algebraic condition or the rank of

$$\text{rank}\begin{bmatrix} \mathbf{B} & \mathbf{AB} & \mathbf{A}^2\mathbf{B} & \dots & \mathbf{A}^{n-1}\mathbf{B} \end{bmatrix} = n \tag{5.96}$$

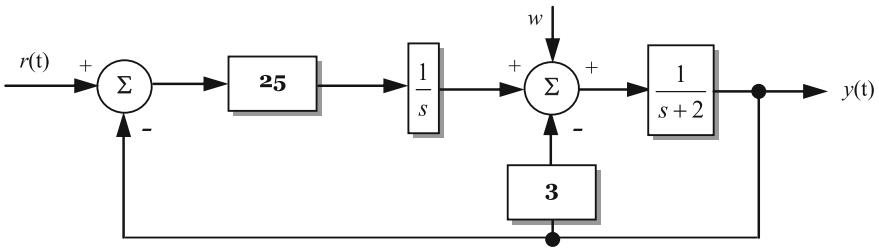


Fig. 5.69 An integral control example

For a SISO system, a controllability matrix $\mathbf{P}_c$ can be defined in terms of $\mathbf{A}$ and $\mathbf{B}$ as

$$\mathbf{P}_c = \begin{bmatrix} \mathbf{B} & \mathbf{AB} & \mathbf{A^2B} & \dots & \mathbf{A}^{n-1}\mathbf{B} \end{bmatrix} \tag{5.97}$$

where $\mathbf{P}_c$ is a $n \times n$ matrix, and if the determinant of $\mathbf{P}_c$ is nonzero, the system is said to be controllable.

Another method to determine whether a system is controllable or not is to sketch the state variable block diagram in the integrator form and check whether the control signal or input u has a path to each state variable. If a path to each state exists, the system is controllable.

For example, a system has state variable matrix $\mathbf{A}$, $\mathbf{B}$, and $\mathbf{C}$ as

$$A = \begin{bmatrix} -2 & 0 \\ d & -4 \end{bmatrix}; \quad B = \begin{bmatrix} 1 \\ 0 \end{bmatrix}; \quad C = \begin{bmatrix} 1 & 0 \end{bmatrix}.$$

The controllability matrix $\mathbf{P}_c$ is defined as

$$P_c = \begin{bmatrix} B & AB \end{bmatrix} = \left\{ \begin{bmatrix} 1 \\ 0 \end{bmatrix} \quad \begin{bmatrix} -2 & 0 \\ d & -4 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \end{bmatrix} \right\} = \begin{bmatrix} 1 & -2 \\ 0 & d \end{bmatrix}$$

The determinant of $\mathbf{P}_c$ is equal to d. If the $d \neq 0$, the system is controllable.

The observability is defined as if the output of a system has a component due to each state variable. In other words, each state variable can be reflected on the output or provides a part of contributions to the output.

A system is observable if, and only if, there is a finite time T such that the initial state $\mathbf{x}(0)$ can be determined from the observation history $y(t)$ given the control input $u(t)$.

For example, we have a system represented by the state variable matrices, $\dot{x} = Ax + Bu$ and $y = Cx$. The system is said to be observable as long as the determinant of the observable matrix $\mathbf{Q}_o$ is nonzero, where $\mathbf{Q}_o$ is a $n \times n$ matrix.

$$Q_o = \begin{bmatrix} C & CA & CA^2 & \dots & CA^{n-1} \end{bmatrix}^{\mathrm{T}} \tag{5.98}$$

Another way is to sketch the block diagram with the integrator level to check whether each state variable has a path to the output $y(t)$.

For example, we still use the system that has state variable matrix $\mathbf{A}$, $\mathbf{B}$, and $\mathbf{C}$ as

$$A = \begin{bmatrix} -2 & 0 \\ d & -4 \end{bmatrix}; \quad B = \begin{bmatrix} 1 \\ 0 \end{bmatrix}; \quad C = \begin{bmatrix} 1 & 0 \end{bmatrix}.$$

The observability matrix $\mathbf{Q_o}$ is defined as

$$Q_o = [C \quad CA]^T = \left(\begin{bmatrix} 1 & 0 \end{bmatrix} \quad \begin{bmatrix} 1 & 0 \end{bmatrix} \begin{bmatrix} -2 & 0 \\ d & -4 \end{bmatrix} \right)^T = \begin{bmatrix} 1 & 0 \\ -2 & 0 \end{bmatrix}$$

The determinant of observability matrix $\mathbf{Q_o}$ is equal to 0, so the system is not observable.

5.4.10 Pole Placement Using State Feedback Control

In this section, we try to use state variable feedback to determine the desired pole locations of a closed-loop transfer function $T(s)$. This design is based on the feedback of the state variables as

$$\mathbf{u} = -\mathbf{H}\boldsymbol{x} \tag{5.99}$$

where the $\mathbf{H}$ is the feedback matrix and the $\boldsymbol{x}$ is the state variables. When using this method, the roots of the characteristic equation are placed where the transient response meets the desired specifications.

For an SISO system, the Ackermann's formula is a convenient tool to determine the state variable feedback matrix $\mathbf{H} = [h_1, h_2, \ldots h_n]$.

For a given desired characteristic equation, $q(s) = s^n + \alpha_1 s^{n-1} + \ldots + \alpha_n$. The related state feedback gain matrix is

$$\mathbf{H} = [0, \quad 0, \quad \ldots, \quad 1]\mathbf{P}_c^{-1} q(\mathbf{A}) \tag{5.100}$$

where $q(\mathbf{A}) = A^n + \alpha_1 A^{n-1} + \cdots + \alpha_{n-1} A + \alpha_n I$, and $\mathbf{P}_c$ is the controllability matrix.

For example, for a given system $G(s) = \frac{1}{s(s+1)}$, determine the feedback gain to place the closed-loop poles at $s = -2 \pm j2$. Thus, the characteristic equation is $q(s) = s^2 + 4s + 8; \alpha_1 = 4, \alpha_2 = 8$. The state variable matrices are

$$A = \begin{bmatrix} 0 & 1 \\ 0 & -1 \end{bmatrix};\ B = \begin{bmatrix} 0 \\ 1 \end{bmatrix};\ C = [1 \quad 0].\ \mathbf{P}_c = [\mathrm{B} \quad \mathrm{AB}] = \begin{bmatrix} 0 & 1 \\ 1 & -1 \end{bmatrix};$$

$$\mathbf{P}_c^{-1} = \frac{1}{-1} \begin{bmatrix} -1 & -1 \\ -1 & 0 \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}.$$

Therefore

$$q(A) = A^2 + \alpha_1 A + \alpha_2 I = \begin{bmatrix} 0 & 1 \\ 0 & -1 \end{bmatrix}^2 + 4 \begin{bmatrix} 0 & 1 \\ 0 & -1 \end{bmatrix} + 8 \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} = \begin{bmatrix} 8 & 3 \\ 0 & 5 \end{bmatrix}.$$

$$\mathbf{H} = [0 \quad 1] \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} 8 & 3 \\ 0 & 5 \end{bmatrix} = [8 \quad 3].$$

5.5 Case Study: Design and Build a PID Controller for a DC Motor System

In Sect. 4.7.4 in Chap. 4, we have provided detailed discussions about the calibration of our example DC motor **Mitsumi 448 PPR Motor**. The purpose of this calibration is to obtain a clear linear relationship between the output values of the TM4C123GH6PM PWM module and the rotation speed of our example DC motor. Based on that calibration result, we performed the identification process for that DC motor to get its dynamic model in Sect. 4.7.5. The identified dynamic model is

$$G(s) = \frac{3.776}{(1+0.56s)} e^{-0.09s} \cong \frac{6.74}{(s+1.79)}$$

We also utilize MATLAB® PID Tuner Tools and Simulink to tune and perform simulation for this DC motor closed-loop system to obtain the optimal control and output performances in Sects. 5.3.6.2 and 5.3.6.3 in this chapter. Now it is the time for us to design a real or actual PID controller to control this DC motor to get an optimal and practical closed-loop control system with TM4C123GH6PM MCU.

The current PWM output control signal is estimated based on the linear equation calculated by using the **CalibEncoder** project in Sect. 4.7.4 and output to the motor control system.

The equation **ES = k** × **PWM + b** can be used to estimate the slope **K** and **b**. Based on the running result of the project **CalibEncoder**, we developed in that section, the estimated average value for the slope **K** is **0.45** and **b** is about 0. Therefore, the speed feedback transfer function is a constant and its value is **0.45**. Based on this equation, for a given **PWM** value, the feedback encoder speed value **ES** should be **ES = 0.45** × **PWM**.

5.5.1 The Continuous and Discrete PID Closed-Loop Control System

A professional and traditional closed-loop control system is a so-called Proportional–Integral–Differential (PID) control system. A block diagram of this kind of PID control system is shown in Fig. 5.70.

The following symbols are used in this continuous closed-loop control system:

- **E(s)**—Laplace transformation of the error signal $e(t) = e(t_n) - e(t_{n-1})$.
- **D(s)**—Laplace transformation of the PID controller $D(t) = K_p + K_i \int dt K_d d/dt$.

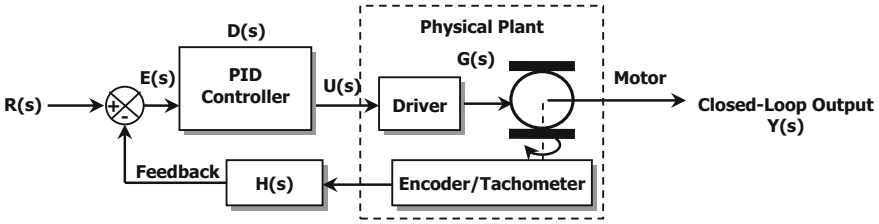


Fig. 5.70 The block diagram of PID closed-loop control system

- **U(s)**—Laplace transformation of the controller output $U(t) = K_p e(t) + K_i \int e(t)dt + K_d de(t)/dt$.
- **G(s)**—Laplace transformation of the physical plant $G(t)$ (including the driver and motor).
- **H(s)**—Laplace transformation of the feedback system $H(t)$.

Regularly, three transformations are important to our system design: the $D(s)$, $G(s)$, and $H(s)$.

The $D(s)$ is the Laplace transformation of PID controller $D(t) = K_p + K_i \int dt + K_d d/dt$. In the frequency domain, this controller becomes $D(s) = K_p + K_i/s + sK_d$.

The $G(s)$ is the Laplace transformation of the physical plant $G(t)$. Basically, most motor systems can be considered or approximated as linear systems, and their transformation functions can be represented by the first- or the second-order differential equations in the form of $K/(1 + \tau s)$ and τ is the time constant.

In most motor control systems, the feedback transformation function $H(s)$ can be considered as a linear coefficient K_H used to convert the encoder/tachometer position/velocity feedback output to the corresponding input variable.

Based on Fig. 5.70, the overall transfer function for the complete closed-loop control system can be derived as follows:

- The error $E(s) = S(s) - H(s)Y(s)$.
- The closed-loop output $Y(s) = D(s)G(s)E(s) = D(s)G(s)[S(s) - H(s)Y(s)]$.
- So the $Y(s) = D(s)G(s)S(s) - D(s)G(s)H(s)Y(s)$.

Finally, the overall closed-loop transfer function is equal to

$$\frac{Y(s)}{S(s)} = \frac{D(s)G(s)}{1 + D(s)G(s)H(s)} = \frac{sK_0 + K}{s(1 + \tau s)}$$

This is a one-zero and two-pole system with zero $= -K/K_0$ and two poles, $s_1 = 0$ and $s_2 = -1/\tau$. In most actual applications, the plant transfer function $G(s)$ can be simplified to

$$G(s) = \frac{K}{1 + \tau s}$$

The open-loop transfer function $D(s)G(s)H(s)$ is an important function used to design the complete closed-loop control system.

The regular PID control system design method and procedure are as follows:

1. Collect the encoder or tachometer feedback values based on a sequence of PWM outputs applied to a motor system (including the driver and motor) when the motor is working in an open-loop condition (as we did in Sect. 4.7.4).
2. Identify the dynamic model of the plant $G(s)$ based on the collected encoder or tachometer values and the corresponding PWM output values applied on the motor system in step 1 by using the MATLAB® Identification Toolbox™.
3. Based on the identified dynamic model in step 2, use any design method, such as root locus, Bode plot, state space, or MATLAB® Control System Toolbox™ to design the desired PID controller.
4. After the PID controller is designed, use MATLAB® PID Tuner Tool or Simulink® method to perform tuning and simulation to obtain the optimal PID parameters by tuning these control gain parameters.
5. Apply the simulated optimal PID parameters on our real motor control system in the following ways in our control software:
   - First get the position or velocity errors $e(t)$ between the setup values and the encoder or tachometer feedback values via encoder or tachometer.
   - Use PID equation with three control gain parameters to get the control output $u(t)$ in terms of the input error $e(t)$ obtained from the last step.
   - Output this control output $u(t)$ with PWM module to the motor system.
   - Loop back to the first step.

Since computer is a digital electronic device and only the digital or discrete variables and functions can be applied and implemented in a computer, therefore, the software control program applied on any computer must also be digital program composed of digital codes. When using the above steps to build the actual PID control software, the following continuous variables must be converted to the discrete variables to match the digital controller:

1. The discrete position error $e(n) = r(n) - h(n)y(n) = r_n - Ky(n) = r - Ky_n$ (if r is a constant and n is an increment discrete number).
2. The discrete velocity error $\Delta e(n)/\Delta n = (e(n) - e(n-1))/[n - (n-1)] \approx e(n) - e(n-1) = e_n - e_{n-1}$.
3. The discrete integration $= \Sigma e(t_i) = e(n_1) + e(n_2) + \cdots + e(n_n) = e_1 + e_2 + \cdots + e_n = \Sigma e_i$.
4. The discrete differential is similar to velocity error, which is $e_n - e_{n-1}$.
5. The controller discrete output $U_n = K_p e_n + K_i \Sigma e_i + K_d(e_n - e_{n-1})$.

In the following sections, we will use these steps and procedures to build our PID controller for this motor control system.

5.5.2 *Build the Control Software to Implement the PID Controller*

Open the Keil® µVersion®5 IDE to create a new project named **PID-Control** under the folder **C-M Control Class Projects\Chapter 5** and add a new C source file **PID-Control.c** into the project.

Enter the first part codes shown in Fig. 5.71 into this source file. Let's have a closer look at this piece of codes to see how it works.

(1) The codes in lines 4 ~ 9 are used to declare some system header files used in this project.
(2) Some system and user-defined macros are declared in lines 10 ~ 17. These macros include the upper and lower bounds of the PWM values **PWMMAX** and **PWMMIN**, the transfer function or the slope between the PWM and the feedback encoder speed value **HS**, and other system macros, such as **PHA0** and **PHB0** phase inputs, GPIO Ports D and F base and commit registers.
(3) Two user-defined functions, **InitPWM()** and **InitQEI()**, are declared in lines 18 and 19.
(4) The **main()** program starts at line 20. Some local variables, **pw** and **es**, represented the output PWM value and encoder speed value **ES**, **upper,** and **motor[]** array are used to define the upper bound of the loop and the collected encoder speed values, and these are declared at the beginning of this program in lines 22.
(5) In line 23, the input velocity error **e[]**, error rate **de,** and integral error **ie**, as well as PID parameters, **Kp**, **Ki,** and **Kd**, are declared and defined.
(6) The system clock is defined in line 24. The GPIO Ports B, D, and F are clocked and enabled in line 25.
(7) The codes in lines 26 ~ 31 are used to configure the related GPIO pins to be digital output or input pins to enable them to work as **PhA** and **PhB** or LED-driven pins.
(8) In lines 32 and 33, both user-defined functions, **InitPWM()** and **InitQEI()**, are called to initialize and configure the **PWM1** and **QEI0** modules to make them ready to output desired PWM values and receive rotary encoder feedback speed values to perform the closed-loop control function.
(9) An infinitive **while()** loop is started at line 34 to enable the program to repeatedly output the desired PWM value to drive the motor to rotate in a constant speed.

    The codes in lines 36 and 37 are used to enable the **PWM1_2B** PWM output pin and the **PWM1** module.

(10) A selected desired PWM value, which is 25%, is assigned to the **s** variable that is to be sent to the DC motor to make the motor rotate in line 38. This PWM value **1000** can be considered as a desired velocity input setup value **450 PPR** for the

```
1  //*****************************************************************************
2  // PID-Control.c - Motor PID Closed-Loop Control Program – QEI0
3  //*****************************************************************************
4  #include <stdint.h>
5  #include <stdbool.h>
6  #include "driverlib/sysctl.h"
7  #include "driverlib/gpio.h"
8  #include "driverlib/qei.h"
9  #include "TM4C123GH6PM.h"
10 #define  PWMMAX                3999
11 #define  PWMMIN                5
12 #define  HS                    2.22                  // HS = 1/K = 1/0.45 = 2.22
13 #define  GPIO_PD6_PHA0         0x00031806
14 #define  GPIO_PD7_PHB0         0x00031C06
15 #define  GPIO_PORTD_BASE       0x40007000            // GPIO Port D
16 #define  GPIO_PORTF_CR_R       (*((volatile uint32_t *)0x40025524))
17 #define  GPIO_PORTD_CR_R       (*((volatile uint32_t *)0x40007524))
18 void InitPWM(void);
19 void InitQEI(void);
20 int main(void)
21 {
22    uint32_t pw, es, index, upper = 1000, s, n = 0, motor[100];
23    double ie = 0, de, e[2], Kp = 0.75, Ki = 2.118, Kd = 0;
24    SysCtlClockSet(SYSCTL_SYSDIV_25|SYSCTL_USE_PLL|SYSCTL_XTAL_4MHZ|SYSCTL_OSC_MAIN);
25    SYSCTL->RCGC2 = 0x2A;                              // enable & clock GPIO Ports B, D & F
26    GPIOB->DEN = 0xF;                                  // enable PB3 ~ PB0 as digital function pins
27    GPIOB->DIR = 0xF;                                  // configure PB3 ~ PB0 as output pins
28    GPIOD->DEN = 0xC0;                                 // enable PD7 ~ PD6 as digital function pins
29    GPIOD->DIR = ~0xC;                                 // configure PD7 ~ PD6 as input pins
30    GPIOF->DEN = 0xF;                                  // enable PF3 ~ PF0 as digital function pins
31    GPIOF->DIR = 0xF;                                  // configure PF3 ~ PF0 as output pins
32    InitPWM();                                         // configure PWM module 1
33    InitQEI();                                         // configure QEI module 0
34    while(1)
35    {
36       PWM1->_2_CTL = 0x1;                             // enable PWM1_2B or M1PWM5
37       PWM1->ENABLE = 0x20;
38       s = 1000;                                       // s = 1000 makes es = 0.45*PWM = 0.45*1000 = 450 ppr
                                                         // the 450 ppr can be considered as a setup input velocity
39       PWM1->_2_CMPB = s;                              // send desired PWM value or 450 ppr to motor
40       e[1] = s;
41       for (index = 0; index < upper; index++)         // output a square wave form to motor
42       {
43          es = QEI0->SPEED;                            // get the current encoder speed value
44          e[0] = s - es*HS;                            // convert speed to PWM value
45          de = e[1] - e[0];                            // get differential error
46          ie = ie + e[0];                              // get integral error
47          pw = (uint32_t)(Kp*e[0] + Ki * ie + Kd * de);// calculate the PID control value
48          e[1] = e[0];                                 // save the next error
49          PWM1->_2_CMPB = pw;                          // output the PID control value to the motor
50          SysCtlDelay(5);
51          if (n < 100)  { motor[n] = es; }             // collect the first 100 encoder speed values
52          n++;
53       }
54       PWM1->_2_CMPB = 0;                              // 0 to motor to stop motor
55       PWM1->_2_CTL = 0x0;                             // disable PWM1_2B or M1PWM5
56       PWM1->ENABLE &= ~0x20;
57       for (index = 0; index < upper; index++)         // stop the motor for a period of time
58          SysCtlDelay(10);
59    }
60 }
```

Fig. 5.71 The first part codes for the project PID control

target motor rotating velocity since $\mathbf{ES = K \times PWM = 0.45 \times PWM = 0.45 \times 1000 = 450}$.

(11) The desired PWM value **1000** or the target motor velocity **450** PPR stored in the variable **s** is sent to the motor to make the motor rotate in line 39, and this

setup motor velocity is sent to the first velocity error **e[1]** to be reserved in line 40.

(12) Starting at line 41, a **for** loop is used to repeatedly send out this desired motor setup velocity value **225 PPR** that is equivalent to **PWM = 500**, and get the feedback encoder velocity value **es** in line 43, then compare both values to calculate the velocity error **e[0]** in line 44, error rate **de** in line 45, and integral error **ie** in line 46.

(13) In line 44, in order to get the velocity error, the feedback encoder velocity is converted to the corresponding PWM value by multiplying **HS = 1/0.45 = 2.22** since **es = 0.45** $\times$ **PWM**, and **PWM = es/0.45 = es** $\times$ **2.22**. The simulated PID control gains **Kp**, **Ki,** and **Kd** are used to estimate the optimal PWM output value in line 47.

(14) In line 48, the current velocity error **e[0]** is sent to the next error **e[1]** to be reserved.

(15) The calculated optimal PID control out is sent to the motor to try to rotate the motor in a constant setup speed in line 49. Then, the system is delayed by some time to make the motor rotating speed stable in line 50.

(16) The codes in lines 51 and 52 are used to collect the first 100 motor encoder feedback velocity values and save them into the data array **motor[100]**. These velocity values can be used to analyze the actual motor step response later with the MATLAB® **plot()** function.

(17) The codes in lines 54 and 56 are used to enable a Low level applied on the motor driver to stop the DC motor and disable the PWM1 module.

(18) Another **for** loop is used to make this Low output with a certain period of time in lines 57 and 58.

The second part codes for this project contain the codes for the **InitPWM()** and **InitQEI()** functions. For the detailed codes about these two functions, refer to the project **CalibEncoder** built in Sect. 4.7.4 in Chap. 4 since both of them are identical with those functions defined in this project. Also refer to Sect. 4.7.4 in Chap. 4 to set up the environments for this project, build, and run the project if everything is fine. As the project runs, one can find that the motor is rotating periodically from Low to High and from High to Low.

To confirm the performance of the PID controller applied to this closed-loop motor control system, one can use the 100 collected encoder velocity values stored in the **motor[]** data array.

One needs to record and convert them from hexadecimal to the decimal values, and then create a MATLAB® script and put these data into a data array or a vector to plot them in the time domain to check its performance.

A piece of example codes is shown in Fig. 5.72 and it is saved as a MATLAB® Script file named **PlotMotorPIDStep.m**. When running this MATLAB® script file, the step response of this PID closed-loop motor control system is plotted in the time domain, as shown in Fig. 5.73.

```
% plot the PID closed-loop motor step response - 100 samples - PlotMotorPIDStep.m
t = 1:100;
y = [0; 0; 0; 0; 0; 49; 49; 49; 179; 179; 179; 179; 267; 267; 267; 267; 326; 326; 326; 364; 364; 364; 364;
     388; 388; 388; 388; 405; 405; 405; 418; 418; 418; 418; 426; 426; 426; 426; 432; 432; 432; 435; 435; 435;
     435; 440; 440; 440; 440; 441; 441; 441; 443; 443; 443; 443; 443; 443; 443; 443; 444; 444; 444; 444; 444;
     444; 444; 445; 445; 445; 445; 446; 446; 446; 450; 450; 450; 450; 449; 449; 449; 449; 449; 449; 449; 449;
     447; 447; 447; 448; 448; 448; 448; 447; 447; 447; 447; 447; 447; 447];
plot(t, y);
grid;
xlabel('Numer of Samples');
ylabel('Motor Output Speed (PPR)');
title('PID Closed-Loop Motor Step Response');
```

Fig. 5.72 The codes to plot the motor step response

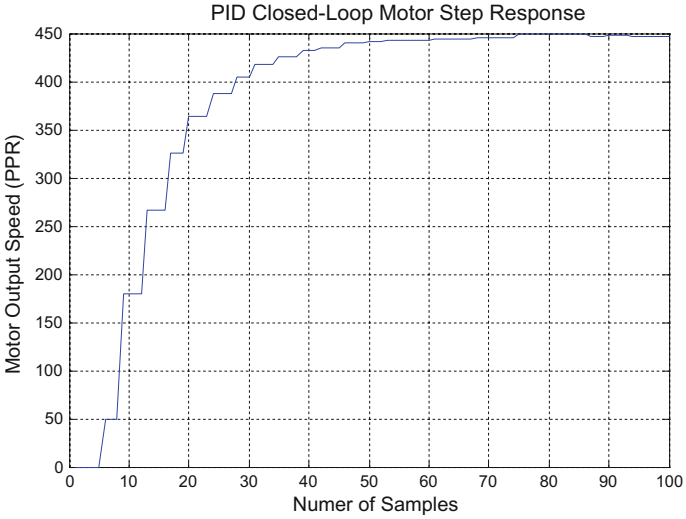


Fig. 5.73 The step response of the actual closed-loop motor control system

5.6 Summary

The main topic of this chapter is concentrated on the PID control system design and analysis. Three popular methods, root locus, bode plot, and state space, are introduced and discussed in details with quite a few examples.

Starting from Sect. 5.2, the root locus strategy is discussed with some special properties used in this method. The effects of poles and zeros on time-domain specifications are discussed in detail. Then, the sketching of root locus by manually is introduced. Two typical compensators, phase lead and phase lag, are introduced and analyzed with some examples. Some popular and powerful computer-aided design and analysis tools with root locus method, such as MATLAB root locus design functions and SISOTool, are discussed in detail.

In Sect. 5.3, the PID controller design with Bode plot frequency response method is introduced and analyzed. Some useful and important features provided by this method are emphasized and highlighted, and these include

- Bode plot techniques,
- Stability and margins,
- Sketching Bode plot, and
- The gain and phase frequency relationship.

Three popular computer-aided design tools, SISOTool, PIDTuner, and MATLAB Simulink, are discussed with some actual example systems. The Nyquist stability criterion is also introduced and discussed with some useful MATLAB functions.

The PID controller design with state-space method is introduced in Sect. 5.4. Some important and related properties and features are discussed in detailed, which include

- State-space variable,
- System transfer functions and eigenvalues in state space,
- Block diagram and state-space design,
- Control Law design for pole placement,
- Estimator design,
- Compensation—Combined control and estimation,
- Introduce the reference input, and
- Integral control.

An actual PID closed-loop controller for a real DC motor system is illustrated and discussed as a case study in Sect. 5.5. A real DC motor, **Mitsumi 448 PPR Motor**, is utilized as a target plant for this closed-loop control system to help users to build a real control system with TM4C123GH6PM MCU system. The detailed control programming codes are provided with detailed discussions.

Homework and Labs

5.1 For a given system $G(s) = \frac{1}{s(s+1)(s+5)}$, consider the root locus for $1 + KG(s) = 0$. Answer the following questions:

(a) Show the locations of poles and zeros for this root locus.
(b) Sketch the asymptotes for $K \to \infty$.
(c) For what value of K are the roots on the imaginary axis?

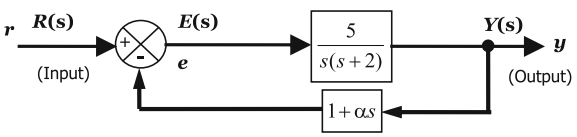


Fig. 5.74 A closed-loop control system

5.2 Sketch the root locus with respect to α for the system shown in Fig. 5.74. Sketch the step response for $\alpha = 0$, $\alpha = 0.5$, and $\alpha = 2$.

5.3 For a given system with the plant $G(s) = \frac{3}{s+2}$, design a proportional controller $C(s) = K$ for a unit negative feedback system and plot the root locus based on the gain K. Is the system stable for all values of $K > 0$? If $K = 8$ where will the closed-loop pole be?

5.4 Sketch the root locus for the system $G(s) = \frac{K(s+1)}{s(s+1)(s+2)}$ and determine the value of the root locus gain K such that the complex conjugate poles have a damping ratio of 0.5.

5.5 For a given system $G(s) = K\frac{s+1}{s+13}\frac{s^2+81}{s^2(s^2-100)}$;

(a) Find the locus of the closed-loop roots versus K.
(b) Is there any value of K that causes all roots to have a damping ratio greater than 0.5?
(c) Assuming the locus goes through a point where the damping ratio is 0.707. Try to find the values of K that result in these roots.

5.6 For a third-order system shown in Fig. 5.75, (a) Sketch the root locus of the closed-loop system. (b) Find the point at which the locus crosses the imaginary axis. Determine the value of K at that point. (c) Assume that the amplifier output u is a function of the input error e, as
(i) $u = e$ if $|e| \leq 1$. (ii) $u = 1$ if $e > 1$. (iii) $u = -1$ if $e < -1$.

5.7 Using the root locus method to determine the values of $\alpha < 0$ for which the polynomial $s^3 + 2s^2 + \alpha = 0$ has repeated roots.

5.8 Plot the root locus for the system $G(s) = K\frac{s+1}{s^2(s+p)}$ for $p = 5$ by hand. Determine the following parameters' values:

(a) Asymptotes parameters.
(b) Break points.
(c) The range of the value of p to make this system stable.

5.9 Using root locus method to design a phase lag compensator for a system that has a transfer function $G(s) = 1/(s+2)(s+3)$, determine the zero and pole locations of the compensator that will produce closed-loop poles at $s = -1 \pm j1$.

5.10 The basic transfer function of a satellite attitude control is $G(s) = \frac{1}{s^2}$. Design a phase lead compensator to meet the specifications, such as damping ratio $\xi = 0.5$ with natural frequency $\omega_n = 1$ rad/s. The ratio of pole and zero of the lead is less than 10. Plot root locus and step response.

5.11 Redo question 5.9 with MATLAB SISOTool.

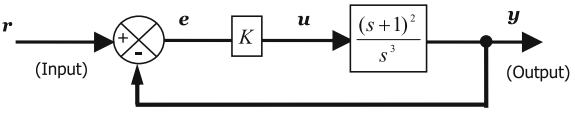


Fig. 5.75 The block diagram of a third-order closed-loop system

5.12 Consider a system given in Fig. 5.76, which may be unstable because it has pole and zero on the RHP. Can this system be made stable for some positive value of K?

5.13 Consider an instrument servomechanism with parameters given in Fig. 5.77. Use SISOTool to design a phase lead compensator to make closed-loop system damping ratio $\xi = 0.707$ and the dominant poles are $s_{1,2} = -10 \pm j10$.

5.14 A system is given as $G(s) = \frac{12}{s-2}$ that may not be stable. (a) Try to use root locus method to design a phase lag compensator $D(s) = K\frac{s+z}{s}$ to make the damping ratio $\xi = 0.7$ for the closed-loop control system and the closed-loop root locus should pass through $s_{1,2} = 5 \pm j5$ point. (b) Use SISOTool to design the same compensator to confirm your design with root locus method.

5.15 Consider a mechanical system shown in Fig. 5.78. The feedback path containing gs controls the amount of rate feedback. Determine the effect of varying g on the root locus when the gain $K = 1$.

5.16 The elevator in an office building has a transfer function $G(s) = K\frac{(s+10)}{s(s+1)(s+20)(s+50)}$. Use MATLAB function to determine the gain value of K when the complex roots have a damping ratio $\xi = 0.8$.

5.17 A control system has a negative unit feedback and process $G(s) = K\frac{s^2+4s+8}{s^2(s+4)}$. It is desired that the dominant roots have a damping ratio $\xi = 0.5$. Use MATLAB root locus function to show that $K = 7.35$ is required and the dominant roots are $s = -1.3 \pm j2.2$.

5.18 A system with a magnitude plot of the transfer function $G(s) = K\frac{(1+0.5s)(1+as)}{s(1+s/8)(1+bs)(1+s/36)}$ is shown in Fig. 5.79. Determine K, a, and b values from the plot.

5.19 A robot's arm has a transfer function with unit negative feedback control system as $G(s) = \frac{K}{s(1+s/10)(1+s/100)}$. Plot the Bode diagram for $K = 100$ and find the frequency where the 20 log $|G|$ is zero dB.

5.20 The Bode diagram of a system is shown in Fig. 5.80. Determine the transfer function $G(s)$ based on this plot.

5.21 A feedback system has a loop transfer function $G(s)H(s) = \frac{50}{s^2+11s+10}$. Sketch the Bode magnitude frequency response and determine the corner frequencies (break points).

5.22 A closed-loop control system is composed of a block diagram shown in Fig. 5.81a. The related transfer functions for these three blocks are represented by the frequency response curves shown in Fig. 5.81b. Calculate the damping

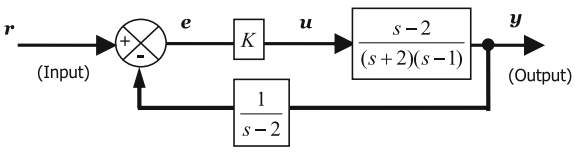


Fig. 5.76 A closed-loop system with unstable plant

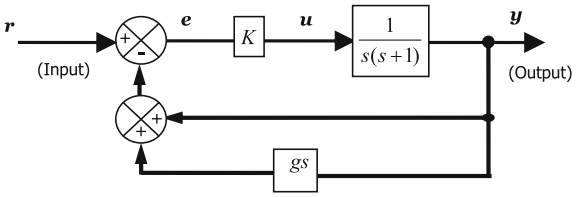


Fig. 5.78 A feedback control system

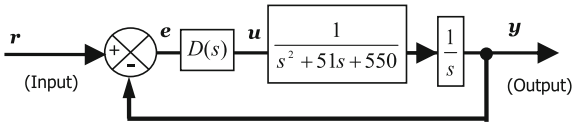


Fig. 5.77 A closed-loop system of an instrument servomechanism

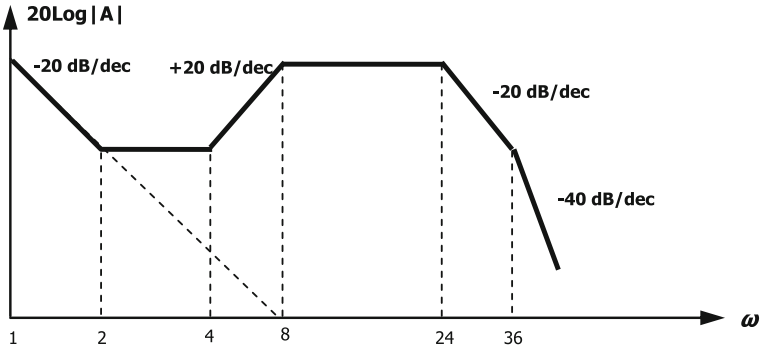
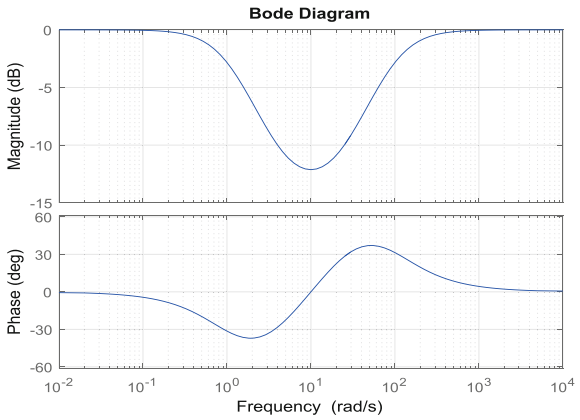


Fig. 5.79 The magnitude frequency response of a system

Fig. 5.80 Bode plot for a control system



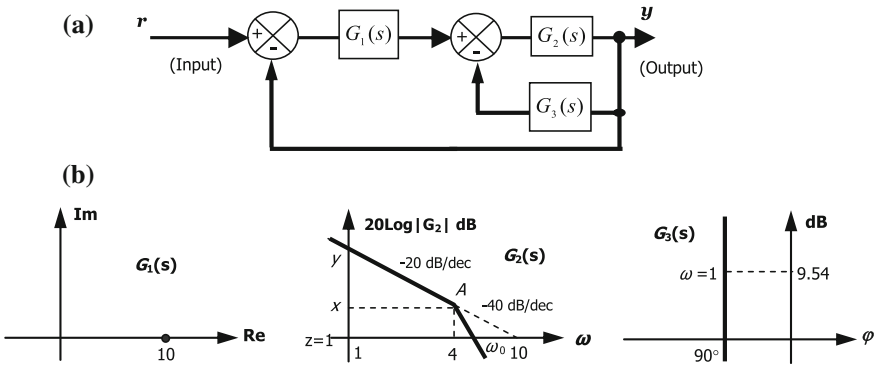


Fig. 5.81 Closed-loop system transfer function blocks and responses

ratio ξ when G_3 is disconnected and connected to this system. Assume that the system is a minimum phase system.

5.23 A robot's arm control system with a transfer function $G(s) = \frac{5}{s(1+s/1.4)(1+s/3)}$. Design a phase lead compensator $D(s)$ to obtain the following design specifications:
(a) Phase margin PM $\geq$ 40°, (b) Track error for a ramp input $r(t)$ is less than or equal to 10%, and (c) Crossover frequency $\omega_c \geq 1$ rad/s.

5.24 An airplane altitude control system is a closed-loop control system shown in Fig. 5.82.

(a) Sketch the magnitude and phase frequency responses of the open-loop system with MATLAB functions for $D(s) = K = 1$.
(b) What value of the gain K would provide a crossover frequency of 0.16 rad/s?
(c) For the value of the gain K in part (b), would the system be stable if the loop is closed?
(d) What is the phase margin PM when the gain is equal to 0 dB?
(e) Sketch the root locus with MATLAB functions and locate the roots for your value of K from part (b).
(f) What is the steady-state error with the value of K from part (b) if the input command is a ramp $R(s) = 1/s^2$?
(g) Design a phase lead compensator $D(s) = K\frac{s+z}{s+p}$ so that the crossover frequency is 0.16 rad/s and the PM is greater than 50°. Confirm your

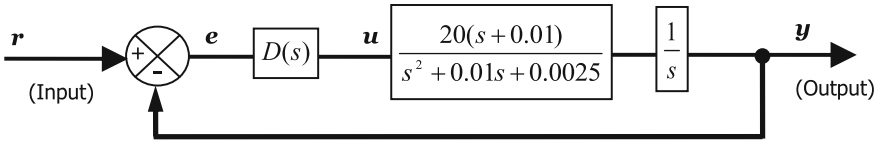


Fig. 5.82 A unit feedback control system

work with Bode plot. You can use SISOTool to help you to finish this compensator design.

(Hint: You need to use MATLAB function `bode(G,{0.01,10});` to plot complete Bode plot, and click any point on the magnitude trajectory and move along the trajectory to get the desired gain and frequency).

5.25 A process has a magnitude response shown in Fig. 5.83. Assume that the process is stable and minimum phase system. (a) What is the velocity constant K_v? (b) Determine the transfer function for this process. (c) What is the phase margin of the process?

5.26 A robotic leg system has a transfer function $G(s) = \frac{1}{s(s^2+3s+20)}$. Try to use MATLAB functions, such as `pidtuneOptions()` and `pidtune()` to design a PID controller to make the closed-loop control system to have a phase margin PM = 80°. Plot the step response with MATLAB functions.

5.27 Try to use MATLAB tool `pidTuner()` to tune the PID controller built in 5.26 to get the optimal step response.

5.28 Use MATLAB Simulink to redo 5.26 to find an optimal PID controller with the optimal step response. Use the PID gains obtained in 5.26 as the initial values for the PID block.

5.29 Given a standard second-order closed-loop system, try to (a) Show that the magnitude $T(j\omega) = \left|\frac{1}{(j\omega/\omega_n)^2+2\xi(j\omega/\omega_n)+1}\right| = \frac{1}{2\xi}$ at $\omega = \omega_n$. (b) Show that $|F(j\omega_c)| = \frac{1}{2\sin(PM/2)}$.

5.30 For a given system $G(s) = \frac{100(1+s/a)}{s(s+1)(1+s/b)}$, where $b = 10a$. Find a value of a to produce good stability by plotting the magnitude frequency response only.

5.31 A unit feedback control system for a robot submarine has a transfer function as $G(s) = \frac{K}{s(s+10)(s+50)}$. Designing a phase lead compensator make the closed-loop system to have 7.5% overshoot for a step input and a 0.4 s settling time. Let the zero of the compensator be $s = -15$. Determine the pole of the compensator and the velocity constant K_v.

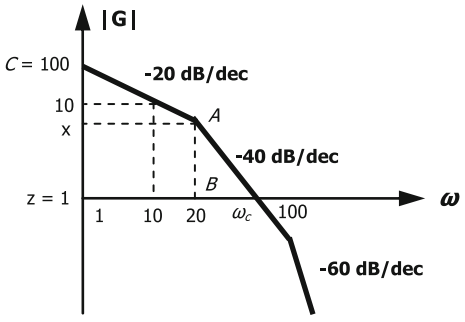


Fig. 5.83 The Bode plot for a process

5.32 A robot rotation system has a transfer function $G(s) = \frac{360}{s(s^2/6400 + s/50 + 1)}$. It is desired to design a compensator to have a velocity constant $K_v = 10$ with an overshoot to a step input less than 15%.

5.33 A robot extender has a transfer function $G(s) = \frac{8}{s(1+2s)(1+0.05s)}$. Design a compensator to make the velocity constant equal to 5 and the damping ratio is about 0.5.

5.34 Use Nyquist criterion to check whether any of the following systems are stable or not. For each case, provide the values of N, P, and Z.
(a) $GH(s) = \frac{1}{(1+0.5s)(1+2s)}$; (b) $GH(s) = \frac{1+0.5s}{s^2}$; (c) $GH(s) = \frac{s+4}{(s^2+5s+25)}$;
(d) $G(s)H(s) = \frac{30(s+8)}{s(s+2)(s+4)}$

5.35 Sketch the polar plots for the following loop transfer functions $G(s)H(s)$ to determine the point where the polar plot across the u-axis, and find the values of K to make systems stable.
(a) $G(s)H(s) = \frac{K}{s(s^2+s+4)}$; (b) $G(s)H(s) = \frac{K(s+2)}{s^2(s+4)}$.

5.36 A speed control system for a gasoline engine is shown in Fig. 5.84.
(a) Find the value of gain of K to make the steady-state error less than 10%. (b) With the gain K determined by (a), use Nyquist criterion to check the stability of the system. (c) Find the phase margin and gain margin of the system.

5.37 A control system has a transfer function $G(s) = \frac{5}{5s+1}$ and the compensator has its transfer function as $D(s) = K_1 + \frac{K_2}{s} = \frac{K_1 s + K_2}{s}$. The delay time for this control system is $T = 1.5$ s.
(a) Sketch the Bode plot for $K_1 = K_2 = 1$ and check the stability of the system. (b) Repeat (a) for $K_1 = 0.1$ and $K_2 = 0.04$. (c) When $K_1 = 0$, use the Nyquist criterion to find the maximum gain K_2 to make system stable.

5.38 A unit negative feedback control system for the yaw control is shown in Fig. 5.85. (a) Use MATLAB function `margin()` to find the phase margin PM, gain margin GM, and crossover frequency ω_c when $K = 0.5$. (b) Using the gain margin from part (a), determine the maximum value of gain K for a stable system.

5.39 A unit negative feedback control system has a transfer function $G(s) = \frac{10}{s(s+1)}e^{-Ts}$. Use Nyquist plot to confirm that the system is stable,

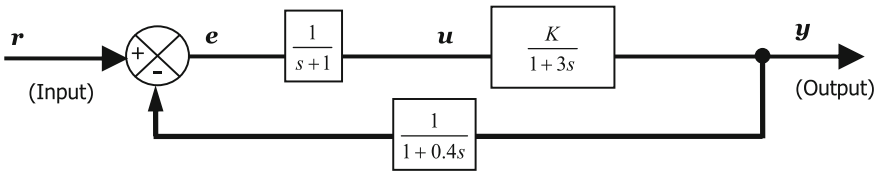


Fig. 5.84 A speed control system for a gasoline engine

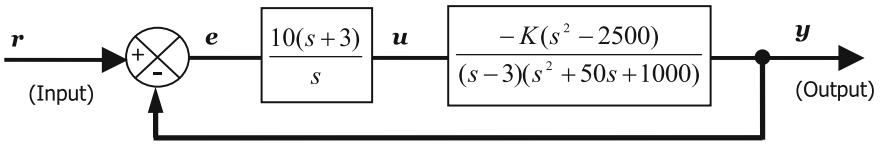


Fig. 5.85 A feedback control system for the yaw acceleration control

marginally stable and unstable with the propagation time delay T = 0.05, 0.1, and 0.2 s.

5.40 A given control system with the transfer function $G(s) = \frac{s}{s^2+4}$. (a) Convert this transfer function to the state-space variables and represent the system in the $\dot{x} = Ax + Bu$ and $y = Cx$ form. (b) Design a control law of the form $u = -[K_1 \ K_2]\begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$ that places the closed-loop poles at $s = -2 \pm j2$.

5.41 A control system has a given plant described by

$$\dot{x} = \begin{bmatrix} 0 & 1 \\ 7 & -4 \end{bmatrix} x + \begin{bmatrix} 1 \\ 2 \end{bmatrix} u, y = [\,1 \quad 3\,]x.$$

(a) Draw the block diagram for the plant at the integrator level. (b) Find the transfer function using matrix algebra. (c) Find the closed-loop characteristic equation if the feedback is $u = -[K_1 \quad K_2]x$.

5.42 The equation of motion of a simple pendulum is $\ddot{\theta} + \omega^2\theta = u$. (a) Write the equation of motion in state-space form. (b) Design an estimator that reconstructs both state variables of the pendulum given measurements of $\dot{\theta}$. Suppose ω = 5 rad/s and the roots of the estimator are $s = -10 \pm j10$. (c) Design a controller to determine the gain values of **K** so that the roots are at $s = -4 \pm j4$. (d) Write the transfer function of the estimator between the measured value of $\dot{\theta}$ and the estimated value of $\hat{\theta}$.

5.43 For each of the transfer functions, convert it to the state equation form. Also draw the related block diagram at the integrator level and give the related **A**, **B,** and **C** matrix.
(a) $G(s) = \frac{s^2-2}{s^2(s^2-1)}$; (b) $G(s) = \frac{3s+4}{s^2+2s+2}$.

5.44 A given motor control system dynamic model is $G(s) = \frac{1}{s(s+1)}$. (a) Let $y = x_1$ and $\dot{x}_1 = x_2$. Determine the state equation by writing it with the state matrix form. (b) Find the vector **K** so $u = -K_1x_1 - K_2x_2$ gives closed-loop poles at $\omega_n = 3$ and ξ = 0.5. (c) Find vector **L** of the state estimator so that the state error equations have characteristic equation with ω_1 = 15 and ξ_1 = 0.5. (d) Develop the transfer function of the controller obtained by combining (b) and (c).

5.45 A system has a state matrix $A = \begin{bmatrix} 0 & 1 \\ -1 & -1 \end{bmatrix}$. Find the characteristic roots of the system.

5.46 Find the state variable matrix with the block diagram for a system with a differential equation as

$$2\frac{d^3y}{dt^3} + 4\frac{d^2y}{dt^2} + 6\frac{dy}{dt} + 8y = 10u(t)$$

5.47 A control system is described by two differential equations as

$$\frac{dy}{dt} + y - 2u + aw = 0, \frac{dw}{dt} - by + 4u = 0,$$

where y and w are functions of time and u is the input. (a) Select a set of state variables. (b) Write the matrix differential equation and indicate the elements of the matrices. (c) Find the characteristic roots of the system in terms of the parameters a and b.

5.48 A depth-control system for a robot is shown in Fig. 5.86. The default control gain $K = 1$. (a) Find the state variable matrix representation for this system. (b) Check whether the system is stable by finding the characteristic roots of the system.

5.49 A given control system has the following state variable differential equations

$$\dot{x} = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ -2 & -2 & -4 \end{bmatrix} x + \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} u; \quad C = [1 \quad 0 \quad 0]x.$$

(a) Determine the related transfer function $G(s)$ using the inverse matrix. (b) Use MATLAB function `ss2tf()` to confirm and check the result from part (a). You can also try to use `ss()` and `tf()` functions to do this conversion. (c) Use MATLAB function `lsim()` to plot the response of the system to the initial condition $x(0) = [0 \quad 0 \quad 1]^T$ for $0 \leq t \leq 10$.

5.50 A unit negative feedback control system is shown in Fig. 5.87. (a) Find the state variable representation for the controller D(s). (b) Repeat part (a) for the plant. (c) With the controller and the plant in state variable form, use

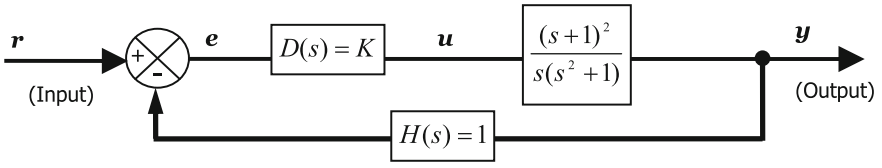


Fig. 5.86 A closed-loop control system for a depth-control robot

MATLAB functions, `series()` and `feedback()`, to get a closed-loop system representation in state variable form and plot the closed-loop system impulse response using MATLAB function `impulse()`.

5.51 A system is described by the matrix equations $\dot{x} = \begin{bmatrix} -4 & 1 \\ 0 & -1 \end{bmatrix} x + \begin{bmatrix} 0 \\ 1 \end{bmatrix} u; \quad y = [\,1 \quad 0\,] x$. Determine whether is controllable and observable.

5.52 A DC motor has a transfer function $G(s) = \frac{4}{s(s^2+5s+4)}$. Determine whether the system is controllable and observable.

5.53 The transfer function of a plant to be controlled is $G(s) = \frac{4}{(s^2+5s+4)}$. Use state variable feedback control to design a controller so that the steady-state error for a step input is zero. Also select the gains to make system have small overshoot, OV = 1% and a settling time less than 1 s.

5.54 A feedback control system has a plant transfer function $G(s) = \frac{K}{s(s+70)}$. Design a state variable feedback system to make the velocity constant $K_v = 35$, and overshoot to a step response is about 4% so that the damping ratio $\xi = 0.707$. The settling time $T_s = 0.11$ s.

5.55 A system has a matrix differential equation $\dot{x} = \begin{bmatrix} 1 & 0 \\ 0 & 2 \end{bmatrix} x + \begin{bmatrix} b_1 \\ b_2 \end{bmatrix} u$. Determine the values of b_1 and b_2 to make system controllable.

5.56 A system has the state matrix differential equation as

$$\dot{x} = \begin{bmatrix} -1 & 1 & 0 \\ 4 & 0 & -3 \\ -6 & 8 & 10 \end{bmatrix} x + \begin{bmatrix} 1 \\ 0 \\ -1 \end{bmatrix} u; \quad y = [\,1 \quad 2 \quad 1\,] x.$$

Use MATLAB functions `ctrb()` and `obsv()` to check whether the system is controllable and observable.

5.57 Consider a system with a transfer function $G(s) = \frac{1}{s^2}$. Design a state variable feedback gain to make the desired roots of the characteristic equation be $s = -3 \pm j1$.

5.58 A system has a plant $G(s) = \frac{3s^2+4s-2}{s^3+3s^2+7s+5}$. Add state variable feedback so that the closed-loop poles are $s = -4$, -4, and -5.

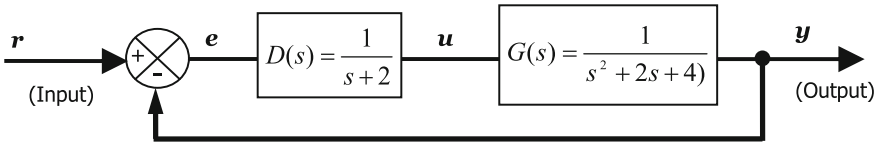


Fig. 5.87 A unit negative feedback closed-loop control system

5.59 A control system is expressed by a differential equation $\frac{d^2y}{dt^2} + 2\frac{dy}{dt} + y = u$. (a) Convert it to a state variable matrix form and confirm whether it is controllable. (b) Determine whether the system is controllable and observable.

5.60 A control system described by a differential equation $\ddot{x} = 1000x + 20u$. Use pole placement method to design the system to meet the specifications as settling time $T_s \leq 0.25$ s and an overshoot OV $\leq$ 20%.

5.61 A manipulator control system has a transfer function $G(s) = \frac{1}{s(s+0.4)}$. (a) Find the state variable matrices and related differential equation and plot the closed-loop step response. (b) Use state variable feedback so that the overshoot is about 5% and the settling time is 1.35 s. (c) Plot the response of the state variable feedback system to a step input.

Chapter 5 Lab Projects—PID Controller Designs

Lab Project 5.1

A given process that is a DC motor has a transfer function $G(s) = \frac{6520}{s(s+430.6)} e^{-0.005s}$. The propagation time delay can be ignored since it is only 5 ms. Using MATLAB SISOTool to design and tune a PD controller for this plant (a) Using the Bode plot method, and (b) Using the root locus method with the following specifications:

(1) Overshoot $\leq$ 15%.
(2) Settling time $\leq$ 1.0 s.

Lab Project 5.2

Use MATLAB Simulink to build a control model to perform simulations to get an optimal PID controller for the plant given in Lab 5.1.

Lab Project 5.3

Use MATLAB pidTuner to tune and build an optimal PID controller for the plant shown in Lab 5.1.

Lab Project 5.4

5.4.1 **Goal**

This lab project allows students to build a bound–bound–control project with PWM module to drive a 12 V DC motor in a changing-duty-cycle mode. Two switch buttons, **SW2** and **SW3** installed in the EduBASE ARM® Trainer, are used as two trigger sources to change the duty cycle of the PWM output via M1PWM5 pin or PWM Module 1 Generator 2.

5.4.2 **Data assignment and hardware configuration**

In this lab project, we need to use a DC motor fan (model number: **EE80251S2-000U-999**) as our target motor to drive. The hardware connection is shown in Fig. 5.88.

The hardware and software configurations for this example project include the following:

- We use the first control and driving block, block A, in the motor driver TB6612FNG to control and drive our target 12 V DC motor and fan.

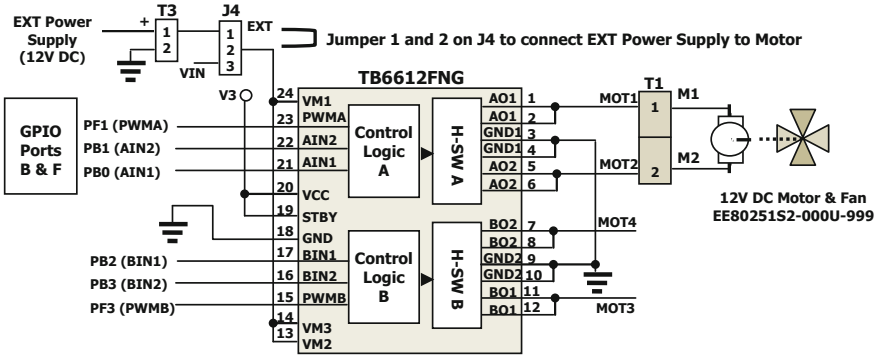


Fig. 5.88 Hardware configurations for the Lab 5.4 project

- The DC motor with a fan is a 12 V DC motor with single rotating direction.
- The GPIO ports and pins to be used include the following:
  - **PF1** is used to provide the PWM driving signal to the motor (**PWMA**).
  - **PB0** (**AIN1**) and **PB1** (**AIN2**) are used to provide the rotating direction signal (**PB1:PB0 = 01** → CW and **PB1: PB0 = 10** → CCW) for the motor. In this project, we use the first rotating direction PB1:PB0 = 01 since the motor can be rotated in a single direction.
  - **PD3** is connected to the switch button **SW2** and **PD2** is connected to the switch button **SW3** in the EduBASE ARM® Trainer. Both **PD2** and **PD3** pins are Low if no any of these two buttons are pressed. However, a High is applied on either pin if either button is pressed.
- Connect the DC motor to two blue interfacing blocks **M1** and **M2** in the EduBASE ARM® Trainer with positive (red color) wire to **M1** and reference (black color) wire to **M2**.
- Connect an external 12 V DC power supply to the blue block **T3** with positive wire to the **+** terminal on the **T3**.
- Change the jumper on the **J4** to connect 1 and 2 pins together on the **J4** to connect the external power supply to the TB6612FNG motor driver.

In this lab project, we use Module 1 PWM Generator 2 (**M1PWM5**) to generate a duty-modified PWM signal to control this DC motor via **pwm2B'** (PWM5) pin since the **PF1** pin is connected to this **pwm2B'** output pin for this generator in the EduBASE Trainer. The system clock we used is 40 MHz; after dividing by 2, a 20 MHzclock is applied for the PWM module 1. The input frequency to the counter is 5 kHz.
Now let's start to create this project **DRAPWM**.

5.4.3 Development of the project

Use steps below to develop this project. Both a header file and a C source file are used in this project since it is a little complicated. Create a new project with the following steps:

(1) Create a new folder **Lab5_4** under the folder **C:\C-M Control Lab Projects\Chapter 5** in the Windows Explorer.
(2) Open the Keil® ARM-MDK μVersion5, create a new project named **Lab5_4**, and save this project in the folder **Lab5_4** created in step 1.
(3) On the next wizard, you need to select the device (MCU) for this project. Expand three icons, **Texas Instruments**, **Tiva C Series**, and **TM4C123x Series**, and select the target device **TM4C123GH6PM** from the list by clicking on it. Click on the **OK** to close this wizard.
(4) Next, the Software Components wizard is opened, and you need to set up the software development environment for your project with this wizard. Expand two icons, **CMSIS** and **Device**, and check the **CORE** and **Startup** checkboxes in the **Sel.** column, and click on the **OK** button since we need these two components to build our project.

5.4.4 Development of the header file

(1) In the **Project** pane, expand the **Target** folder, and right-click on the **Source Group 1** folder and select the **Add New Item to Group 'Source Group 1'**.
(2) Select the **Header File (.h)** and enter **Lab5_4** into the **Name:** box, and click on the **Add** button to add this header file into the project.
(3) Include the following system header files and macros:

- **#include <stdint.h>**
- **#include <stdbool.h>**
- **#include "TM4C123GH6PM.h"**
- **#define GPIO_ports_PORTF_CR_R (*((volatile uint32_t *)0x40025524))**
- **#define SYSCTL_SYSDIV_5 0x02400000**
- **#define SYSCTL_USE_PLL 0x00000000**
- **#define SYSCTL_OSC_MAIN 0x00000000**
- **define SYSCTL_XTAL_16MHz 0x00000540**

The reason to include the system header file "**TM4C123GH6PM.h**" is that we want to use structure pointers for all related registers on GPIO Ports and

PWM modules. Those definitions for five macros are used to access related system constants and register. Save this header file as **Lab5_4.h**.

5.4.5 Development of the C source file

(1) In the **Project** pane, expand the **Target** folder and right-click on the **Source Group 1** folder and select the **Add New Item to Group 'Source Group 1'**.
(2) Select the **C File (.c)** and enter **Lab5_4** into the **Name:** box, and click on the **Add** button to add this source file into the project.
(3) Include the header file "**Lab5_4.h**" into this source file.
(4) Declare the user-defined function **void Delay(uint32_t time);**.
(5) Inside the **main(void)** program, declare two unsigned 32-bit integer local variables, **pw = 20** and **RCC**. The first variable is used to set up and maintain the duty cycle value. The second one works as a temporary value holder for different values.
(6) Use the following macros to set up the system clock. **ORing** them together and assign it to the variable **RCC**.

- **SYSCTL_SYSDIV_5**
- **SYSCTL_USE_PLL**
- **SYSCTL_OSC_MAIN**
- **SYSCTL_XTAL_16MHz**

(7) Assign the resulted RCC to the Run-Mode Clock Configuration (**RCC**) register via the structure pointer **SYSCTL→RCC**.
(8) Enable the PWM1 module with clock by **ORing** and assigning **(|=) 0x2** to the Pulse Width Modulator Run-Mode Clock Gating Control (**RCGCPWM**) register using the structure pointer format (**SYSCTL→RCGCPWM**).
(9) Use a **while()** loop to wait for this enabling and clocking configuration to be finished. The loop condition for this **while()** loop is **(SYSCTL->PRPWM & 0x2) == 0**. The Pulse-Width Modulator Peripheral Ready (**PRPWM**) register is used to monitor and indicate whether the required peripheral (PWM1) is ready or not. The bit-1 (**R1**) on this register is set to 1 when the PWM1 is ready.
(10) Enable GPIO Ports F, D, and B with clocks by assigning appropriate values to the General-Purpose Input/Output Run-Mode Clock Gating Control (**RCGCGPIO**) register.
(11) Similarly to step 9, use another **while()** loop to wait for the GPIO Ports to be configured and enabled. The loop condition can be **(SYSCTL->PRGPIO & 0x2A) == 0**. The General-Purpose Input/Output Peripheral Ready (**PRGPIO**) register is used to monitor and indicate whether the selected GPIO Port is ready or not. Starting from bit-0 (**R0**) to bit-5 (**R5**), each bit is associated with a GPIO Port, **R0**—Port A, **R1**—Port B, … **R5**—Port F. if the selected port is ready, the corresponding bit is set to 1.

(12) Configure the **RCC** register again to get a 20 MHzclock source for the PWM module 1 by configuring the bits **USEPWMDIV** and **PWMDIV** on this register.
(13) Enable PWM1 module again by assigning appropriate value to the **RCGC0** register.
(14) Unlock the GPTO Port F Commit Register **GPIOCR** by assigning the unlock-key **0x4C4F434B** to the GPIO Port F Lock register (**GPIOF→LOCK**).
(15) Enable **PF1** to be committed by **ORing** and assigning (**|=**) 0x2 to the GPIO Port F Commit Register macro (**GPIO_PORTF_CR_R**). This step enables the following configurations for the GPIO Port F **AFSEL** and **PCTL** registers to be committed immediately when they are set up or configured.
(16) Then lock the GPIOF commit register by assigning **0x0** to the GPIO Port F Lock register macro (**GPIOF→LOCK**) to keep this configuration (refer to Sect. 3.2.1.2.3 in Chap. 3).

The following codes are used to configure and setup PWM Module 1 Generator 2 or M1PWM5 pin to enable it to output a PWM signal.

(17) Disable the **PWM1_2B** or M1PWM5 pin by assigning **0x0** to the PWM Module 1 Generator 2 Control Register macro **PWM1→_2_CTL**.
(18) Configure the PWM Module 1 Generator 2 register **PWM1GENB** by assigning a value of **0x0000080C** to the **PWM1GENB** register macro (**PWM1→_2_GENB**). This configuration enables the M1PWM5 pin to output High when the counter equals to the **LOAD** value and output Low when the counter matches the **CMPB** value when it is counting down.
(19) Load the period value **3999** (4000 – 1) to the PWM1 **LOAD** register (**PWM1→_2_LOAD**).
(20) Enable the **PWM1_2B** or M1PWM5 pin by assigning **0x1** to the PWM Module 1 Generator 2 Control Register macro **PWM1→_2_CTL**.
(21) Enable the PWM1 module by assigning **0x20** to the PWM Master Control register macro **PWM1→ENABLE**.

The following codes are used to initialize and configure GPIO Ports B, D, and F.

(22) Configure the **PF1** pin to work as an output and digital function pin by assigning appropriate values to the Port F Direction and Digital Enable registers (**GPIOF→DIR** and **GPIOF→DEN**).
(23) Configure the **PF1** pin to work as an alternate function, M1PWM5 output pin, by assigning an appropriate value to the GPIO Port F AFSEL register **GPIOF→AFSEL** (refer to Fig. 3.7 and Table 3.2 in Chap. 3 to get more details about this register).

(24) Configure **PF1** pin to work as a PWM output pin by assigning **0x00000050** to the Port F Port Control register macro **GPIOF→PCTL**.
(25) Disable analog function on **PF1** pin by **ANDing** and assigning (**&=**) an inverse value of **0x02** to the Port F Analog Mode Select register macro **GPIOF→AMSEL**.
(26) Configure **PD3 ~ PD0** pins to work as input and digital function pins by assigning appropriate values to the Port D Direction and Digital Enable registers (**GPIOD→DIR** and **GPIOD→DEN**).
(27) Configure **PB1 ~ PB0** pins to work as output and digital function pins by assigning appropriate values to the Port B Direction and Digital Enable registers (**GPIOB→DIR** and **GPIOB→DEN**).
(28) Set **PB0** to 1 to make **AIN1** as HIGH to enable DC motor to rotate in the CW direction. This setup can be accomplished by assigning **0x1** to the Port B **DATA** register.
(29) Clear the Port D by assigning **0x0** to the Port D **DATA** register.

The following codes are used to monitor two switch buttons **SW2** and **SW3** to perform related duty cycle increment or decrement operations based on these two buttons.

(30) An infinitive **while()** loop is used to start this process.
(31) First read the GPIO Port D **DATA** register and assign the reading value to the **RCC** variable.
(32) Use the **if()** selection structure to check whether the **SW2** is pressed (PD3 = 1) or the **SW3** is pressed (PD2 = 1). The first checking condition is (**RCC & 0x8**) and the second checking condition is (**RCC & 0x4**).
(33) If the first checking condition is true, the **SW2** is pressed. The variable **pw** is incremented by 1. If this value is greater than 3999, adjust it to 3990. Then assign this **pw** value to the PWM Module 1 Generator 2 Compare B register macro **PWM1→_2_CMPB**.
(34) If the second checking condition is true, the **SW3** is pressed. The variable **pw** is decremented by 1. If this value is less than 100, adjust it to 100. Then assign this **pw** value to the PWM Module 1 Generator 2 Compare B register macro **PWM1→_2_CMPB**.
(35) Call **Delay(1000)** function to delay the program a period of time.
(36) For the **Delay()** function, just put a blank **for()** loop to repeat looping a certain time.

5.4.6 Set up the environment to build and run the project

The only environment to be set is to make sure that the debugger shown in the **Use:** box in the **Debug** tab under the **Project|Options for Target 'Target 1'** menu item is **Stellaris ICDI**. Otherwise, you can click on the drop-down arrow to select this debugger from the list.

5.4.7 Demonstrate your program

Perform the following operations to run your program and check the running results:

- Go to **Flash|Download** menu item to download your program into the flash ROM.
- Go to **Debug|Start/Stop Debug Session** to begin debugging your program. Click on the **OK** button on the 32 KB memory size limitation message box to continue.
- Then go to **Debug|Run** menu item to run your program.

As the project runs, when you pressed **SW2** button, the duty cycle is incremented and the rotation speed of the DC motor becomes faster. As you pressed the **SW3** button, the duty cycle is decreased and the rotating speed of the DC motor is reduced.

Based on these results, try to answer the following questions:

- When either **SW2** or **SW3** button is pressed as the project runs, the rotation speed of the DC motor is changed, either goes up or goes down, why this happened?
- What did you learn from this project?

Chapter 6
Practical Nonlinear Control Systems



All practical systems, including common control systems, contain nonlinearities. Yet, linear control theory, with its powerful design methods discussed earlier, more often than not works well. This is because the nonlinearities may be either unengaged or negligible. A DC motor, for instance, has both velocity and acceleration limits. However, careful planning of the motor's commanded motions may help avoid ever hitting these saturation levels, allowing the linear model assumption to hold. As another example, quantization errors in the digital control algorithm (modeled by staircase-shaped nonlinearity) may become negligible for large word length selection.

This chapter focuses on nonlinearities that are either unavoidable or inherent in the controlled process model, such as motion kinematics that causes trigonometric functions to appear in the process model as it results directly from the motion geometry. Other nonlinearities discussed in this chapter are those that are added intentionally to the control loop, such as relays, in order to achieve control performance, for certain specific command inputs, that may be superior to that obtained via linear control means.

Even a single nonlinearity that exists in the control loop, that is engaged actively or is inherent in the model, immediately renders any application of linear control theory totally useless—there are no more transfer functions, and the superposition principle is no longer valid. Closed-loop stability is harder to assess whenever nonlinearities exist. Hence, root locus and Bode plots design methods are not possible.

Most biological and physiological control processes are inherently nonlinear. Examples: A concentration of a substance cannot be negative. The rate of chemical reaction often depends on the product of reactants concentrations ("the Law of Mass Action"). In an enzyme-assisted reaction, the rate of product formation depends hyperbolically on the substrates concentrations ("Michaelis–Menten Enzyme Kinetics").

In the control of electrical power grids, there are many nonlinearities to consider, one of which is that the transfer of real power from one station to another depends

© Springer Nature Switzerland AG 2019

Y. Bai and Z.S. Roth, *Classical and Modern Controls with Microcontrollers*,
Advances in Industrial Control, https://doi.org/10.1007/978-3-030-01382-0_6

on the sine function of the phase difference between the two stations. Such nonlinearities may cause instability in the form of a station that may become unsynchronized with the grid.

Phase-locked loops (PLL) that play an important role in communication systems are inherently nonlinear as well, as the phase detector (one of the key components in such a loop) has a periodic nonlinearity because a phase of 0°, for instance, is indistinguishable from a phase difference of 360°. PLLs, like electrical power stations, may go out of sync for large and sudden changes in the incoming frequency signal.

There are many more control systems with major nonlinearities and all such systems are highly practical.

Very often the only tool that may be available to analyze a nonlinear system is a simulator. This chapter focuses on dynamic simulations using MATLAB Simulink. Dynamic simulation is often very useful in linear models as well but its real effectiveness is felt when models are nonlinear. This chapter which is full of Simulink examples can serve as a useful tutorial to such software tool, along with other handbooks such as [1].

6.1 MATLAB Simulink Basics for Analysis of Nonlinear Systems

Simulink®, developed by MathWorks®, is a graphical programming environment for modeling and simulation of dynamic systems. MATLAB Simulink simulation model in its most basic form consists of a single Simulink model that is self-contained: its input signals are generated internally using preexisting components library blocks, as well as the run commands, and the graphical outputs come from components such as Scopes that can be edited and manipulated entirely within Simulink. In the basic form, there is no need to use MATLAB script for controlling the Simulink model or for postprocessing of the simulation results.

The basic ideas of how to set and run a Simulink model are explained in this section by means of examples of practical systems.

6.1.1 Simulink Model of a Pendulum

We shall simulate an idealized friction-free model of the pendulum [2] shown in Fig. 6.1.

The rod that has a length L is considered "mass-less". The entire mass m of the pendulum is assumed to be concentrated in the bob. The pendulum rotates on a plane that is parallel to the page. The bob's rest position is $\theta = 0$. If we assign the bob an initial potential energy by setting it at an angle θ_0, it begins to swing back

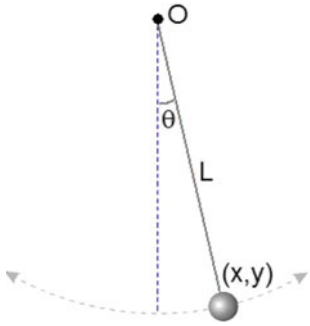


Fig. 6.1 Pendulum rotating in a vertical plane

and forth. With no friction, the pendulum oscillates endlessly about $\theta = 0$. The amplitude of oscillations depends on the initial amount of potential energy. Another possible mode of operation happens whenever certain initial angle θ_0 combined with some initial angular velocity $\dot{\theta}_0$ may cause the pendulum to tip over and rotate endlessly either in the clockwise or counterclockwise direction.

The only force that acts on the mass is the gravitational force mg (pointing downwards). The force of mg has two components. The tangential component equals $-\text{mg}\cdot\sin(\theta)$. The reason for the $(-)$ sign is that for any θ direction, the tangential force attempts to bring the mass back to its rest position. The gravity acceleration is $g = 9.807\ \text{m/s}^2 = 32.17\ \text{ft/s}^2$. We now apply Newton's Second Law to the moving bob which moves along a circular path s. The derivative $\text{d}s/\text{d}t$ denotes the rate of change of the distance (that is, the linear velocity of the bob) along the path. The second derivative $\text{d}^2s/\text{d}t^2$ is therefore the rate of change of the velocity (which is the linear acceleration of the bob) and by Newton's second law, the pendulum's motion equation is

$$m\frac{\text{d}^2 s}{\text{d}t^2} = -mg\sin(\theta) \tag{6.1}$$

The circular distance s and the rotation angle θ are related through the radius of rotation L:

$$s = L\theta \Rightarrow \frac{\text{d}^2 s}{\text{d}t^2} = L\frac{\text{d}^2\theta}{\text{d}t^2} \tag{6.2}$$

The resulting mathematical model of the pendulum is

$$\frac{\text{d}^2\theta}{\text{d}t^2} = -\frac{g}{L}\sin(\theta) \tag{6.3}$$

In order to perform dynamic simulation of system (6.3), we have to convert the second-order differential equation to an equivalent model represented by a family of first-order differential equations (known as a state variable model) as follows:

$$x_1 = \theta \quad x_2 = \frac{\mathrm{d}\theta}{\mathrm{d}t} \tag{6.4}$$

$$\frac{\mathrm{d}x_1}{\mathrm{d}t} = x_2 \quad \frac{\mathrm{d}x_2}{\mathrm{d}t} = -\frac{g}{L}\sin(x_1) \tag{6.5}$$

The goal of the simulation is to gain better insight about the performance of the pendulum. Questions that may be answered via simulation include: (a) For what sets of initial conditions $\{x_1(0), x_2(0)\}$ does the pendulum swing back and forth, and for what initial conditions does the pendulum swing in one direction? (b) Whenever the pendulum rotates back and forth, how do the rotation frequency and amplitude depend on L and on the initial conditions?

The Simulink model of (6.5) (shown below) uses two integrators (taken from the *Continuous* library), the outputs of which are x_1 and x_2, respectively. The input of the leftmost integrator (whose output is x_2), which is $\mathrm{d}x_2/\mathrm{d}t$, is constructed graphically based on Eq. (6.5). Note that annotations may be placed anywhere in the workspace. The trigonometric function, product, divide, and gain parts are all taken from the *Math* library. The constant blocks are taken from the *Source* directory. Double-clicking on an integrator component brings up a dialog box that allows a choice between internal and external initial conditions inputting.

For the initial position and velocity shown in Fig. 6.2, the scope output is shown in Fig. 6.3. The blue curve is the angle of the pendulum as function of time.

In Fig. 6.2, both integrators have initial conditions assigned externally from constant blocks. Externally assigned initial values are almost always preferred, not only because we want the parameters to be visible but also even more so due to a powerful property of constant blocks. Such blocks may accept algebraic values recognized as global parameters in a MATLAB script that runs the Simulink model. Numerical values to constant block parameters can be assigned within the controlling script. The Scope block can be edited to have multiple ports and to have light colored background and other style features, via the View → Style option.

In the model shown in Fig. 6.2, the angular acceleration expression of model (6.5) is fully created graphically allowing access to every interim calculation. In many practical applications, this is too excessive, and no generality is lost by creating the rate of change expression algebraically, as shown in Fig. 6.4 for an equivalent model. The model utilizes a multiplexer block (MUX, from the *Signal Routing* library) to combine signals and constants into a vector of signals. The vector elements are denoted $u(1)$, $u(2)$, … from the upper port of the MUX downwards. An algebraic expression in terms of the MUX output signals can be written in order to create algebraically the full rate of change expression.

For the initial conditions shown in Fig. 6.4, the scope output is shown in Fig. 6.5, demonstrating the other mode of operation of the pendulum. The red curve is the pendulum's angle as function of time whereas the blue curve is the pendulum angular velocity.

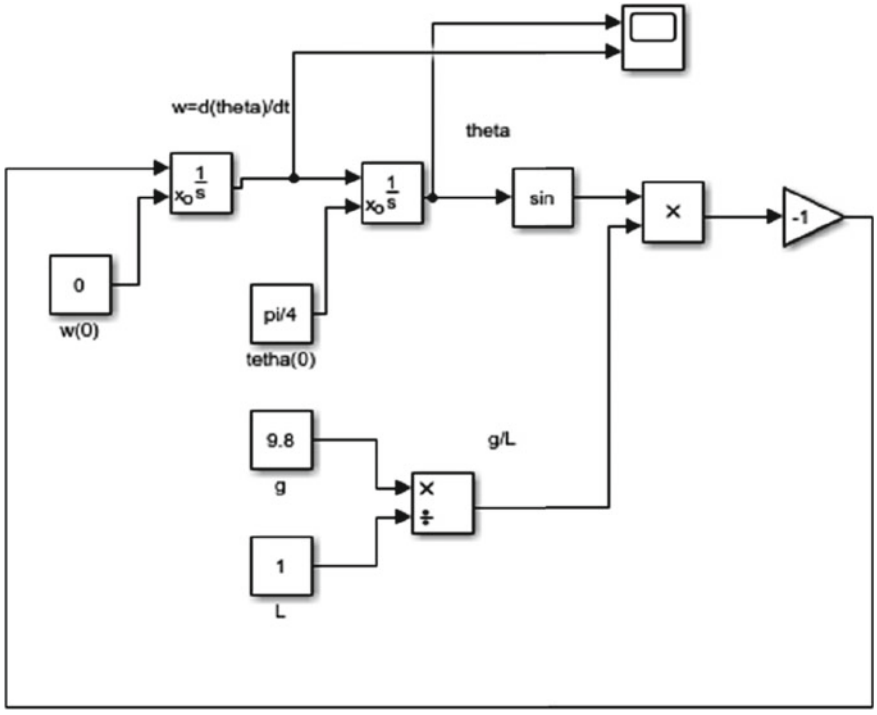
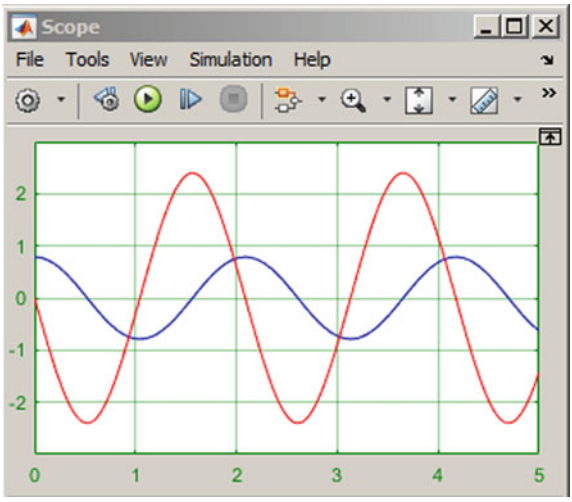


Fig. 6.2 Simulink block diagram of the pendulum equation

Fig. 6.3 Pendulum oscillating back and forth



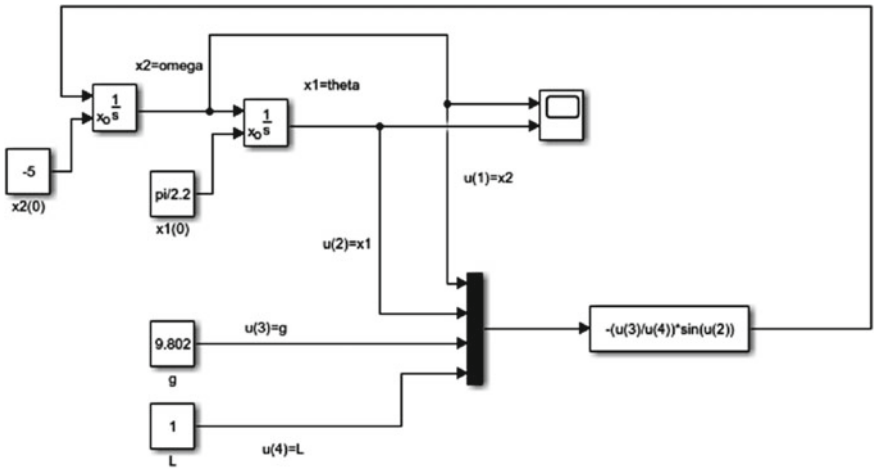


Fig. 6.4 Pendulum equivalent Simulink model using user-defined algebraic expressions

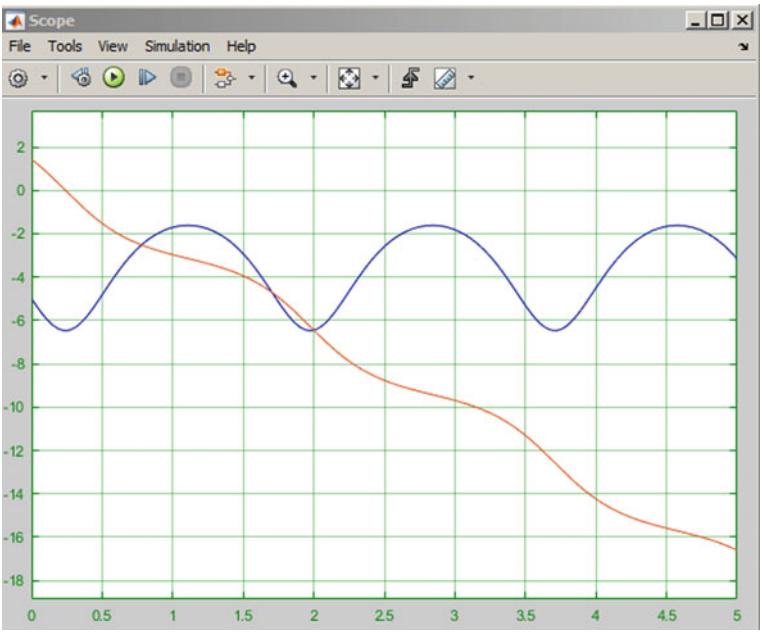


Fig. 6.5 Pendulum rotating in the counterclockwise direction

6.1.2 Simulink Model of a Master and Dog Line-of-Sight Tracking

The following example is based on the lecture notes of Prof. Schoen (Technion, Israel Institute of technology) [3]. In this "dog chasing a master" control scenario (which can easily be applied to a military application involving say a missile chasing a slower vehicle based on line-of-sight visual information), the master is assumed to be moving at a constant speed V_M parallel to some x-axis. The dog, running faster than its master at a speed of V_D, always runs towards the master along a radial direction. At any time $t > 0$, the radial direction creates an angle φ with respect to the negative direction of the x-axis. The initial distance between the master and the dog is $R(0)$ and the initial angle that the radial direction has with respect to the x-axis is $\varphi(0)$, as seen from Fig. 6.6.

The rate of change of R equals

$$\frac{dR}{dt} = -V_D + V_M \cos\varphi \tag{6.6}$$

In order to understand (6.6), assume first that $V_M = 0$ implying that in such a case, the rate of reduction in R equals to the dog's speed. Now, assume $V_D = 0$ implying that the increase in R equals $V_M \cos\varphi$, which is the radial component of V_M. The tangential component of V_M, which is equal to $V_M \sin\varphi$, causes a reduction of φ whenever $V_M > 0$. Therefore, the second system equation is

$$R\frac{d\varphi}{dt} = -V_M \sin\varphi \tag{6.7}$$

The two state variables of the system are therefore R and φ.

The Simulink model is shown in Fig. 6.7.

The Simulink model construction approach follows that of the model of Fig. 6.4 utilizing integrators, constant blocks, multiplexer, and user-defined function blocks. In the model, an integrator is assigned to each of the state variables. The initial conditions are taken as $R(0) = 1000$ m and $\varphi(0) = \pi/4$ rad. The upper two user-defined function blocks create dR/dt and $d\varphi/dt$, respectively. The lower two function blocks create the x and y coordinates of the dog, assuming that the dog's initial position is (0,0). A scope is assigned to each of the integrator outputs.

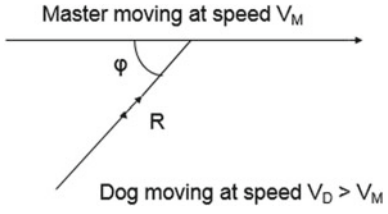


Fig. 6.6 Top view of a field depicting a dog chasing its master

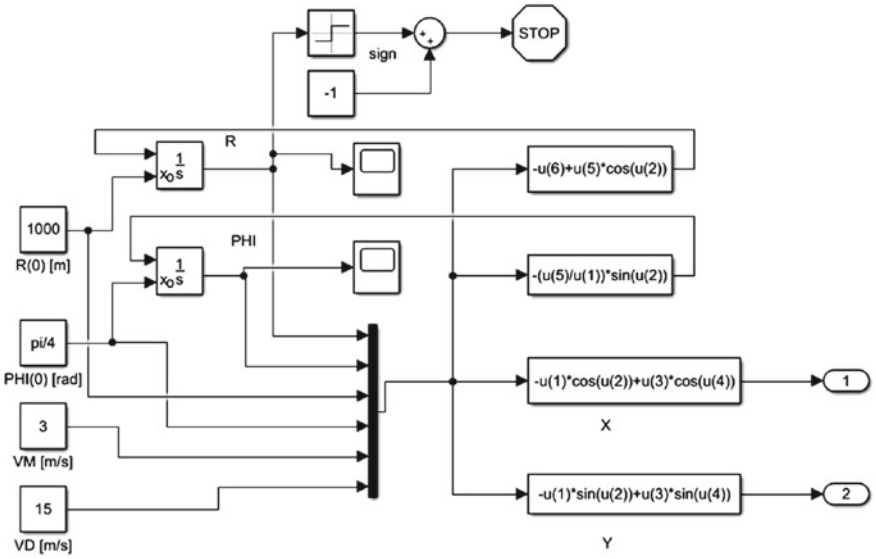


Fig. 6.7 Simulink model of the master and dog system

Of course due to the different scales, it would have been a mistake to send both state variables to the same scope.

The model has a little mechanism to cause a stopping of the simulation (using a STOP block taken from the *Sinks* library) when the model ceases to be valid, that is whenever $R(t)$ becomes negative. A STOP block terminates the simulation when its input signal becomes nonzero. As long as $R(t) > 0$, the output of the "sign" block (taken from the *Math Operations* library) equals 1 assuring, by summation of $a - 1$ constant, using a "summing" block (taken from the *Math Operations* library), that the input of the STOP block is zero. The "sign" block output becomes 0 whenever its input is 0. That "sign" block output becomes −1 when the input to the block is negative. As can be seen, the input of the STOP block becomes nonzero (instantly stopping the simulation) whenever R becomes nonpositive.

One more interesting feature of the Simulink model is the use of Out ports (taken from the *Sinks* library). The use of at least one Out port enables the availability of the "states" (that is, integrators outputs) as functions of time for plotting and analysis within a MATLAB script that can run the Simulink model. Other signals that become available are the Out port signals (the "outputs" signals). In this example, the two outputs can be plotted in MATLAB on an *XY* graph providing the dog's trajectory.

The scopes can be set to display the following $R(t)$ and $\varphi(t)$ outputs (Fig. 6.8).

A simple MATLAB script can access the Simulink model via the "sim" command, as shown in Fig. 6.9. A tentative name for the Simulink model of Fig. 6.7 is "book_master_dog_020918", used as the first argument of the "sim" command. The second argument is the simulation final time. More Simulink configuration

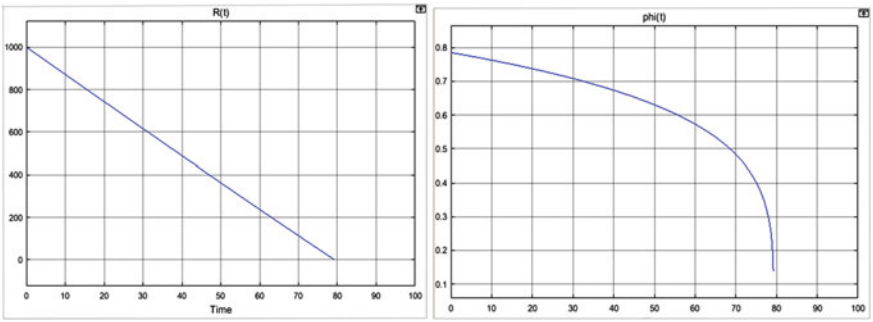


Fig. 6.8 Scope plots of $R(t)$ and $\varphi(t)$. Note that the simulation was stopped around $t = 79$ s

```
book_master_dog_plot_020918.m
1 -   [t,x,y]=sim('book_master_dog_020918',100);
2 -   plot(y(:,1),y(:,2))
3 -   axis([0,1000,0,1000]);
4 -   title('trajectory of dog chasing its master')
5 -   grid on
6 -   xlabel('x')
7 -   ylabel('y')
```

Fig. 6.9 MATLAB script for running a Simulink model and plotting an *XY* graph

parameters (such as the relative tolerance parameter "RelTol") can be added to the "sim" command using the "simset" command, not used in the file shown in Fig. 6.9. Two non-default Simulink configuration parameters, set within the Simulink model, were the solver relative tolerance = 1e-6 (changed from its default value of 1e-3) and the Data Interpolation Refine Output = 5 (changed from its default 1 value). The output of the "sim" command consists of the time vector t, the sub-matrix x of all integrators outputs (each as a vector of the same dimension as t), and sub-matrix y of all Out port output signals. The script of Fig. 6.9 has one purpose and it is to plot the *XY* graph of the cartesian coordinates of the dog's motion.

The resulting XY graph is shown in Fig. 6.10.

The simulation may be run for other master and dog speeds. A more realistic master's brisk walking speed is around 3.5 mph which is around 1.5 m/s. Some of the fastest dogs may reach a speed of 40 mph which is around $17 \sim 18$ m/s but most dogs are a lot slower. Regarding military applications—an F-16 fighter jet's speed is around 1400 mph, which is around 600 m/s, whereas the fastest missile's speed is around 5 machs, which is around 1700 m/s.

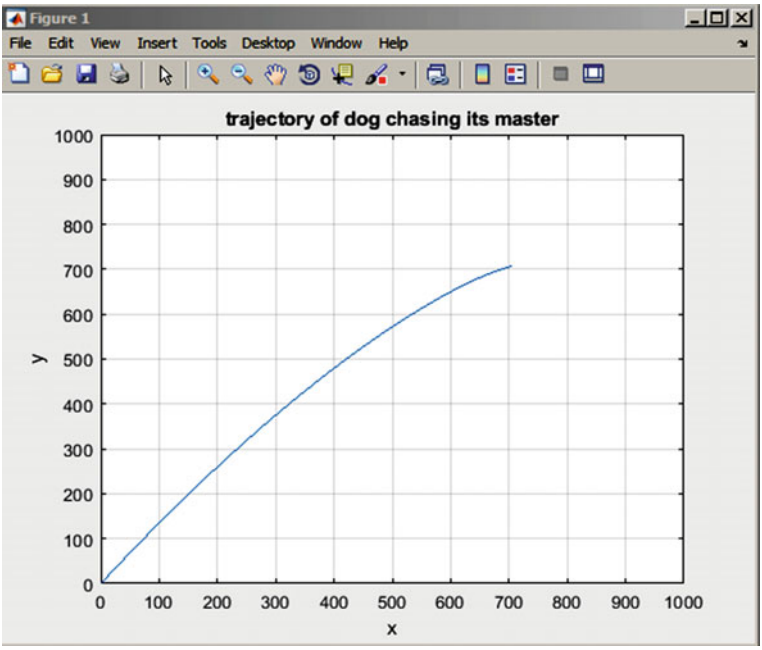


Fig. 6.10 *XY* graph of dog's motion starting at $(x, y) = (0,0)$ and ending at the master's coordinates as $R(t) = 0$

6.1.3 Case Study: Aircraft Landing Modeling and Control

This example too is based on the lecture notes of Prof. Eliezer Schoen (Technion, Israel Institute of Technology) [3].

When needs to automatically land an aircraft, using available feedback measurements. The part of the plane's autopilot that is discussed in this case study is the one that guides the plane to enter a landing runway at a plane parallel to the ground elevated some height from the ground. The actual descent and touchdown procedure is a follow-up control problem that requires other control means and other control models. The autopilot needs to receive information from transmitters located on the ground at the far edge of the runway. The transmitted information includes the angle ε and the range R (see Fig. 6.11). Let us also assume that the geographic orientation of the landing runway is known, and therefore, by using gyros residing on the plane, the angle ψ and its rate of change $\mathrm{d}\psi/\mathrm{d}t$ can both be measured in real time.

The aircraft needs to be automatically guided to enter the landing runway at angles that are close to $\varepsilon = \psi = 0$ and at a distance from the runway axis which is as small as possible.

The autopilot takes over at an initial range of R_0 and initial angles of ε_0 and ψ_0 while the plane flies along a straight line $(\mathrm{d}\psi/\mathrm{d}t(0) = 0)$ parallel to the ground. To

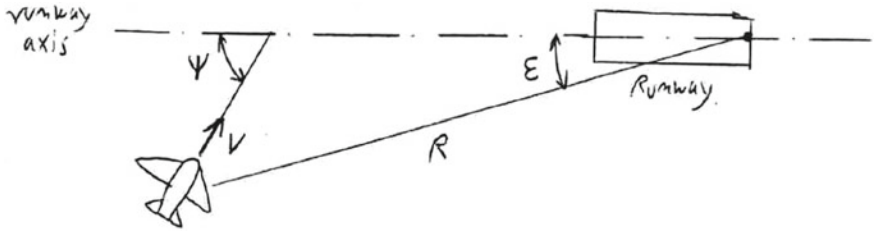
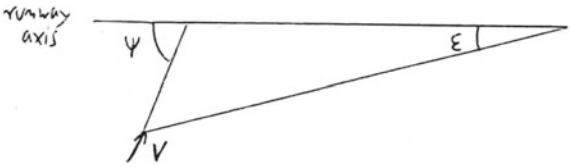


Fig. 6.11 Top view of the aircraft planar motion towards the landing runway prior to descent

Fig. 6.12 Planar geometry of the plane motion



simplify the task, let us assume that the plane moves at a constant speed V. The schematic drawing shown in Fig. 6.12 describes the horizontal geometry of the problem.

The radial component of the velocity vector V (the one that is pointing along the direction of R) is $V\cos(\psi - \varepsilon)$. This velocity component acts to reduce R. The component of the velocity vector V that is vertical to R is $V\sin(\psi - \varepsilon)$. This component acts to reduce ε.

The motion equations are therefore

$$\frac{\mathrm{d}R}{\mathrm{d}t} = -V\cos(\psi - \varepsilon) \tag{6.8}$$

$$R\frac{\mathrm{d}\varepsilon}{\mathrm{d}t} = -V\sin(\psi - \varepsilon) \tag{6.9}$$

For the airplane to make a turn in a horizontal plane, it needs to roll, at an angle ϕ about its body axis, as shown in Fig. 6.13.

The reason for rolling the body of the aircraft is to create a lift force L in a direction which is not equal to the Earth gravity direction. One vector component of L balances the aircraft's weight. The other vector component of L creates a centripetal force that causes the airplane to steer in a horizontal plane.

$$L\cos\phi = mg \tag{6.10}$$

$$L\sin\phi = \frac{mV^2}{r} = -mV\frac{\mathrm{d}\psi}{\mathrm{d}t} \tag{6.11}$$

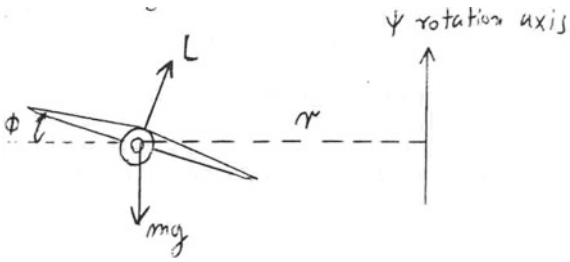


Fig. 6.13 Front view of the aircraft roll motion in order to do horizontal turning

The negative sign in the second equation above comes from an arbitrary definition of the positive direction of the rotation ϕ.

Dividing the above equations by one another yields the relationship:

$$\frac{\mathrm{d}\psi}{\mathrm{d}t} = -\frac{g}{V}\tan(\phi) \tag{6.12}$$

It is interesting to note that $\mathrm{d}\psi/\mathrm{d}t$ is independent of the lift force L.

Let the aircraft's steering angle command be denoted as δ, and let us assume that the relationship between the roll angle ϕ and the steering angle δ is according to the following second-order differential equation:

$$J\frac{\mathrm{d}^2\phi}{\mathrm{d}t^2} + C\frac{\mathrm{d}\phi}{\mathrm{d}t} = K\delta \tag{6.13}$$

where J is the aircraft's moment of inertia about the roll axis, C is an aerodynamic friction coefficient, and K is a constant gain. In the above equation, $T = J/C$ is the aircraft's time constant for the roll motion. We may also define: $K_1 = K/C$, and rewrite (6.13) as

$$T\frac{\mathrm{d}^2\phi}{\mathrm{d}t^2} + \frac{\mathrm{d}\phi}{\mathrm{d}t} = K_1\delta \tag{6.14}$$

Let us assume that it is given that $T = 0.5$ s.

It is proposed that the autopilot's feedback law be

$$K_1\delta = \alpha\varepsilon + \beta\psi + \gamma\frac{\mathrm{d}\psi}{\mathrm{d}t} \tag{6.15}$$

where α, β, and γ are gains that need to be determined by the control designer.

In this example, we assume that $R_0 = 10$ km, $\psi_0 = 0.4$ rad and $\varepsilon_0 = 0.3$ rad.

We further assume that $V = 200$ m/s (even though this is not a true aircraft landing speed).

As can be seen from Eqs. (6.8) to (6.15), the system is highly nonlinear and so far a student studying this book may have no theoretical basis as to how to optimally (or at least reasonably) select the control gains. In this case, the study lets us

make a simplifying assumption that prevents a totally blind trial-and-error approach to the feedback gains selection.

It is assumed that an experienced engineer estimates that the resulting design parameters should probably lie in the following ranges:

$$-10 \le \alpha \le -8$$
$$3 \le \beta \le 4$$
$$10.5 \le \gamma \le 11.5$$

The designer's job is to search for a suitable set of values of α, β, and γ that result in the best landing performance. The advice by the experienced engineer is not binding, but one will be foolish to ignore it. It is determined that a "good" entry to the landing runway is one happening at any distance that falls within ± 10 m from the runway center axis. Therefore, one of the output variables of the simulation must be the vertical distance of the aircraft from the runway axis as a function of t.

It is given that the runway length is about 2 km.

As a first step in the design, the whole system must be described as a block diagram. In doing so, the designer must understand the cause–effect relationship among the variables. This is done as follows.

In an open-loop system, the steering angle (δ) would be the system's external input. The variable δ influences φ (the roll angle) via a second-order Eq. (6.13). Variable φ affects the variable ψ (the plane planar rotation) via Eq. (6.12). Variable ψ affects both R and ε (Eqs. 6.8 and 6.9). Angle ε affects R (Eq. 6.8).

In a closed-loop system, δ becomes a function of some of the system's measured feedback sensors signals that are measured in real time. This is the feedback law. The closed-loop system then operates autonomously—it is driven only by the system's initial conditions.

This concludes Problem Formulation, and the related modeling theory. Each semester, slightly different initial conditions or parameters such as plane velocity were assigned.

Solution (Benjamin Coleman, Fall 2016, Florida Atlantic University)

The problem geometry is redrawn in Fig. 6.14. The block diagram is as shown in Fig. 6.15. Initial conditions R_0, ε_0, and ψ_0 are shown in blue. Display blocks (used for controlling and plotting the output of the simulation) are shown in purple—these simulation blocks could be removed without changing the mathematic behavior of the system. The design parameters are shown in salmon.

To estimate how long the plane will take to go from $R = R_0$ to $R = R_F = 2$ km, we can approximate the planes trajectory using straight lines. The quickest path will be a straight line directly towards the destination. If this is the case then the time will be

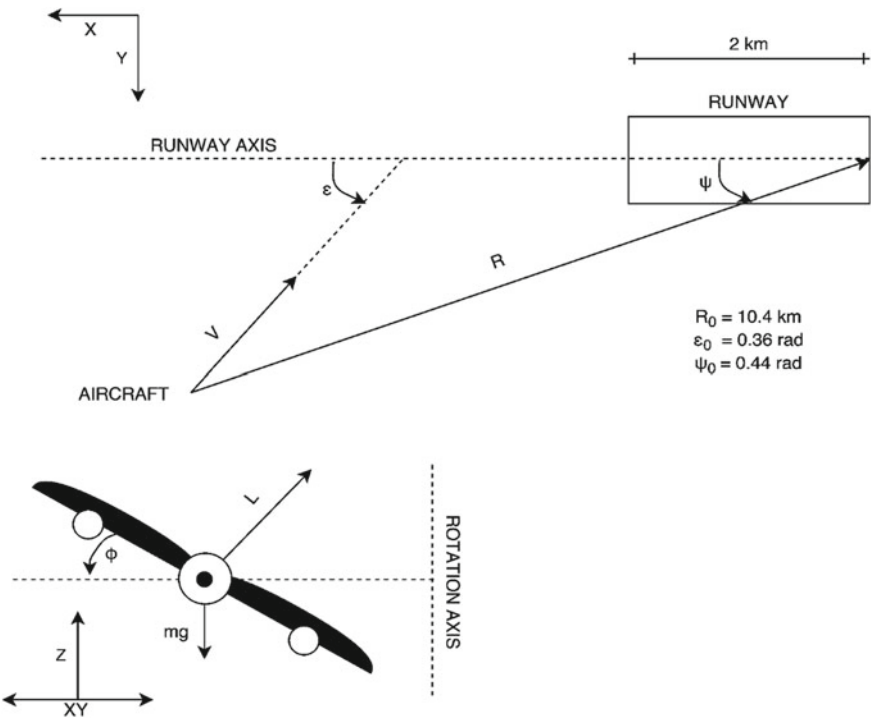


Fig. 6.14 Problem geometry and set of initial conditions

$$t_{\min} = \frac{R_0 - R_f}{V} = \frac{8.4\,\text{km}}{200\ \frac{\text{m}}{\text{s}}} \approx 42\,\text{s}$$

The plane will most likely take longer than this, however. Looking at the diagram in the problem statement, the longest feasible path would be to first go in the *y*-direction and then move along the runway axis. Of course, this is a ridiculous and nonphysical trajectory, but it will give an upper bound on the simulation time. This value will be

$$t_{\max} = \frac{R_0 \sin \varepsilon_0 + R_0 \cos \varepsilon_0 - R_f}{V} \approx 57\,\text{s}$$

It will probably suffice to use 50 s as the simulation time.

To determine the optimal set of system gains, we start right in the middle of the range of the parameters given by the experienced engineer, at $\alpha = -9$, $\beta = 3.5$, and $\gamma = 11.0$. The results are the graphs of ε, ψ, and R shown in Fig. 6.16.

These simulations show that the control system is definitely successful in bringing ε and ψ down to very small values while causing R to decrease steadily down to 2 km (the entry point of the runway). The two parameters that are of

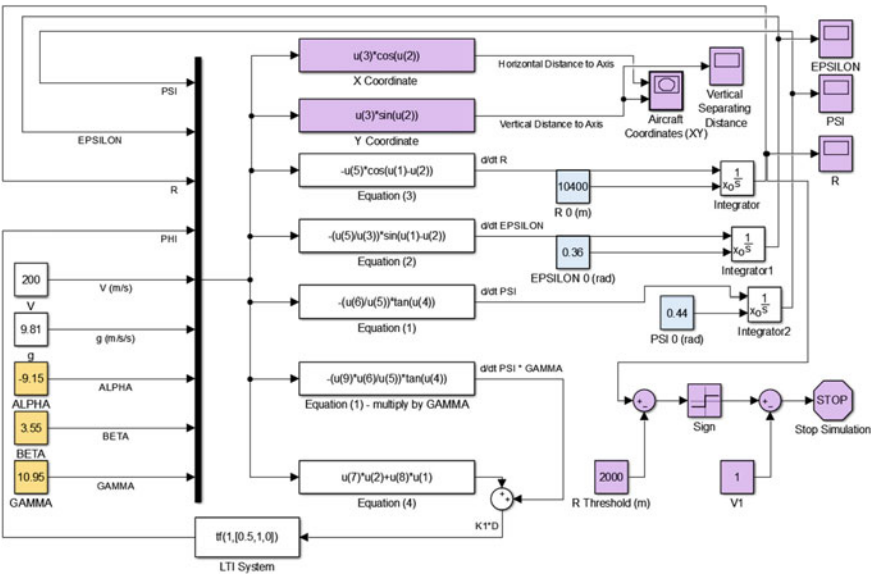


Fig. 6.15 Simulink block diagram of the aircraft landing model

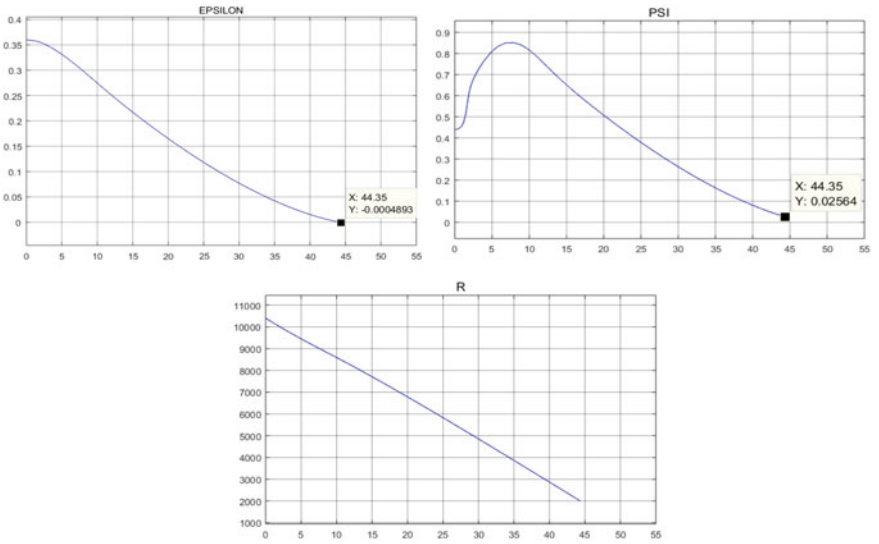


Fig. 6.16 Simulation results for midrange control parameter values

particular importance in this design are ε, the angle at which the aircraft approaches the runway, and R_y, the vertical distance from the runway axis when runway entry occurs.

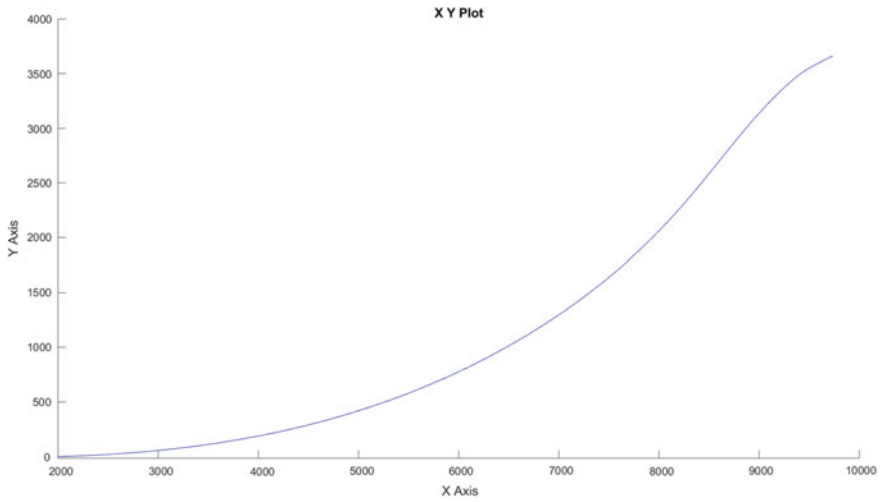


Fig. 6.17 Plane trajectory for mid-value control coefficients

The overall trajectory of the plane can be found by plotting the vertical and horizontal components of the plane location on the same *XY*. The result is a parametric function of t showing the exact location of the plane (Fig. 6.17). The runway starts at the $x = 2000$ m marker and the plane starts at the initial location of about (3500, 9500).

To get the vertical distance from the axis, the value of the *Y*-axis from the previous plot can be plotted against time (Fig. 6.18). This plot shows that when the plane is within 2 km of the landing point, it is within 1 m of the runway axis. This

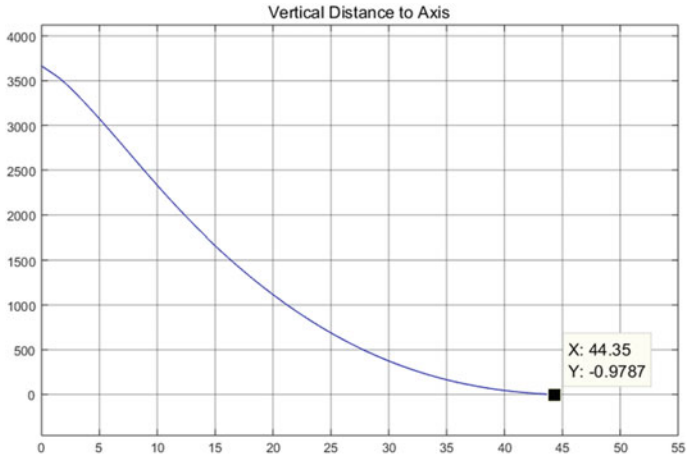


Fig. 6.18 Vertical distance from the runway axis for mid-value control parameters

Table 6.1 Output parameters versus design parameters

α	β	γ	$R_{y\text{Final}}$ (m)	ψ_{Final} (rad)
−9	3.5	11	−0.9787	0.02564
−10	3.5	11	−5.557	0.004899
−9	4.0	11	30.83	0.07109
−9	3.5	11.5	−4.013	0.02317

is a sufficiently good landing, since a good landing is considered within $\pm$10 m of the axis and with an angle of approach close to 0. The angle of approach is 1.5° and the plane is within 1 m of the runway.

The initial design could be considered safe, but it might be possible to improve performance for both of the important variables—the vertical distance error R_Y and the angle of approach ψ. This model is nonlinear, but it would still be beneficial for the design purpose to know what effect each parameter—α, β, and γ—has on the output parameters. To do this, we will vary each parameter while holding the others constant and make a table. Note that ε and R_Y essentially represent the same information, since R_Y is a function of ε and t only.

Increasing α causes the angle of approach to go toward zero, while increasing the overshoot of the runway axis. Increasing β results in a serious undershoot R_Y and also increases the angle of approach. Increasing γ causes the angle of approach to stay about the same while decreasing slightly the angle of approach (Table 6.1). Assuming relatively consistent behavior in the range of parameters that is considered, this leads to a design approach where α is set at a high value to decrease ψ, γ is decreased to fix the overshoot, and β is increased only a slight amount.

Following this process and iterating many times and selecting the best set of coefficients yield good results. For the sake of brevity, the ten intermediary results are not shown, but it is important to note that the design rules in the previous paragraph ended up being correct over the range of parameters suggested by the senior engineer. This was useful for finding the optimal solution:

$$\alpha = -9.6 \quad \beta = 3.53 \quad \gamma = 10.2$$

These gains result in the best overall performance out of all of the combinations that were tried. The optimized performance has an entry angle of 1° and an overshoot of less than 0.2 m. This is shown in Fig. 6.19.

The values of ε and R_Y are coupled but are both shown for completeness (Fig. 6.20). They are both very small. Finally, the trajectory looks appropriate, so the final design with $\alpha = -9.6$, $\beta = 3.53$, $\gamma = 10.2$. Initially, there may be some concern that the rapid change in φ might be a problem, but the XY plot shows that the aircraft actually takes a very smooth path to the landing strip.

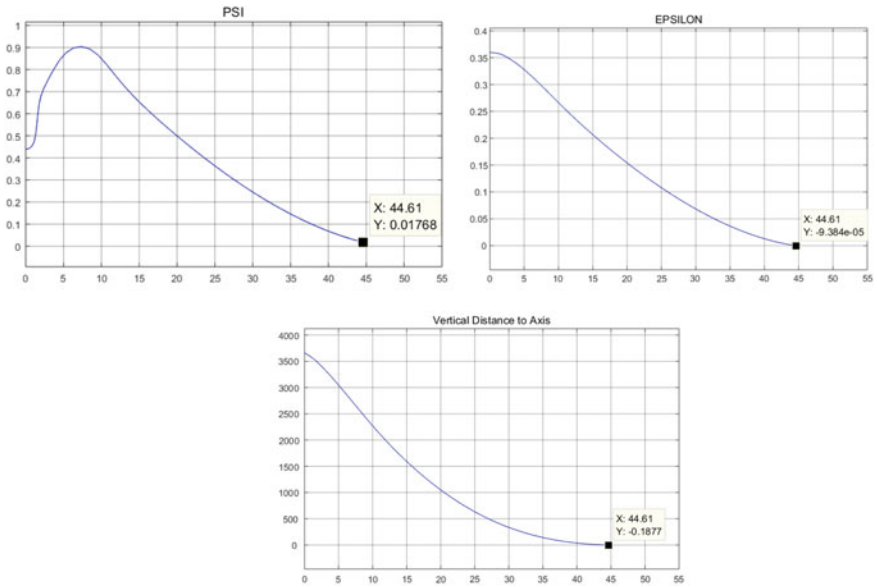


Fig. 6.19 Optimized performance

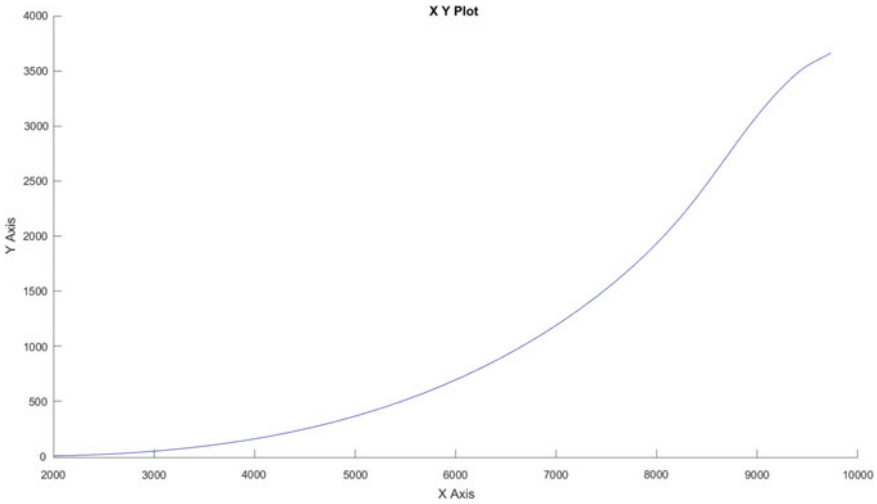


Fig. 6.20 Plane trajectory for optimized control parameters

6.2 More Advanced Aspects of MATLAB Simulink

What is considered "a basic Simulink analysis", as demonstrated in the previous section, involves single Simulink runs, each is initiated from either the Simulink model itself, or from a controlling MATLAB script. A "more advanced analysis" (as well as design) Simulink tool may involve multiple Simulink runs all initiated from a single MATLAB script.

One of the control analyses needs that may necessitate multiple Simulink runs is to present *XY* graphs with multiple solution trajectories starting each at a different initial state. Such a portrait of phase plane solutions (as it is called for some second-order systems) is often more insightful than the observation of multiple time solutions of the state variables.

The demand for multiple Simulink runs often goes far beyond mere presentation needs. When multiple Simulink runs, where each run is done with a different set of system model parameters, facilitate the task of system identification. Some outputs of each run may be compared against actual output signals obtained from real physical measurements to determine which of the models yields outputs that are closest to the actual results. Multiple runs may be needed to explore different control law parameters, or different control strategies.

A MATLAB script that controls multiple Simulink runs can accumulate the results from all the runs, for the sake of postprocessing. Consider, for example, an irreversible chemical reaction in which the rate of product concentration build-up is proportional (by the Law of Mass Action) to the product of the concentrations of some of the reactants. The proportionality constant k, known as the forward reaction rate constant, may be unknown. If one knows one or more data points (say, reactant A decreases to a certain concentration at a given time), then multiple Simulink runs, each with a different value of k, can help identify the value of k that causes the reactant concentration time curve to be closest to the given data points.

Multiple simulations that can be done much faster than the system's real-time scale enable an off-line assessment of many simulated control strategies, and adoption of the best one.

6.2.1 Multiple Simulink Runs of Two-Species Growth Models

The well-known predator–prey model (formulated by Lotka and Volterra in $1925 \sim 1926$) is a simple population growth model of two interacting species [4]. The rate of interaction between the two species is assumed to be proportional to the product of the two population sizes. That rate term is positive for the predator equation and negative for the prey equation. We denote the prey's population by x and the predator's by y. Without interaction (that is whenever $a = b = 0$), the

prey's population grows exponentially as the per-capita growth rate r is positive, and the predator's population decays exponentially as $m > 0$.

$$\begin{aligned} \frac{dx}{dt} &= rx - axy \\ \frac{dy}{dt} &= -my + bxy \end{aligned} \tag{6.16}$$

A Simulink model is shown in Fig. 6.21.

The MATLAB script shown in Fig. 6.22 runs the Simulink model to obtain a phase plane plot of the two populations, starting at some arbitrary initial populations. The script allows the relative tolerance simulation parameter to be adjusted.

The adjustment starts by running the model for the default relative tolerance 1e-3. The simulation result is shown in Fig. 6.23 (left). The result appears to show a transient of a few cycles converging to a closed trajectory that is an oscillation. Decreasing RelTol to 1e-4 reveals a different plot. As long as the results keep changing, the RelTol parameter is considered to be not small enough. Only when the result stops being sensitive to the selection of the simulation parameter, such a

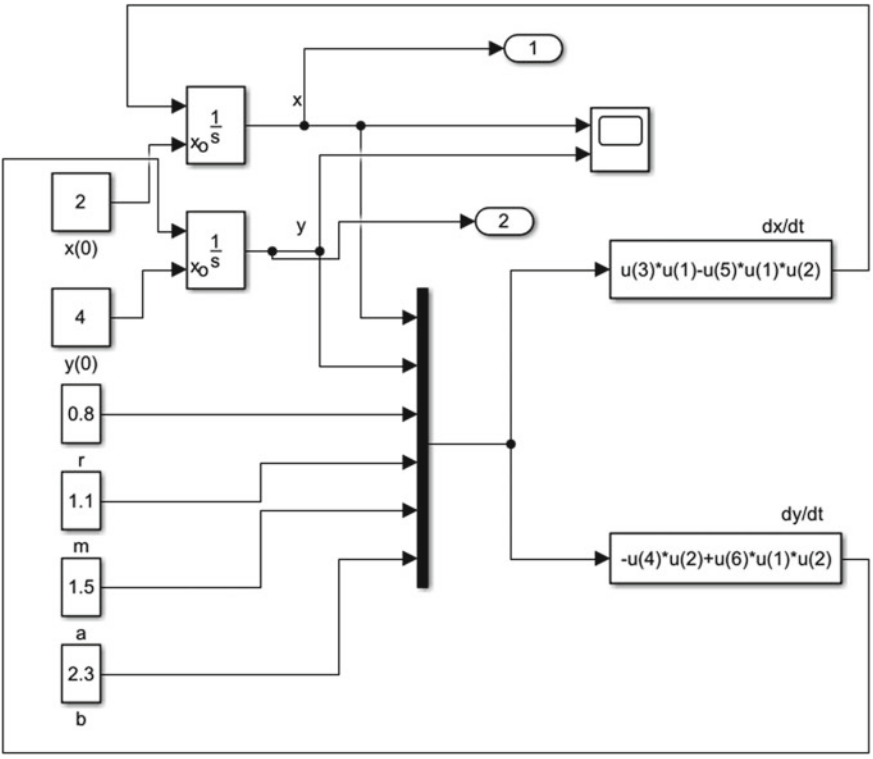


Fig. 6.21 Simulink model of the basic predator–prey model

bme_5742_sp18_predator_prey_012518.m

```
options=simset('RelTol',1e-7);
[t,x,y]=sim('bme_5742_sp18_predator_prey_M_012518',80,options);
plot(y(:,1),y(:,2));
xlabel('x');
ylabel('y')
```

Fig. 6.22 MATLAB script for running of the basic predator–prey model for obtaining a phase plane solution

parameter value is considered acceptable. Figure 6.23 (right) shows the correct simulation result obtained for RelTol = 1e-7.

As seen in Fig. 6.23 (right), there is no transient. The system oscillates periodically right from the start. The time-domain oscillations are shown in Fig. 6.24.

There are two more basic two-species growth models. Our next example presents a Competition Model in which the species interaction term affects negatively the rate of change of each population growth. The other (Symbiosis Model) is covered in the end-of-chapter problems.

One of the most basic forms of a two-species competition model involves logistic growth for each species, in the absence of interspecies interaction, and negative growth rates due to the interaction, and these terms are proportional to the product of the two populations:

$$\begin{aligned}\frac{dx}{dt} &= r_x x\left(1-\frac{x}{K_x}\right) - b_{xy}xy \\ \frac{dy}{dt} &= r_y y\left(1-\frac{y}{K_y}\right) - b_{yx}xy\end{aligned} \tag{6.17}$$

With no species interaction ($b_{xy} = b_{yx} = 0$), each species has its own independent logistic growth. The r coefficients are called "the per-capita growth rate at

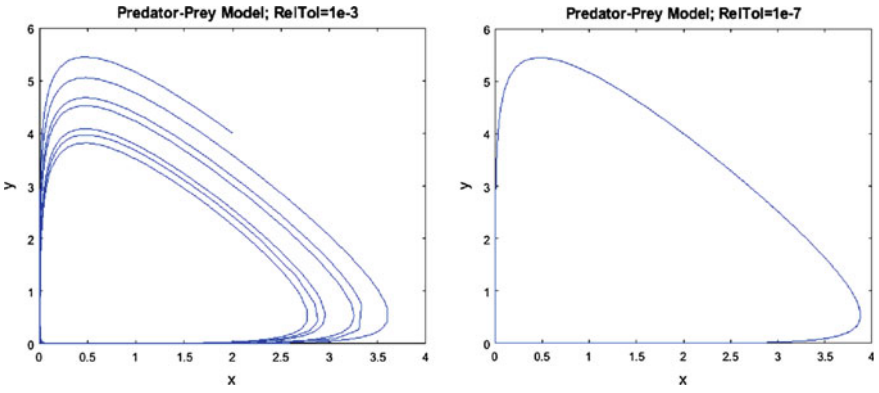


Fig. 6.23 Phase plane trajectory for the predator–prey model starting at (x_0, y_0) = (2, 4) with RelTol = 1e-3 and with RelTol = 1e-7

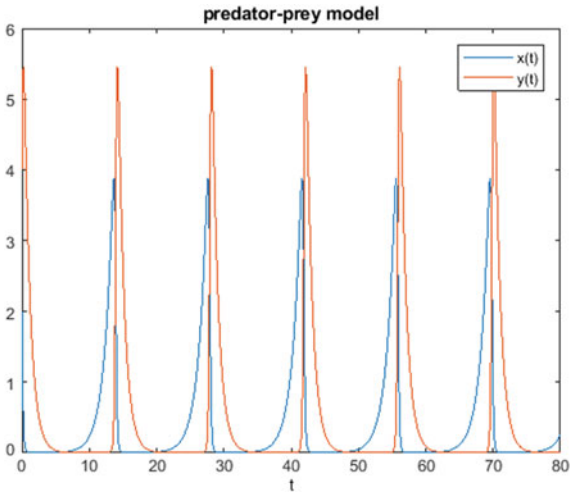


Fig. 6.24 Time-domain solutions of the predator–prey model for specific initial conditions

small value of the population" and the K coefficients are called "the carrying capacity of the species population". A logistic model has two equilibrium points (at which, for instance $dx/dt = 0$). One of the equilibrium points $x = x_{e1} = 0$ is unstable, and the other $x = x_{e2} = K_x$ is stable. In other words, a population that obeys a logistic model converges to its carrying capacity from any initial condition. If $x(0) \ll K_x$, the beginning of the population growth $x(t)$ resembles an exponentially increasing curve. This exponential growth tapers later as $x(t)$ approaches the carrying capacity [4].

When the interaction coefficients are nonzero, the species compete with one another. When this happens, most parameter values and initial condition values follow what is known as "Principle of Competitive Exclusion"—one of the species wins, and proceeds to converge to its carrying capacity, whereas the other species loses and becomes extinct.

A Simulink model of system (6.17) for some arbitrary choices of coefficient values and initial conditions is given in Fig. 6.25.

Let us explain the reasons for including the computations of the a_{12} and a_{21} coefficients in the above model (referring again to [4]).

In general, any model can be normalized by some smart magnitude and time scaling. Normalization of the system can reduce the number of parameters to a minimum, which is very helpful for comprehensive parametric studies. How to perform normalization is often an art, and it is beyond the scope of this book. We refer to [4] for the normalizing details of the competition model. The normalizing of (6.17) is done by scaling each population by its own carrying capacity, and scaling time according to one of the species "birth rates":

$$u_x = \frac{x}{K_x} \quad u_y = \frac{y}{K_y} \quad \tau = r_x t \tag{6.18}$$

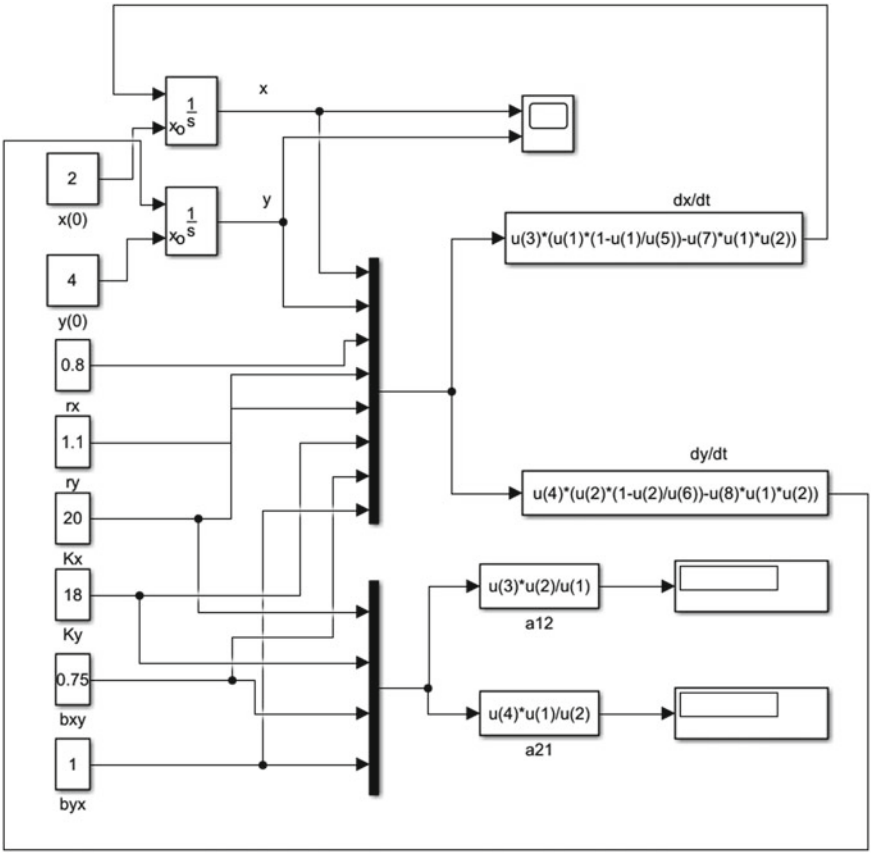


Fig. 6.25 Simulink model of the basic competition model

As a result, one can show that the number of system parameters reduces from 6 to 3. The normalized model and its parameters are shown below:

$$\begin{aligned} \frac{\mathrm{d}u_x}{\mathrm{d}\tau} &= u_x(1 - u_x - a_{12}u_y) \\ \frac{\mathrm{d}u_y}{\mathrm{d}\tau} &= \rho u_y(1 - u_y - a_{21}u_x) \end{aligned} \tag{6.19}$$

where

$$\rho = \frac{r_y}{r_x} \quad a_{12} = b_{xy}\frac{K_y}{K_x} \quad a_{21} = b_{yx}\frac{K_x}{K_y} \tag{6.20}$$

It can be shown that system (6.19), and consequently the equivalent unnormalized system (6.17), has four equilibrium states. Three of which are: (0,0) which

is always unstable, (1,0) which is stable if $a_{21} > 1$ and it is unstable if $a_{21} < 1$ and (0,1) which is stable if $a_{12} > 1$ and is unstable if $a_{12} < 1$. The fourth equilibrium state is $\left(\frac{1-a_{12}}{1-a_{12}a_{21}}, \frac{1-a_{21}}{1-a_{12}a_{21}}\right)$. Obviously, the states (0,1) and (1,0) correspond with the principle of competitive exclusion, whereas the fourth equilibrium state hints that under certain circumstances ($a_{12} < 1$ and $a_{21} < 1$, and for some values of ρ), the two species can coexist in what is called "a nonaggressive competition". The Simulink model (Fig. 6.25) therefore exhibits four different modes of behavior: (a) Species x wins for any initial conditions, (b) Species y wins for any initial conditions, (c) Either species may be the winner if the initial conditions are right, and (d) the two species coexist. That is the reason for including the normalized interaction coefficients, which facilitates the parameter selection for each behavior mode.

In Fig. 6.26 we slightly revise the Simulink model, assigning algebraic values to some of the parameters. Such values become global parameters in a MATLAB

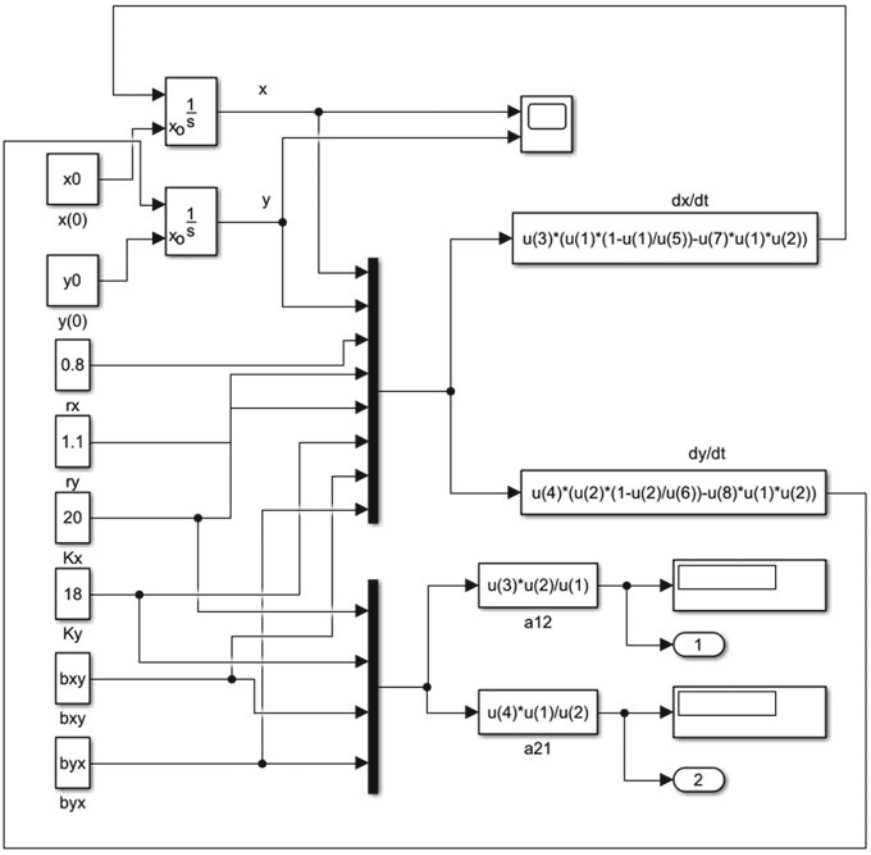


Fig. 6.26 Simulink model in which the initial conditions are assigned algebraically

bme_5742_sp18_predator_prey_012518.m | bme_5742_sp18_competition_b_param_020518.m

```
for x0=4:1:6
    for y0=2:1:4
[t,x,y]=sim('bme_5742_sp18_competition_XY_IC_020218',20);
plot(x(:,1),x(:,2))
hold on;
    end
end
title('competition model')
xlabel('x')
ylabel('y')
a12=max(y(:,1))
a21=max(y(:,2))

```

Fig. 6.27 Script for plotting multiple phase plane trajectories for one of the competition model cases

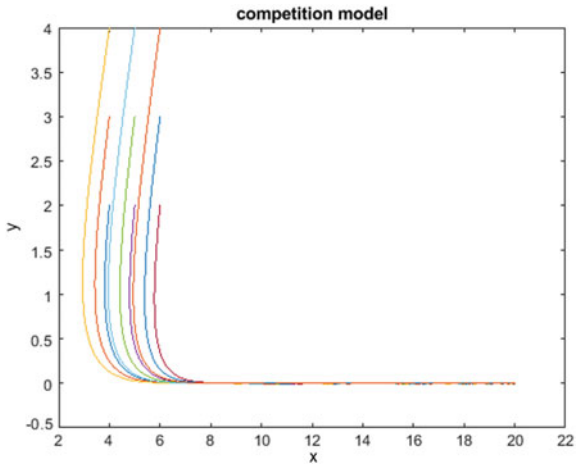


Fig. 6.28 Multiple competition model phase plane trajectories, each starts at a different initial state

script that assigns numerical values to such parameters and runs that specific Simulink model. The parameters in Fig. 6.26 that become algebraic are the integrators' initial conditions as well as the unnormalized interaction coefficients. This is done to first allow the varying of such initial conditions in the script that initiates multiple runs of the model (Fig. 6.27). As a result, an *XY* plot of the species can now exhibit a portrait of phase plane trajectories, as shown in Fig. 6.28. A script in which the initial state is fixed but one of the interaction parameters is varied is given

Editor - Z:\profile_documents\My Documents\MATLAB\bme_5742_sp18_competition_b_param_020518.m

bme_5742_sp18_predator_prey_012518.m | bme_5742_sp18_competition_b_param_020518.m

```
x0=10;
y0=3;
byx=0.2;
n=0;
for bxy=0.5:1:3.5
    opt=simset('RelTol',1e-6);
[t,x,y]=sim('bme_5742_sp18_competition_b_020518',20,opt);
plot(x(:,1),x(:,2))
%axis([0,22,0,22])
hold on;
n=n+1
a12=max(y(:,1))
end
title('competition model with variable bxy')
xlabel('x')
ylabel('y')
a21=max(y(:,2))
hold off
```

Fig. 6.29 Script for parametric study of one of the competition model coefficients

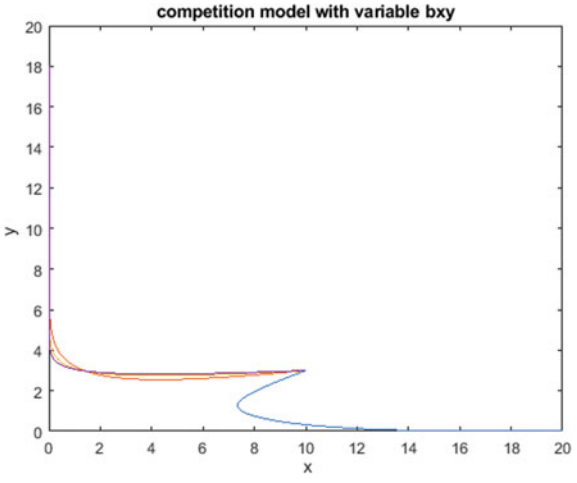


Fig. 6.30 Phase plane trajectories for the competition model as one of the interaction coefficients is varied. The portrait shows two different modes of behavior of the competition model

in Fig. 6.29, with the multiple run results shown in Fig. 6.30. A correct interpretation of such portrait of plots requires the finding of the color code that is used, as well as a list of the respective normalized coefficients for each run included in the MATLAB command line (Fig. 6.31).

```
>> bme_5742_sp18_competition_b_param_020518

n =

     1


a12 =

    0.4500


n =

     2


a12 =

    1.3500


n =

     3


a12 =

    2.2500


n =

     4


a12 =

    3.1500
```

Fig. 6.31 Computation results for multiple Simulink runs shown on the MATLAB command line

6.2.2 *Case Study: Parameter Identification in a Spread of Infectious Diseases and Determination of Immunization Control Strategy*

Many models have been introduced over the years to model the spread of infectious diseases. Some are very simple, having very few population categories. Other models are more complicated. We shall refer to [5]. The simplest model is the SI model consisting of susceptible $S(t)$ and infected $I(t)$ individuals. For a short duration disease, one may assume that the total population N is constant. Thus,

$$S(t) + I(t) = N \tag{6.21}$$

It is assumed that the rate of disease contact is proportional to the product of the susceptible and infected populations. Therefore, the model equations are

$$\frac{\mathrm{d}S}{\mathrm{d}t} = -\beta IS \quad \frac{\mathrm{d}I}{\mathrm{d}t} = \beta IS = \beta I(N - I) = (\beta N)I\left(1 - \frac{I}{N}\right) \tag{6.22}$$

The coefficient β is called "the pairwise infectious contact rate". This coefficient is usually unknown as it varies not only from disease to disease but even for the same disease, depending on the geographic location and many factors that affect people mutual contacts.

Due to the constraint (6.21), the variables S and I are dependent and thus the model (6.22) is one-dimensional. We observe that $I(t)$ obeys a logistic growth model in which the birth rate is $r = \beta N$ and the carrying capacity $K = N$. In other words, in a disease modeled by the SI model, the entire population becomes infected over time. The SI model contains no recovery from disease possibility.

The simplest models that include a recovered population are the SIS and SIR models. In the SIS model, infected individuals may recover with no immunity at a rate that is proportional to the number of infected individuals. They immediately become susceptible again. In the SIR model, individuals recover with immunity.

The SIS model is

$$\frac{\mathrm{d}S}{\mathrm{d}t} = -\beta IS + \gamma I \quad \frac{\mathrm{d}I}{\mathrm{d}t} = \beta IS - \gamma I \tag{6.23}$$

The coefficient γ is called the "recovery rate". Due to the constraint (6.21), the SIS model is one-dimensional as well.

$$\frac{\mathrm{d}I}{\mathrm{d}t} = \beta I(N - I) - \gamma I = \gamma(R_0 - 1)I\left(1 - \frac{I}{\frac{(R_0-1)N}{R_0}}\right) \tag{6.24}$$

where

$$R_0 = \frac{\beta N}{\gamma} \tag{6.25}$$

R_0 is called "the reproductive ratio" and it determines two distinct modes of behavior of the SIS model: (a) If $R_0 < 1$, the model (6.24) has a stable equilibrium $I_{e1} = 0$. As the number of infected individuals goes to zero, the disease becomes eradicated, (b) If $R_0 > 1$, the model has a logistic growth to a carrying capacity of $(R_0 - 1)N/R_0$. The disease stays and it becomes what is called "endemic".

The SIR model can represent many diseases in which infected individuals become "recovered". There are three variables $S(t)$, $I(t)$, and $R(t)$. For short duration diseases, we assume the constraint $S(t) + I(t) + R(t) = N$, similar to (6.21). The SIR model (given below) is therefore two-dimensional.

$$\frac{dS}{dt} = -\beta IS \quad \frac{dI}{dt} = \beta IS - \gamma I \quad \frac{dR}{dt} = \gamma I \tag{6.26}$$

$$\frac{dS}{dt} = -\beta IS \quad \frac{dI}{dt} = \beta IS - \gamma I \quad R = N - S - I \tag{6.26'}$$

In the SIR model, the model stops being valid whenever $I \to 0$ (disease eradication) for some arbitrary value of $S(t)$ or whenever $S(t) \to 0$. The reproduction ratio defined in (6.25) plays an important role in determining whether $I(t)$ goes to zero monotonically (for $R_0 < 1$) or reaches a peak before converging to zero (for $R_0 > 1$). A Simulink model of (6.26′) is shown in Fig. 6.32.

The simulation stops whenever the number of susceptibles or infected falls below 1. As seen (above), a little precaution was exercised in the STOP condition for $I(t)$, as many diseases often start with $I(0) = 1$. A typical simulation result is given in Fig. 6.33.

A key challenge in all spread of infectious diseases models is the need to identify the model parameters (β and γ) based on available data. These coefficients are never given explicitly. What is given typically are graphs (or tables) that show the number of infected as function of time, or the number of deaths as function of time. A tuning of the β parameter, in particular, allows the fitting of the data to the known time frame for the disease spread and duration. In the above simulation (taking 1 s of simulation time to represent 1 day), the disease duration is 12 days.

A validated spread of disease model may be used in planning population vaccination strategies as shown in Fig. 6.34. The number of vaccinated individuals (set by one or more pulse generators) decimates directly the rate equation for the susceptibles.

If for instance, we start to vaccinate on Day 3, vaccinating 10 people, and then stopping the vaccination process, the property editor for the pulse generator can be set as: Amplitude = 10, period = 30, pulse width (% of the Period) = 3.33, and phase delay = 2. The generator creates a single pulse of width 1 day because the

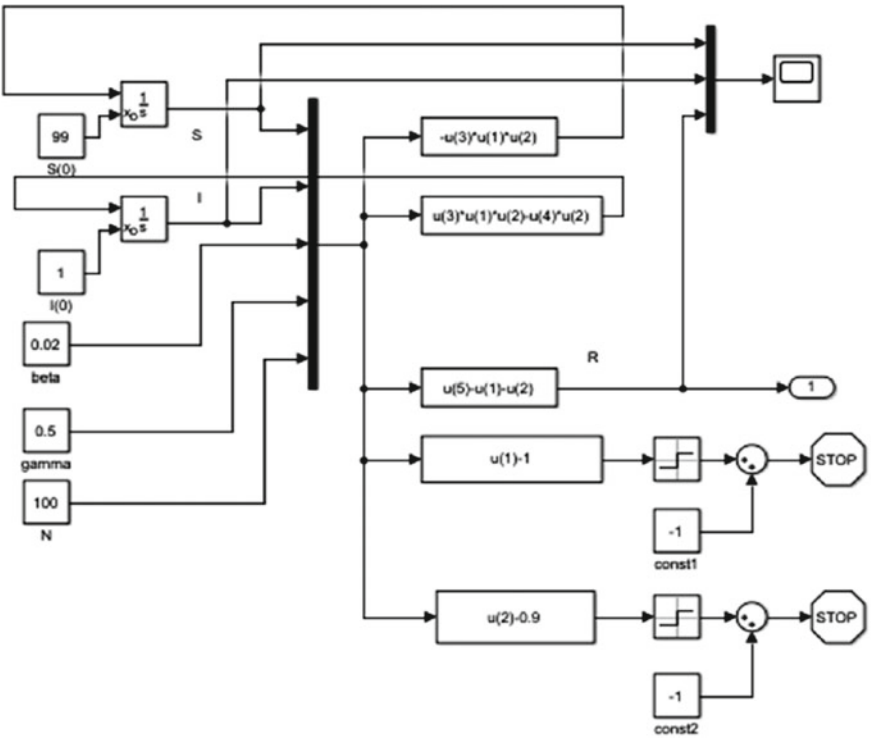


Fig. 6.32 Simulink diagram of an SIR model

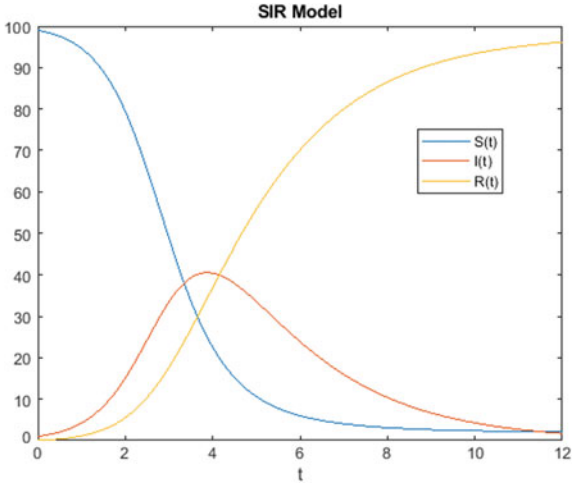


Fig. 6.33 SIR model results for $R_0 > 1$. Time units are (days)

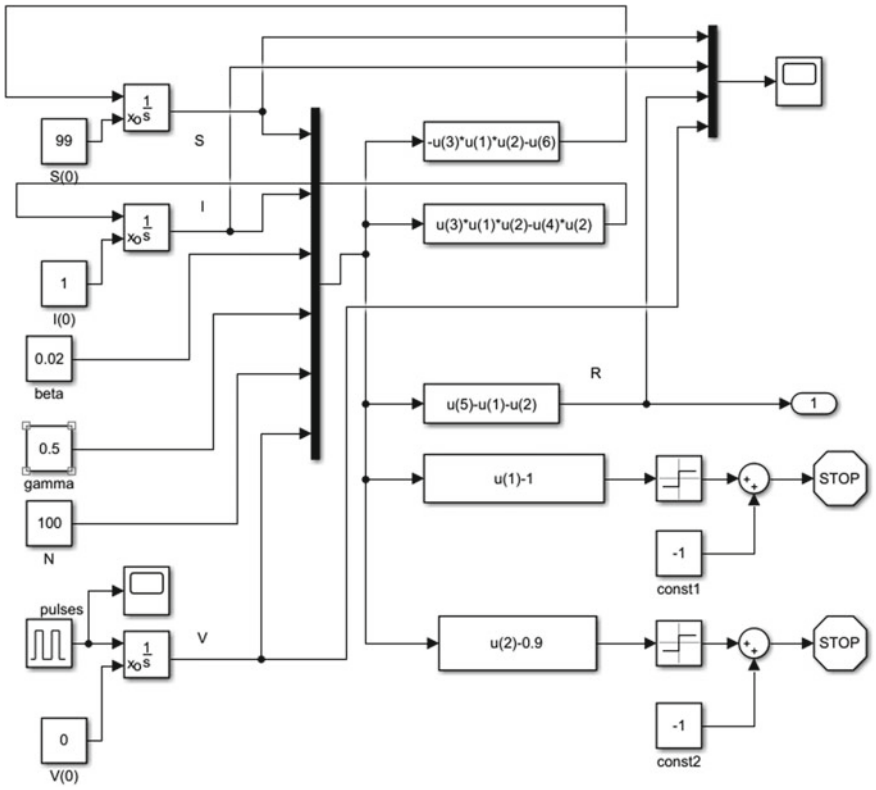


Fig. 6.34 SIR model with vaccination actions

period is set to a large value larger than the simulation final time. The results of such vaccination are shown in Fig. 6.35.

Another vaccination strategy is shown in Figs. 6.36 and 6.37, again starting at Day 2, vaccinating two people for half a day each day. The results of the simulation are shown in Fig. 6.38.

The basic SIR model being incapable of including mortality due to the disease and delay effects such as disease incubation often needs to be extended in order to model real-life data. We conclude this case study with a SIRD model identification for Ebola pandemic data.

Task: Need to refer to the US CDC web page about the 2014∼2016 Ebola Pandemic in West Africa (https://www.cdc.gov/vhf/ebola/outbreaks/2014-west-africa/index.html) [6]. There are graphs of the infected individuals as function of time (for the three West African countries of Liberia, Guinea, and Sierra Leone). One can actually download the spreadsheet that has the daily count of cumulative number of infections and cumulative number of deaths. The task is to pick up any one of the three countries and best fit to the given data a SIRD model with time delay effects.

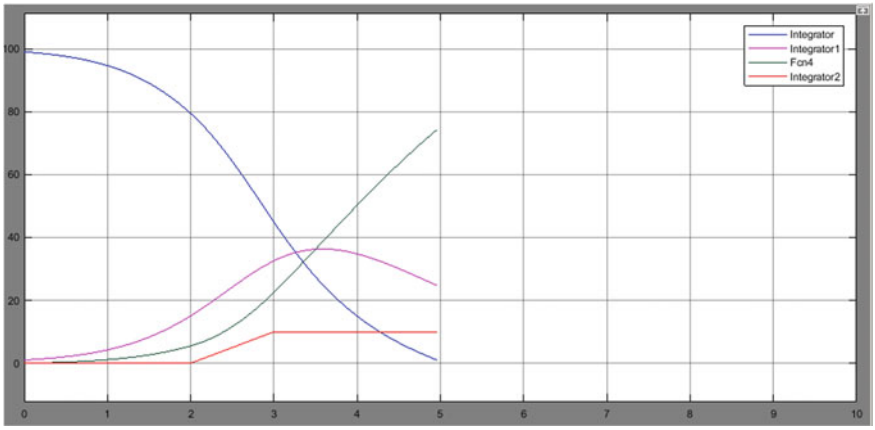


Fig. 6.35 The vaccinated individuals (red) cause a rapid reduction of $S(t)$ (in blue). The simulation stops on Day 5. Also shown $I(t)$ (in purple) and $R(t)$ (in green)

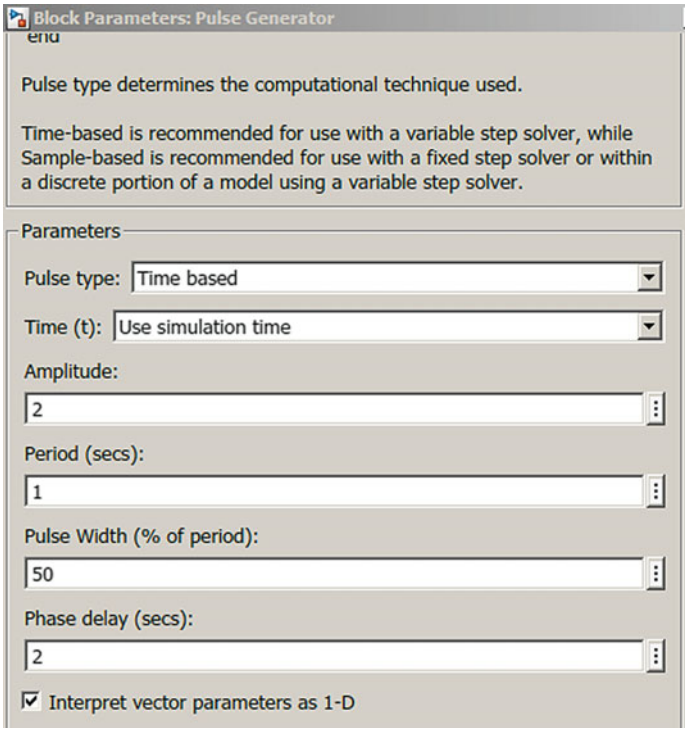


Fig. 6.36 Pulse generator set to generate periodic pulses with a delay of 2

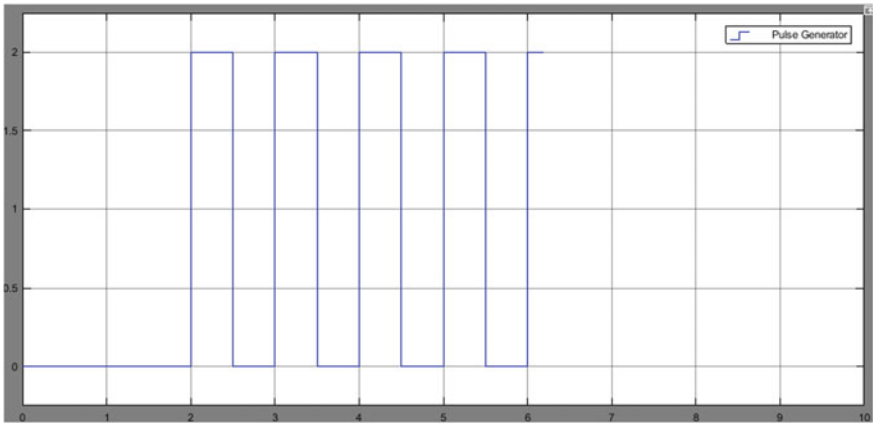


Fig. 6.37 The pulses generated according to Fig. 6.36

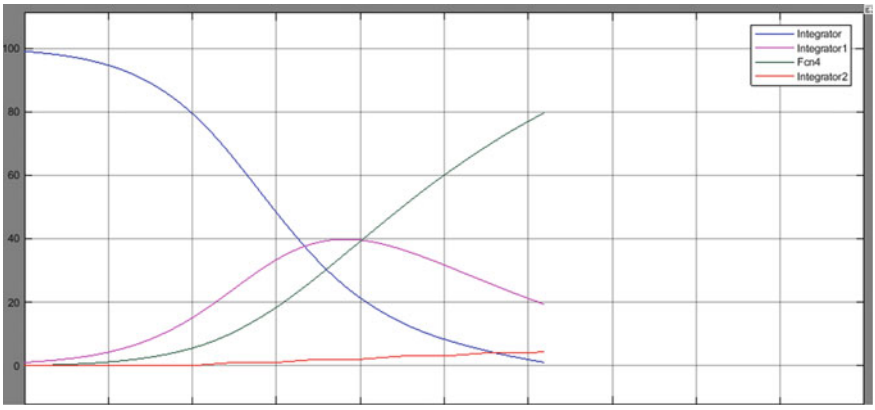


Fig. 6.38 SIR model results for periodic vaccinations (color code as in Fig. 6.34)

Solution (Dennis Estrada, Spring 2018 semester, Florida Atlantic University)

The raw data is shown in Fig. 6.39. It is a plot of the number of Ebola cases in the country of Guinea between March 2014 and February 2016 and a plot of the deaths during that period. A SIRD model can be formulated by first formulating a SI model for the total number of cases versus Time. The SI model is given in (6.21) ∼ (6.22). The model behaves as a logistic growth model with carrying capacity of N and an initial per-capita growth rate of βN. Both can be estimated from the data presented in Table 6.2. We see that $N = 3804$. In order to estimate the per-capita growth rate, one should use data points that are near $t = 0$.

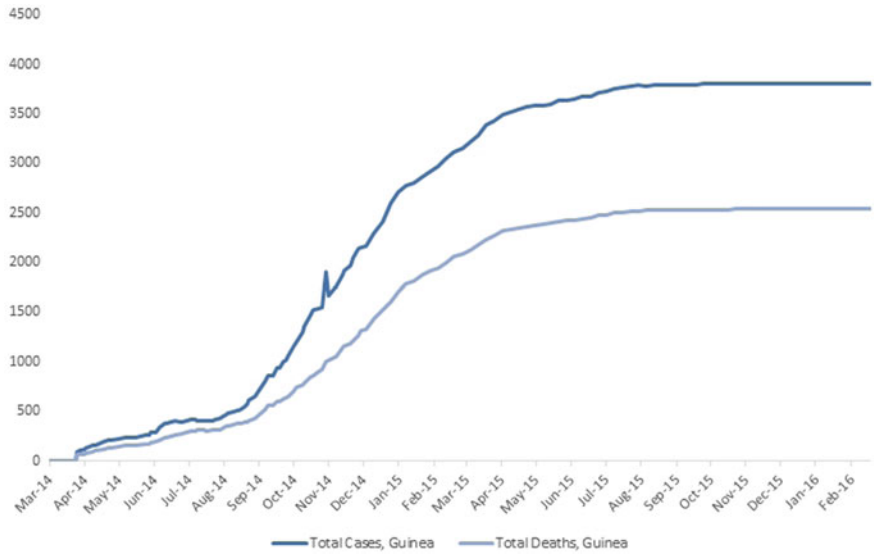


Fig. 6.39 Total reported Ebola cases and deaths versus time in Guinea

Table 6.2 Total reported Ebola cases and deaths versus time in Guinea

WHO report date	Total cases, Guinea	Total deaths, Guinea
3/25/2014	86	59
3/26/2014	86	60
3/27/2014	103	66
3/31/2014	112	70
4/1/2014	122	80
⋮	⋮	⋮
2/3/2016	3804	2536
2/10/2016	3804	2536
2/17/2016	3804	2536

Note the dates chosen for the calculation of the initial growth rate were 3/27/14 and 3/31/14.

$$\frac{\mathrm{d}I}{\mathrm{d}t} = \frac{112 - 103}{4} = 2.25 = (\beta N)103\left(1 - \frac{103}{3804}\right) = (\beta N)100.211$$

$$\beta N = \frac{2.25}{100.211} = 0.02245$$

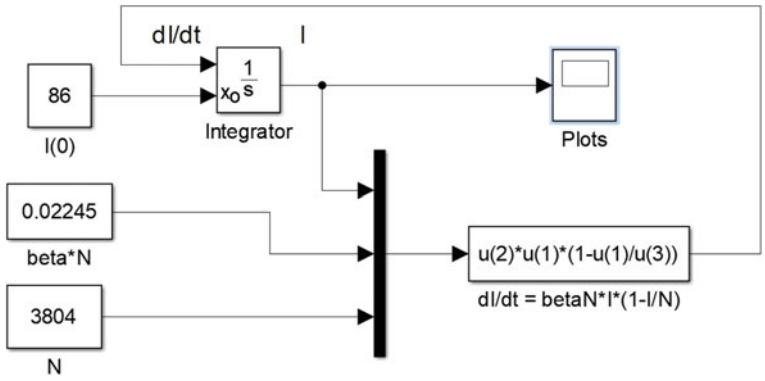


Fig. 6.40 Ebola SI model

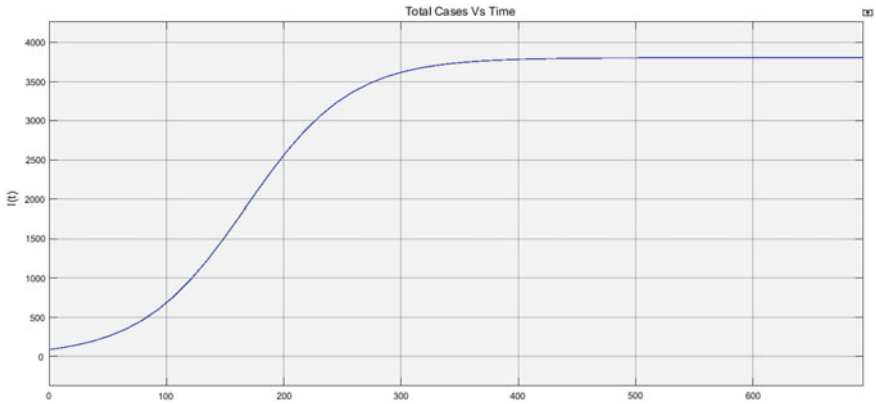


Fig. 6.41 Ebola SI model number of cases, $\beta N = 0.02245$

A MATLAB Simulink model can be created to verify that this SI model does resemble the total cases versus time plot. The model is run for the same number of days 694 that the data was available. The Simulink model and plot for the total cases versus time is shown in Figs. 6.40 and 6.41.

It is seen that the model reaches its carrying capacity at a faster rate than that of the actual data. This can be adjusted by reducing the variable βN by trial and error to find a best fit match. This is done and the new value for βN is 0.018. The new model for the total cases versus time is shown in Fig. 6.42.

Once the β coefficient is estimated, the SI model is expanded to include recovered and dead populations with the constraint that $S + I + R + D = N$:

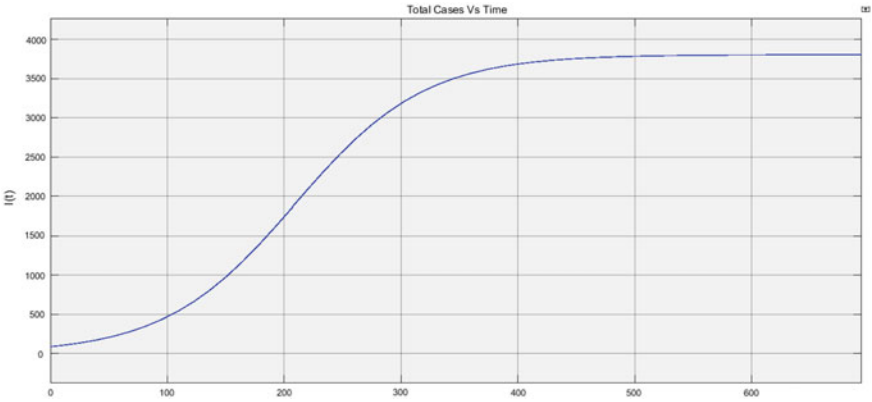


Fig. 6.42 Ebola SI model number of cases, $\beta N = 0.018$

$$\frac{\mathrm{d}R}{\mathrm{d}t} = \gamma I \quad \frac{\mathrm{d}D}{\mathrm{d}t} = \delta I$$

The death rate δ is found by taking two early points on the table that corresponds with the $D(t)$ curve (in Fig. 6.39). We may use the same dates as in the calculation of β.

$$\frac{\mathrm{d}D}{\mathrm{d}t} \approx \frac{70 - 66}{4} = 1 = \delta \cdot 103$$

An estimate for the death rate δ is therefore $\delta = 1/103 = 0.0097$. An estimate of the recovery rate is done based on the ratio of the total recovered and total death counts:

$$\frac{\gamma}{\delta} \approx \frac{3804 - 2536}{2536} \rightarrow \gamma \approx 0.00485$$

In order to complete the SIRD model, time delay effects must be added to account for the disease incubation period and for the recovery/death delay. Ebola is known to have an incubation period that ranges between 2 days and 21 days. From the time when symptoms show a recovery or death occur red between 6 and 16 days, a reasonable initial estimate of such delay effects may rely on midrange estimates.

$$T_{d,\text{incubation}} \approx 11 \quad T_{d,\text{recovery_death}} \approx 11 + 11 = 22 \text{ days}$$

The full SIRD model that includes time delay effects is

$$\begin{aligned}
\frac{\mathrm{d}S(t)}{\mathrm{d}t} &= -\beta I\left(t - T_{d,\text{incub}}\right)S\left(t - T_{d,\text{incub}}\right) \\
\frac{\mathrm{d}I(t)}{\mathrm{d}t} &= \beta I\left(t - T_{d,\text{incub}}\right)S\left(t - T_{d,\text{incub}}\right) - \gamma I\left(t - T_{d,\text{recov}}\right) - \delta I\left(t - T_{d,\text{recov}}\right) \\
\frac{\mathrm{d}R(t)}{\mathrm{d}t} &= \gamma I\left(t - T_{d,\text{recov}}\right) \\
\frac{\mathrm{d}D(t)}{\mathrm{d}t} &= \delta I\left(t - T_{d,\text{recov}}\right)
\end{aligned} \tag{6.27}$$

In a Simulink model of (6.27), we include integrators for just three of the four equations above, most commonly for $S(t)$, $I(t)$, and $D(t)$. $R(t)$ is then computed from the equality constraint as an output map: $R(t) = N - S(t) - I(t) - D(t)$.

In the Simulink model shown in Fig. 6.43, it is $S(t)$ that is computed from the constraint equation. Rather than creating two delayed versions of $I(t)$, one for the incubation delay and one for the recovery/death delay, the model shown in Fig. 6.43 delays the constant parameters β, γ, and δ. That is, the constant block value prior to the delay time is zero.

The simulation results of the model shown in Fig. 6.43 are given in Fig. 6.44.

The Simulink model of Fig. 6.43 must be modified prior to an implementation of any open-loop control strategy. That is, any vaccination strategy involves an

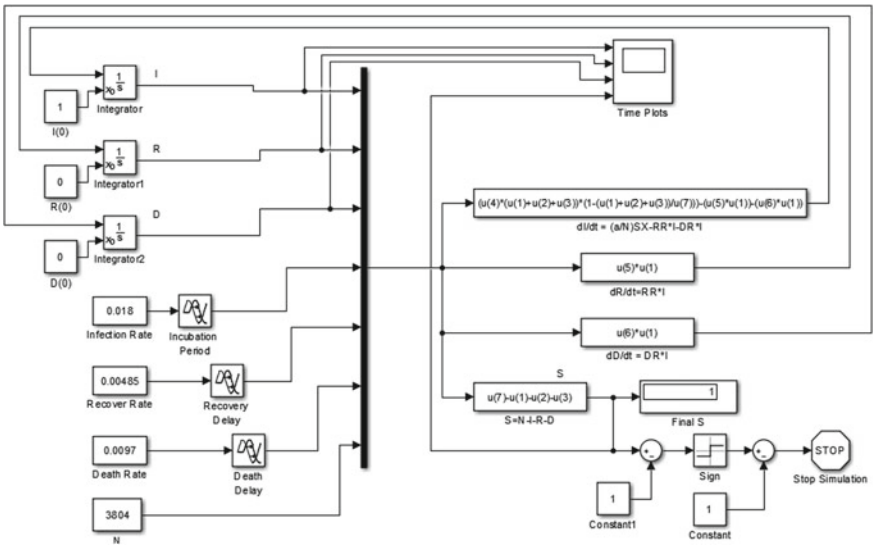


Fig. 6.43 Simulink diagram of the SIRD model with time delay effects

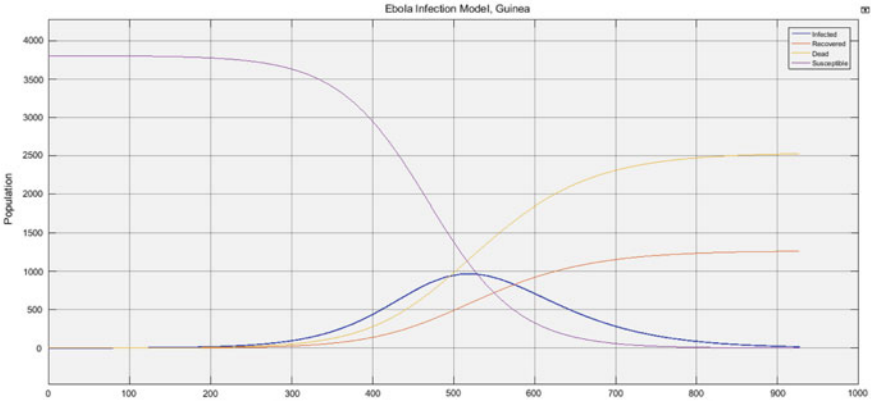


Fig. 6.44 Graphs of $S(t)$, $I(t)$, $R(t)$ and $D(t)$ that match the CDC data

introduction of a new population $V(t)$, the number of vaccinated individuals. The variable $V(t)$ decimates the number of susceptible individuals $S(t)$. Likewise, $\mathrm{d}V(t)/\mathrm{d}t$ decimates the right-hand side of the $\mathrm{d}S(t)/\mathrm{d}t$ equation. Therefore, in order to study vaccination effects and compare the effectiveness of various vaccination strategies, the variable $S(t)$ must be generated by an integrator and not as a mere output map.

Finally, let us discuss the role of N (the total population) in the above simulation studies and in potential follow-up studies. For a correct development of a vaccination strategy, one has to consider the country population N. In the simulation studies above, N was taken (due to the assistive SI model) to be the total number of Ebola cases. Obviously, the identification is not unique. There are infinitely many pairs of $\{N, \beta\}$ that produce the same set of curves as in Fig. 6.44. Replacement of N_{old} = Total number of Ebola cases, by N_{new} = The country population, necessitates the rescaling of β, in order to maintain a constant value of βN. Likewise, the parameters γ and δ have to be rescaled in order to maintain a constant value of the reproduction ratio R_0.

Another complication arises due to the relatively long period that the Ebola pandemic was active. Between 2014 and 2016, the total population N of the country of Guinea was not constant. It increased from approximately 11.8 M (in 2014) to 12.4 M (in 2016). In the simulation, one may replace the constant block of N by an input signal block created based on the above total population figures. As a result, the creation of $N = N(t)$ implies that β, γ, and δ all become time functions in order to maintain consistency.

All these follow-up studies are beyond the scope of this book.

6.3 Example of Control Systems with Simulator in the Loop: Glucose–Insulin Dynamics

The total execution time of a MATLAB script that controls multiple runs of a Simulink model may range from tens of milliseconds to several seconds, depending on the number of runs. It is still way shorter compared to the time constants of many real-life control processes. Multiple simulator runs may be motivated by both system identification and system control needs.

We studied earlier spread of infectious diseases models and vaccination strategies. A time constant for the spread of any disease is a number of days, and many (if not all) of the parameters of the model are unknown. Any collected data regarding the number of infected individuals and mortality allow for model parameters identification that results from multiple runs of the simulator. There is plenty of time to optimize the identification of the missing parameters, and consequently for using the identified model to explore possible control actions. This is what we call "simulator in the loop".

In this section, we demonstrate a simulator in the loop control strategy for a biomedical process involving the blood glucose regulation of diabetic patients.

A biomedical compartmental modeling is essentially a state-variable model for the concentrations of physiological substances (which may include drugs and metabolites) in one or more locations in the body. A compartmental model may describe what happens to substance A as it goes from one compartment (say the GI tract) to another compartment (say the blood system) and then to another compartment (a certain tissue). Another compartmental model may describe what happens to two substances A and B that may occupy the same vessels.

The "minimal model" for glucose and insulin dynamics consists of only two compartments: glucose in the blood and insulin in the blood. Let us discuss the glucose–insulin minimal model and the related simulation model as described in [7].

The system that regulates the level of blood glucose is a negative feedback control system: If the glucose level is elevated, insulin secretion in the pancreas is stimulated. As the blood insulin level increases, the uptake of glucose by certain insulin-sensitive tissues increases. This leads to a reduction of glucose blood concentration and consequently to less production of insulin.

Let us denote blood glucose concentration as x, and its unit is either mg/ml (as in Khoo's model) or mg/dl (as is more familiar to patients who self-administer blood glucose measurements. Blood insulin concentration is denoted as y, with some compatible units (mU/ml), without being more specific.

Glucose arrives at the blood system from multiple sources and under variety of conditions, ranging from ongoing processing of carbohydrates done by the gastrointestinal tract, through forming, storage, and processing of more complicated sugar molecules, as well as body fat, done by the liver, and by other body parts. In our simplistic dynamic model, we shall aggregate blood glucose intake as Q_L (mg/h). In many simulation scenarios, this intake will be considered constant.

A typical adult value of Q_L is Q_L = 8400 mg/h. Another glucose intake $U(t)$ is included in the model to account for a glucose test or other glucose control intakes.

In the blood system, glucose is either absorbed into insulin-sensitive tissues (such as certain muscles), or into other tissues that are not sensitive to insulin, or (if the level of glucose is very high, it gets excreted by the kidneys system). The rate of glucose absorption into tissues that are nonsensitive to insulin is λx (mg/h). That is, it is proportional to the blood glucose level x. A typical normal adult value for the coefficient λ is λ = 2470 ml/h.

The rate of glucose absorption into insulin-sensitive tissues is vxy. That is, it is proportional to the product of the concentrations x and y. A typical value of the coefficient v is v = 139,000 (ml/h)/(ml/mU).

The rate of glucose excretion by the kidneys is zero if $x \leq \theta$, where θ is an activation level of the kidneys. For $x > \theta$, the glucose loss is linearly proportional to the excess of x (above the activation threshold $\mu(x - \theta)$). A typical value for the threshold θ is θ = 2.5 mg/ml and for the coefficient μ is μ = 7200 ml/h.

Finally, a "glucose capacitance" C_G is included to scale correctly the time constant for glucose concentration changes due to abrupt intakes. For instance, a glucose test administered to a normal adult causes $x(t)$ to reach a peak after 30 min approximately, and thereafter it takes $4 \sim 5$ h for the glucose level to reduce back to normal. A typical value for C_G is C_G = 1/15,000 ml.

The rate of change equation for blood glucose is therefore

$$C_G \frac{\mathrm{d}x}{\mathrm{d}t} = \begin{cases} U(t) + Q_L - \lambda x - vxy & x \leq \theta \\ U(t) + Q_L - \lambda x - vxy - \mu(x - \theta) & x > \theta \end{cases} \tag{6.27'}$$

The rate of change equation for blood insulin includes a possible insulin intake $I(t)$, for instance as a control signal, an insulin generation term that is affected by the blood glucose level, and an ongoing insulin destruction at a rate proportional to the blood insulin level.

Insulin production features one more activation level: If $x \leq \varphi$ then no insulin is produced, and if x exceeds the activation level φ, then the production of insulin is linearly proportional to the excess of x above the threshold $\beta(x - \varphi)$. A typical insulin production threshold is φ = 0.51 mg/ml and a typical β coefficient value is β = 1430 (mU/h)(ml/mg). Insulin destruction is assumed to be αy, where typical coefficient value is α = 7600 ml/h. The insulin capacitance C_I is taken as C_I = 1/15,000 ml. The resulting insulin rate of change equation is

$$C_I \frac{\mathrm{d}y}{\mathrm{d}t} = \begin{cases} I(t) - \alpha y & x \leq \varphi \\ I(t) + \beta(x - \varphi) - \alpha y & x > \varphi \end{cases} \tag{6.28}$$

This second-order system is highly nonlinear due to the two conditional activation levels, the product of variables x and y related to glucose absorption by insulin-sensitive cells, and also due to the nonnegativity of the variables $x(t)$ and $y(t)$, as substance concentrations cannot be negative.

The given model coefficients represent a typical healthy adult. For $U(t) = 0$ and $I(t) = 0$, the model reaches an equilibrium state: $x_e = 0.81$ mg/ml (or the more familiar $x_e = 81$ mg/dl) and $y_e = 0.055$ mU/ml. In such an equilibrium, the glucose and insulin entering the blood system are balanced by the glucose and insulin that leave the blood system.

The same model, with slightly altered coefficients, is used to characterize diabetic patients as well. Type-1 diabetes results from lower than normal insulin production ability by the pancreas. That is, an insulin production coefficient β that can be as low as 0 and any value that is below the normal level of $\beta = 1430$ (mU/h)(ml/mg). Type-2 diabetes relates to a lower than normal glucose absorption ability by insulin-sensitive tissues. A continuum of disease severity can be modeled by taking the coefficient v to be anywhere between 0 and 139,000 (ml/h)(ml/mU).

Type-1 diabetes is frequent among young juveniles, whereas Type-2 diabetes is common among elderly people. Over time some Type-2 diabetes patients may develop a certain degree of Type-1 diabetes as well.

A Simulink model of (6.27′)–(6.28) is shown in Fig. 6.45 and is further explained below. Such a simulation model allows many questions to be asked and be readily answered:

(a) What is the equilibrium state (x_e, y_e) of a patient that has some given values of the coefficients β and v? One can show, as intuitively expected, that in Type-1 diabetes, the steady-state glucose level is elevated compared to normal and the

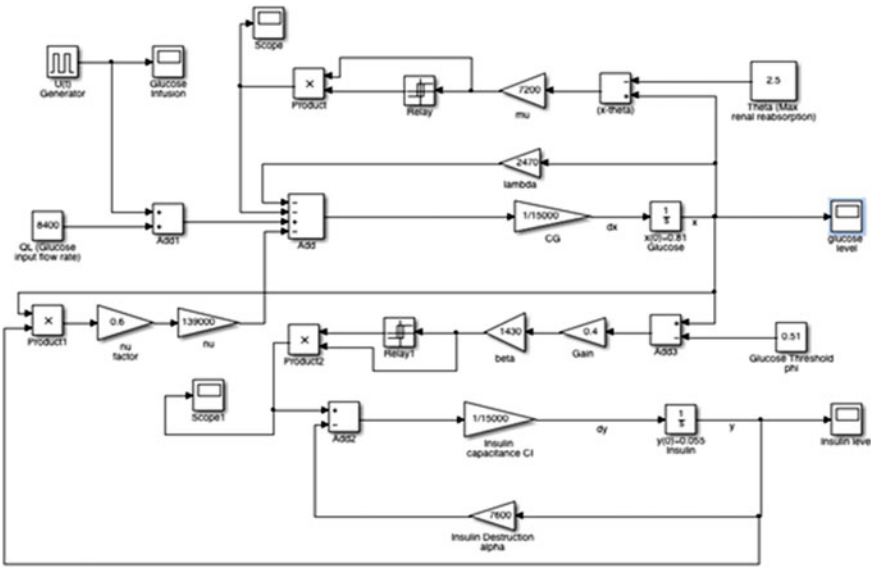


Fig. 6.45 Simulink model of the two-compartment glucose–insulin system

insulin level is lower than normal. In Type-2 diabetes, both steady-state glucose level and steady-state insulin level are elevated.

(b) A glucose test consists of a momentary consumption of glucose (represented by a narrow glucose input pulse $U(t)$) followed by a multi-hour observation of $x(t)$. How is the glucose test curve of a diabetic patient (with some disease severity coefficients) different than that of a healthy individual or from that of another diabetic patient?

(c) Interestingly, a real-time measurement of blood glucose $x(t)$ is readily available. On the other hand, the reading of blood insulin $y(t)$ takes a long time and is often skipped whenever medical decisions must be made quickly. How long it takes using $x(t)$ samples only to uniquely identify the correct coefficients of a diabetic patient?

(d) Once a patient's diabetes condition is fully identified, how should the treatment proceed? Let us break this problem into more manageable subproblems:

  (i) In a Type-1 diabetic patient, once the correct coefficient β is found, how should the open-loop control be designed? The control consists of a signal $I(t)$ consisting of a train of pulses. That is, the frequency and dosage of insulin injections must be decided in order to stabilize the patient's blood glucose level around the normal healthy level. Experimentation with the various pulses $I(t)$ may require multiple simulation runs. The simulations should decide what is the appropriate medications regime for the patient.

  (ii) The open-loop control of a Type-1 diabetic patient may be replaced with feedback control in which the signal $I(t)$ that now comes from an insulin pump is driven by the feedback signal $x(t)$.

  (iii) Similar open-loop control search strategy may be attempted for Type-2 diabetes patients once we can model the effect of certain medications. For instance, a common medication Metformin affects the value of Q_L in the model.

Some of the easier to handle questions are answered in the case study below (based on Claude Lieber MD and Ivan Bertaska, Spring 2014, Florida Atlantic University). However, prior to starting the case study, let us explain some of the features of the Simulink model in Fig. 6.45. Each of the model Eqs. (6.27) and (6.28) has conditional expressions. The implementation of such conditions is enabled by two-position relays that each has output levels of 0 (whenever the relay's input is negative, and that happens when $x(t)$ is below the respective activation level) and 1 (when the activation level is exceeded). The relay's binary output multiplies the conditional additive signal (the kidneys excretion on one branch of the model and the insulin production term on another branch), causing the term to pass or to be blocked. The gain blocks that pre-multiply the gains β and v provide the disease severity coefficients.

6.3.1 Case Study: System Identification and Open-Loop Control of Diabetic Patient's Blood Glucose Level

The case study below has severity factors of ($v = 0.6$ and $\beta = 0.4$).

In order to find the equilibrium state for this particular disease condition, the model is running with no $U(t)$ and $I(t)$ inputs, using the normal healthy initial state of ($x_e = 0.81$, $y_e = 0.055$). The new steady-state conditions, as shown in Fig. 6.46, become thereafter the initial conditions for this study's specific patient.

The next set of figures compares the glucose test results of healthy and diabetic individuals.

See Figs. 6.47 and 6.48. The glucose tolerance test consists of 100 mg of glucose administered over 15 min. The experiment starts 30 min after the start of the simulation run. Figure 6.47 presents the glucose test results for a healthy individual. The red curve is $x(t)$ and the blue curve is $y(t)$. One time unit equals 1 h.

Figure 6.48 shows the abnormal glucose test results of the patient discussed in this case study, using the same glucose input $U(t)$. Interestingly, the manner by which $U(t)$ is connected to the model (and consequently the almost immediate glucose level rise) indicates an intravenous glucose administration. In an oral glucose test, it takes longer for the glucose to be absorbed from the gut. For this patient, the glucose level reaches a dangerous peak, and so is the insulin level. Both take longer to clear leading to an elevated steady state. If the patient has unknown diabetes conditions, the first few hours of the test clearly indicate Type-2 diabetes conditions, however, this test alone is insufficient in order to quantitatively diagnose the patient's disease severity. The glucose test results are compared against those coming from simulation models of a healthy and various diseased individuals, this can complete the identification of the system that needs to be controlled.

Let us assume next that a patient, suspected to be diabetic, is to be diagnosed (or in control systems language—be system identified) in real time, and once identified, the patient is to be stabilized by medications. For the identification, we set two parallel Simulink models (subsystems) as shown in Figs. 6.49 and 6.50.

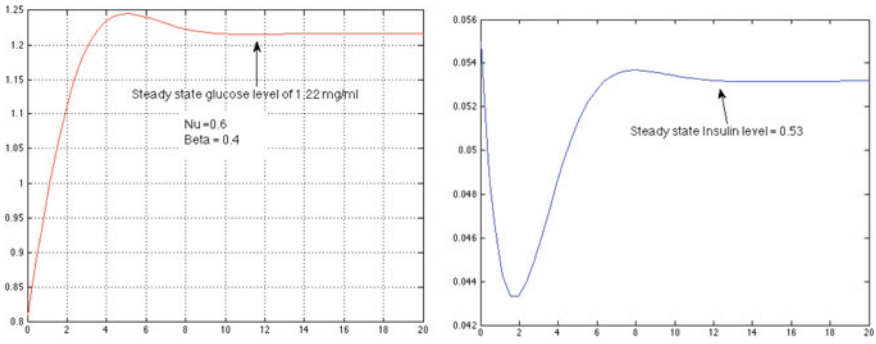


Fig. 6.46 Simulation run using arbitrary initial state in order to determine the correct initial state

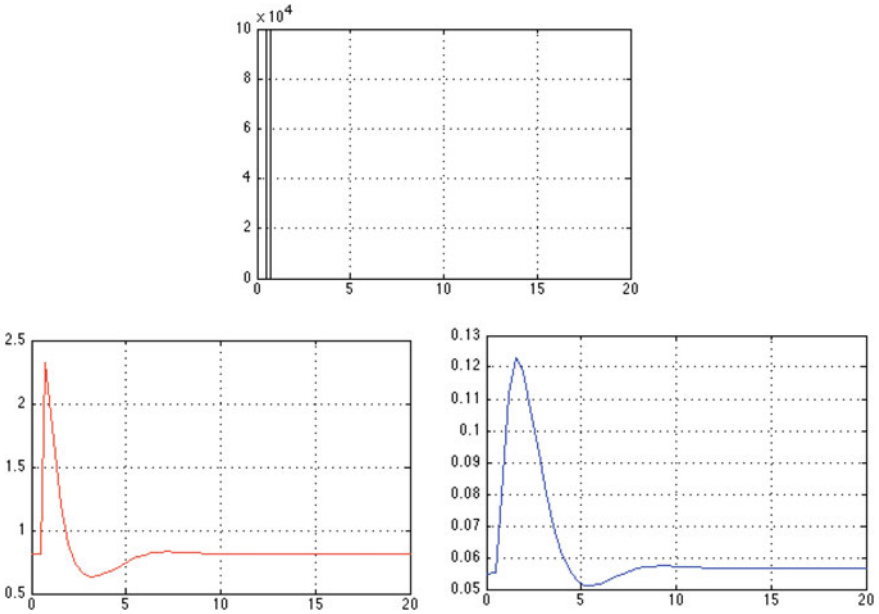


Fig. 6.47 Glucose test of a healthy individual

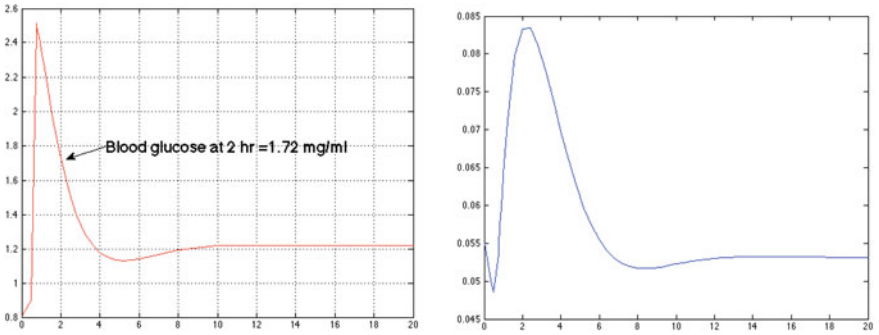


Fig. 6.48 Glucose test results of a diabetic patient with $v = 0.6$ and $\beta = 0.4$

The model above, as a subsystem called patient—froze the parameters $v = 0.6$ and $\beta = 0.4$, was compared to a facsimile wherein both parameters could be varied from 0.1 to 1. That is, in the case of β—its gain block (showing the normal β value) is pre-multiplied by another gain that runs between 0 and 1, implemented in terms of a combination of a multiplier and a constant block. The constant block can be assigned an algebraic value as a global parameter in a MATLAB script that calls that Simulink model. Likewise for the v parameter.

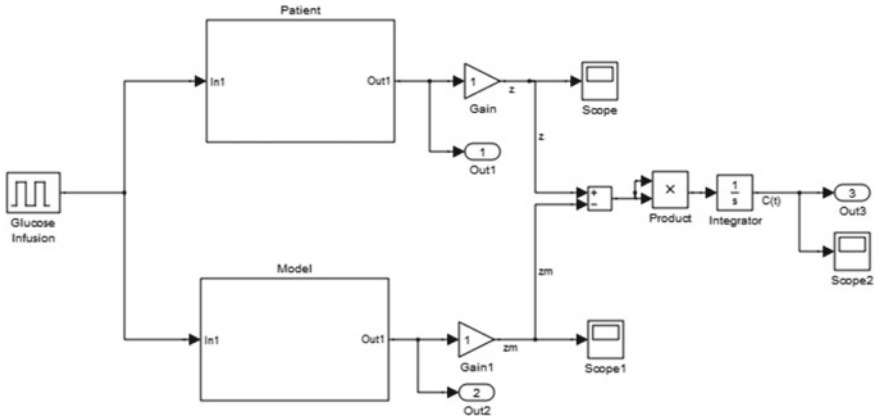


Fig. 6.49 Variable model output compared against patient's data

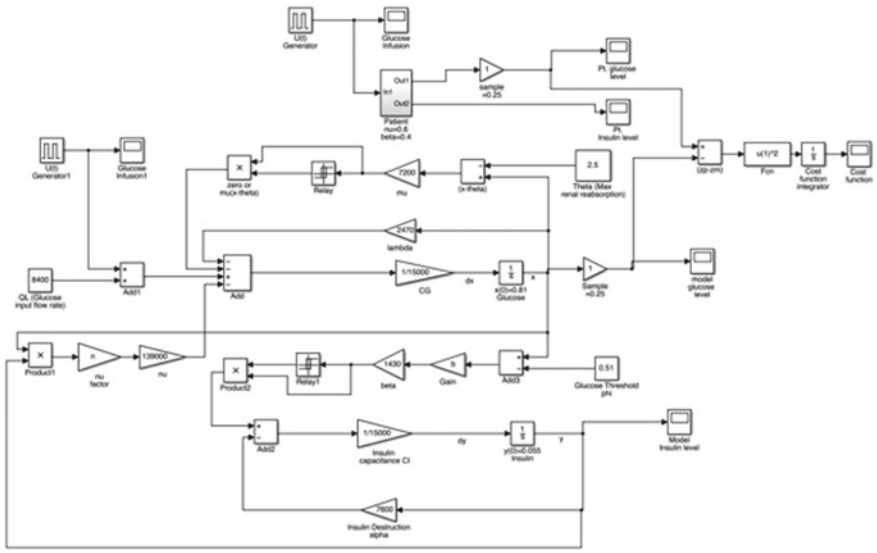


Fig. 6.50 Modifiable model compared against a patient subsystem model

The idea is that given a glucose tolerance test, a cost function can be developed and calculated so that the minimum of the cost function will reflect the v and β parameters of the patient and say whether the patient has type I or type II diabetes (assuming that this is unknown), without measuring the concomitant insulin levels. Only glucose levels are to be monitored every $T = 15$ min. The cost function takes the square of the difference between the patient's glucose $x(t)$ and the simultaneous model glucose $x_m(t)$ and integrates that function over the time of the trial.

$$C(t) = \int_0^t \left[x^2(\tau) - x_m(\tau)\right] \mathrm{d}\tau \approx \sum_{i=1}^{N} \left[x^2(iT) - x_m^2(iT)\right]$$

The comparison is carried for a duration of t (h) with the variables $x(t)$ sampled every $T = 0.25$ h. The cost function is tabulated against the values of the searched parameters. The results for $t = 2$ h based on 8 glucose readings clearly picked up the true patient's condition, as shown in Table 6.3.

Unique and correct identification is obtained with as few as 3 glucose measurements, as shown in Table 6.4.

Running the comparison with only two glucose readings produces multiple table cells with minimal value of the cost function, one of which is the correct set of patient severity coefficients.

Let us now demonstrate several open-loop control strategies to medicate a patient who is correctly identified and now has known β and v coefficients.

The Simulink model shown in Fig. 6.51 includes two means of control: (a) A hypothetical drug that can enhance the insulin-assisted glucose absorption by gradually increasing the v coefficient. The drug acts like a first-order linear model with a time constant of approximately 30 min, (b) Insulin injections. These are effective instantly and bring the patient's insulin levels up.

For each medication, the control input signal consists of a train of pulses. The pulse width is the drug intake time. The total medication ingested in one pulse is the area under the pulse (its amplitude times the pulse width).

We first study the effect of the first medication alone (without adding insulin). The design parameters are the dosage and frequency of the medication. The first result is for an amplitude of 10^6 and pulse width of 15 min shown in Fig. 6.52. Due to the time constant of the drug absorption, the effects of the medication are delayed.

The response shown in Fig. 6.52 shows oscillations near the desired steady-state value of 0.81 mg/ml with a settling time of nearly 5 h after the starting of the glucose tolerance test. The response shown in Fig. 6.53 involves a doubling of the medication frequency (that is, pill taken every 30 min) but with a reduction of pulse amplitude to 60% of that administered in the previous simulated trial. The settling time is reduced to four and a half hours and there is a constant steady-state value of 0.81 mg/ml.

Another medicating strategy would be to administer insulin in order to compensate for the reduced β level. The control strategy is first explored without adding to it the v-enhancing medication. The insulin pulses were taken every hour at a dose of 3×10^4 and a pulse width of 1 min (representing quick injections). The results shown in Fig. 6.54 demonstrate a settling time of around 3.5 h, an improvement compared to the other medication. As the frequency of insulin administration is doubled, no further improvement in settling time is obtained. The response is worse as the peak of the glucose level increases (Fig. 6.55).

Table 6.3 Uniquely identified parameters after eight glucose measurements

Nu	Beta									
	0.1000	0.2000	0.3000	0.4000	0.5000	0.6000	0.7000	0.8000	0.9000	1.0000
0.100	0.3529	0.3347	0.3173	0.3006	0.2845	0.2691	0.2544	0.2401	0.2262	0.2128
0.200	0.2681	0.2390	0.2117	0.1859	0.1618	0.1397	0.1196	0.1015	0.0852	0.0708
0.300	0.1978	0.1605	0.1274	0.0990	0.0749	0.0550	0.0389	0.0262	0.0168	0.0102
0.400	0.1362	0.0966	0.0652	0.0411	0.0235	0.0115	0.0046	0.0021	0.0035	0.0082
0.500	0.0857	0.0500	0.0251	0.0096	0.0019	0.0009	0.0055	0.0150	0.0286	0.0456
0.600	0.0480	0.0198	0.0046	**0.0000**	0.0041	0.0152	0.0323	0.0540	0.0797	0.1085
0.700	0.0226	0.0045	0.0007	0.0083	0.0247	0.0483	0.0775	0.1110	0.1480	0.1875
0.800	0.0086	0.0021	0.0106	0.0306	0.0595	0.0950	0.1356	0.1800	0.2272	0.2763
0.900	0.0048	0.0106	0.0317	0.0641	0.1048	0.1516	0.2028	0.2570	0.3132	0.3706
1.000	0.0098	0.0282	0.0618	0.1062	0.1581	0.2154	0.2760	0.3389	0.4030	0.4675

Table 6.4 Identified parameters after $N = 3$ measurements

0.00	0.10	0.20	0.30	0.40	0.50	0.60	0.70	0.80	0.90	1.00
0.10	6.2599 E−04	6.2410 E−04	6.2222 E−04	6.2033 E−04	6.1845 E−04	6.1658 E−04	6.1471 E−04	6.1284 E−04	6.1097 E−04	6.0911 E−04
0.20	4.0083 E−04	3.9788 E−04	3.9494 E−04	3.9201 E−04	3.8909 E−04	3.8619 E−04	3.8329 E−04	3.8041 E−04	3.7754 E−04	3.7468 E−04
0.30	2.2743 E−04	2.2417 E−04	2.2094 E−04	2.1773 E−04	2.1455 E−04	2.1139 E−04	2.0826 E−04	2.0515 E−04	2.0207 E−04	1.9901 E−04
0.40	1.0402 E−04	1.0116 E−04	9.8335 E−05	9.5556 E−05	9.2819 E−05	9.0122 E−05	8.7468 E−05	8.4854 E−05	8.2282 E−05	7.9751 E−05
0.50	2.8889 E−05	2.7062 E−05	2.5296 E−05	2.3590 E−05	2.1946 E−05	2.0362 E−05	1.8839 E−05	1.7376 E−05	1.5973 E−05	1.4630 E−05
0.60	3.6919 E−07	1.6393 E−07	4.0944 E−08	**0.0000 E+00**	4.0868 E−08	1.6332 E−07	3.6712 E−07	6.5206 E−07	1.0179 E−06	1.4644 E−06
0.70	1.6838 E−05	1.8790 E−05	2.0846 E−05	2.3007 E−05	2.5271 E−05	2.7640 E−05	3.0111 E−05	3.2686 E−05	3.5363 E−05	3.8143 E−05
0.80	7.6719 E−05	8.1314 E−05	8.6036 E−05	9.0885 E−05	9.5862 E−05	1.0096 E−04	1.0619 E−04	1.1155 E−04	1.1702 E−04	1.2263 E−04
0.90	1.7848 E−04	1.8616 E−04	1.9399 E−04	2.0196 E−04	2.1009 E−04	2.1837 E−04	2.2679 E−04	2.3537 E−04	2.4409 E−04	2.5295 E−04
1.00	3.2063 E−04	3.3179 E−04	3.4312 E−04	3.5462 E−04	3.6630 E−04	3.7814 E−04	3.9016 E−04	4.0234 E−04	4.1470 E−04	4.2722 E−04

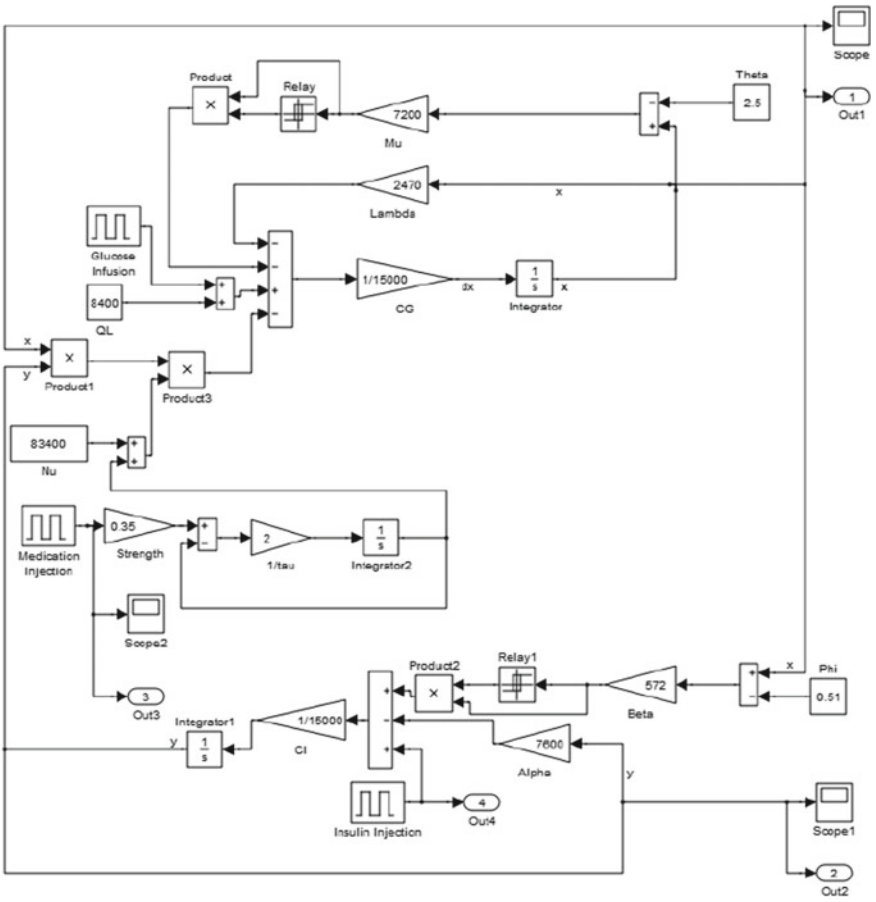


Fig. 6.51 Open-loop control model of an identified patient

Mixing of the two medications is the next exploration phase. The result shown in Fig. 6.56 was obtained by taking both medications every hour. The v-enhancement medication is taken at a strength of 0.35 and the insulin amplitude is 7600. Obviously, the response is unsatisfactory due to the dangerous overshoot that pushed the glucose level to hypogycemic levels (that is, a level of $x(t)$ that falls below 70 mg/dl). No further exploration was done because after all, there is no medication that directly enhances glucose absorption into insulin-sensitive cells. That v-enhancement medication was totally hypothetical.

This leads to the final question of this case study: How do we model the effect of real medications given to Type-2 diabetic patients?

Stabilizing the blood glucose levels in a patient like the one featured in this case study ($v = 0.6$ and $\beta = 0.4$) given is most commonly done with oral medications (for type-2 diabetes). Insulin can also be used. Although insulin resistance is the hallmark

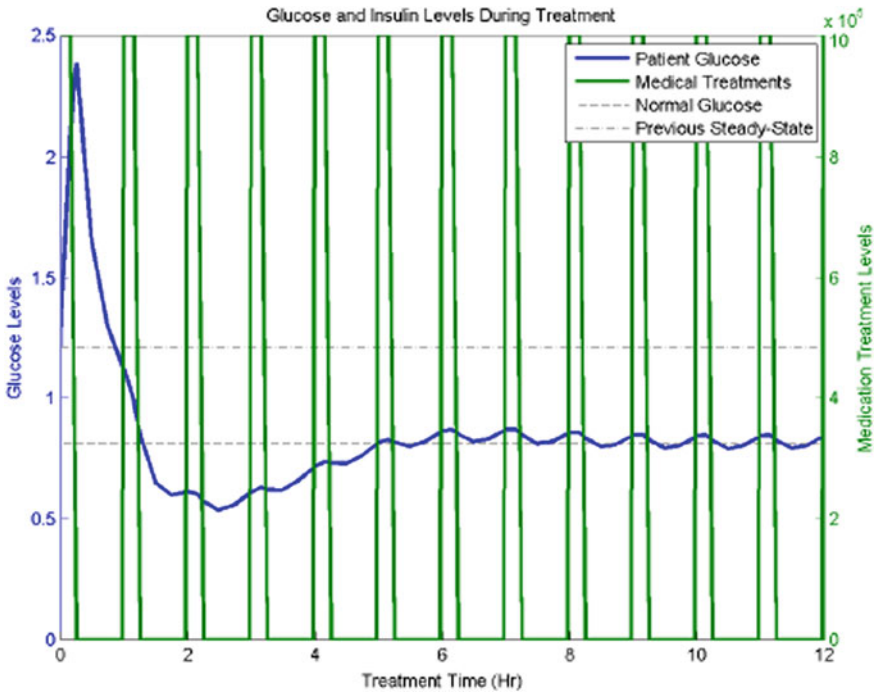


Fig. 6.52 Glucose level stabilization with hourly v-enhancement medications

of type-2 diabetes, the resistance is almost always not absolute and administration of extra insulin can stabilize these patients very nicely. It does represent an added burden to the patient who has to inject himself rather than take pills. All diabetics should follow a diet balanced in carbohydrates. The two medications most commonly used are Metformin and/or a Sulfonourea (glyburide or glipizide). Many Type-2 diabetes patients are stabilized by Metformin alone. The Metformin works by reducing the production of glucose by the liver. The effect of Metformin is to reduce the effective Q_L in our model. The Sulfonourea drugs work essentially as insulin augmentation with the onset of action that is around half an hour and the peak is at $2 \sim 3$ h with a meaningful duration of action of about $6 \sim 8$ h.

Metformin is rapidly absorbed from the gut over about half an hour. The kidney excretes it with a half-life of $4 \sim 5$ h (exponential decay). The literature about how much the drug reduces the hepatic production of glucose puts values of $40 \sim 50\%$. Let us say that at peak blood levels, the medication reduces Q_L by 50% and proportionately less at lower levels. Since it does not increase the insulin secretion, hypoglycemia does not occur with this drug (a major advantage over the Sulfonourea drugs). Let us represent the administration of Metformin as having rapid absorption (peak at time zero). This is not correct pharmacokinetics but Metformin is absorbed rapidly. The excretion of Metformin by the kidneys, at the above half-life, is completed in about $10 \sim 14$ h. This is approximated by

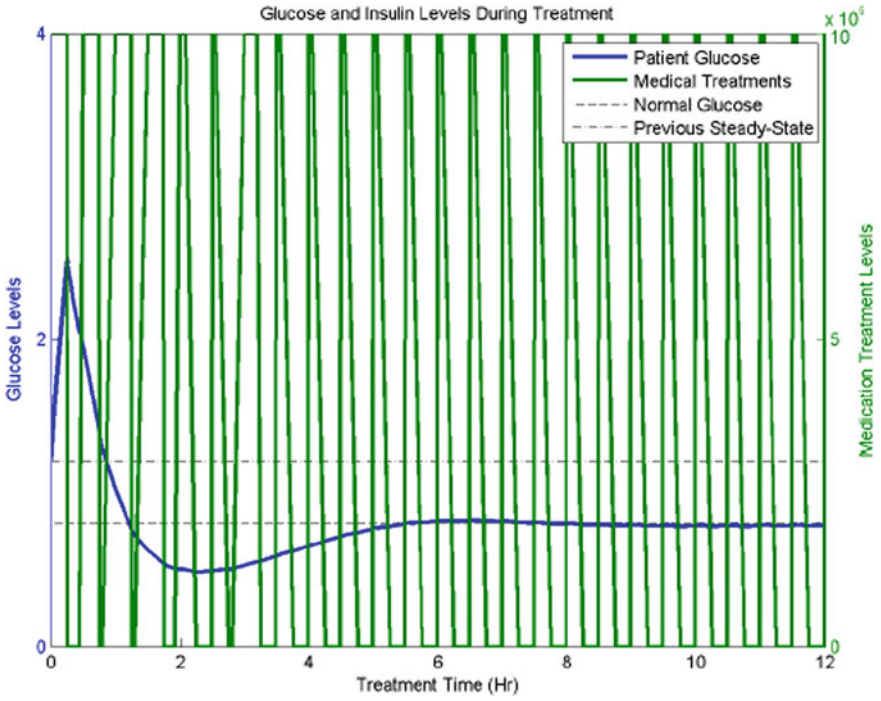


Fig. 6.53 Glucose-level stabilization with more frequent and reduced strength medication

subtracting $0.5\exp(-0.15t)Q_L$ from the Q_L input as shown in the little subsystem in Fig. 6.57 (inserted into the model as in Fig. 6.58). A single dose of Metformin creates a new glucose inflow that starts at 4200 and grows exponentially to 8400 over about 20 h. Metformin is given twice daily (every 12 h) and it typically does not bring the patient's blood glucose $x(t)$ to 0.8, but rather to values such as $1.1 \sim 1.2$ mg/ml as shown in Fig. 6.59.

As I stated above, this had the effect of reducing the Q_L which slowly came back to basal levels over around 12 h (time for the next dose).

One may argue, based on Fig. 6.59, that the case study patient has actually a mild case of the disease, and may be nothing is necessary except for watchful waiting to see if diabetes gets worse over time.

Next, the administration of Sulfonourea is modeled (Fig. 6.60). Severely hyperglycemic patients typically respond to glyburide with a $70 \sim 80\%$ increase in insulin levels. The effect is less dramatic in the patient with more normal glucose levels like our patient. Here, we can assume a $40 \sim 50\%$ increase in insulin levels.

The final set of results compares glucose test results for the patient treated with Metformin, the patient treated with Sulfonourea, and the patient treated with both medications (Fig. 6.61). A clear conclusion is that for such a mildly sick patient, any treatment with insulin (or Sulfonourea) is unnecessary and creates the

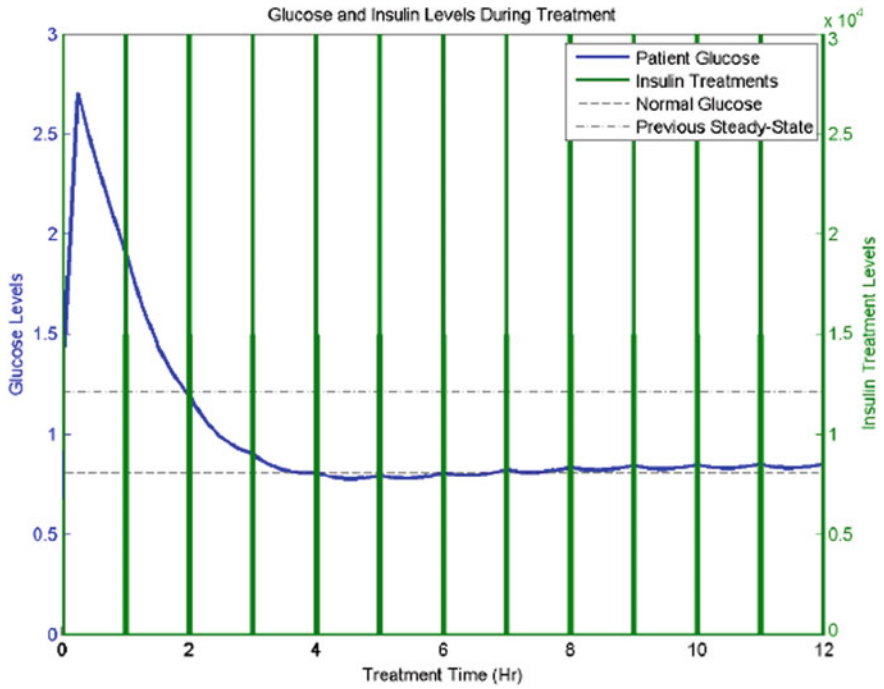


Fig. 6.54 Patient stabilization by means of insulin injections only

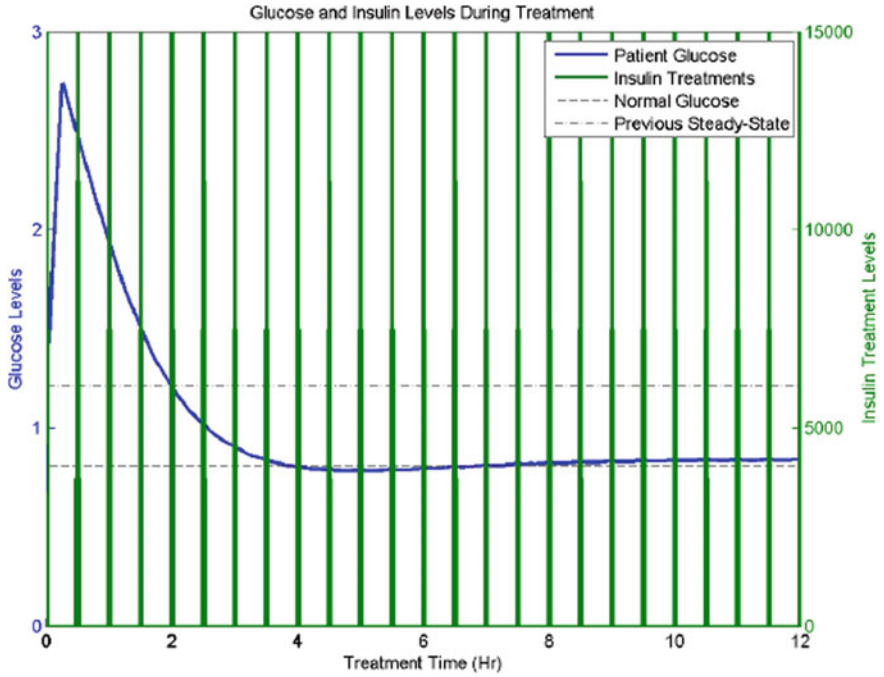


Fig. 6.55 Patient stabilization by double frequency of insulin injections

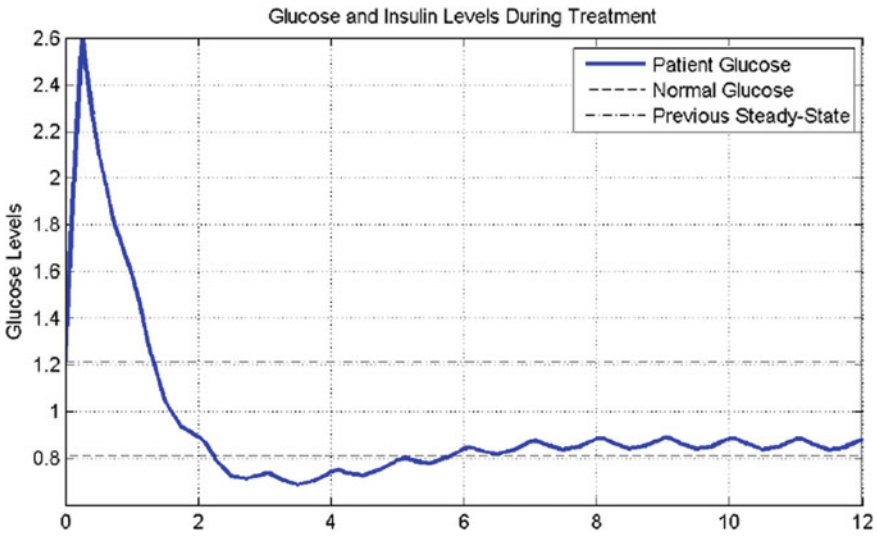


Fig. 6.56 Mixing the two medications may lead to a dangerous hypoglycemia

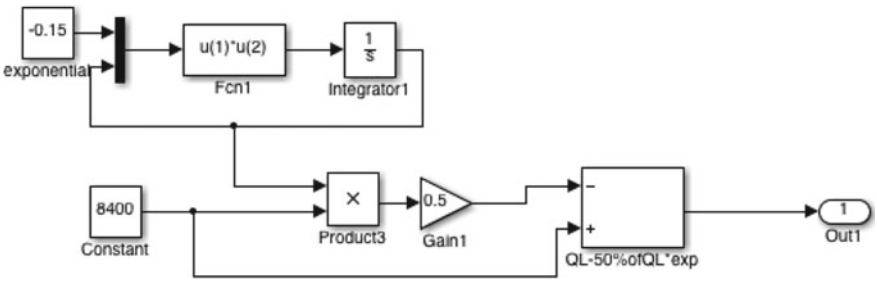


Fig. 6.57 Model that shows how Metformin intake affects the system input Q_L

unnecessary risk of hypoglycemia. Mild cases of type-2 diabetes are better treated with Metformin.

Many more exploration issues are beyond the scope of this book. These include:

(1) Simulation-based development of closed-loop control strategies for stabilizing the glucose level
(2) Comparison of the above two-compartment model with a three-compartment model (one that may involve blood glucagon, a hormone that elevates blood glucose levels).

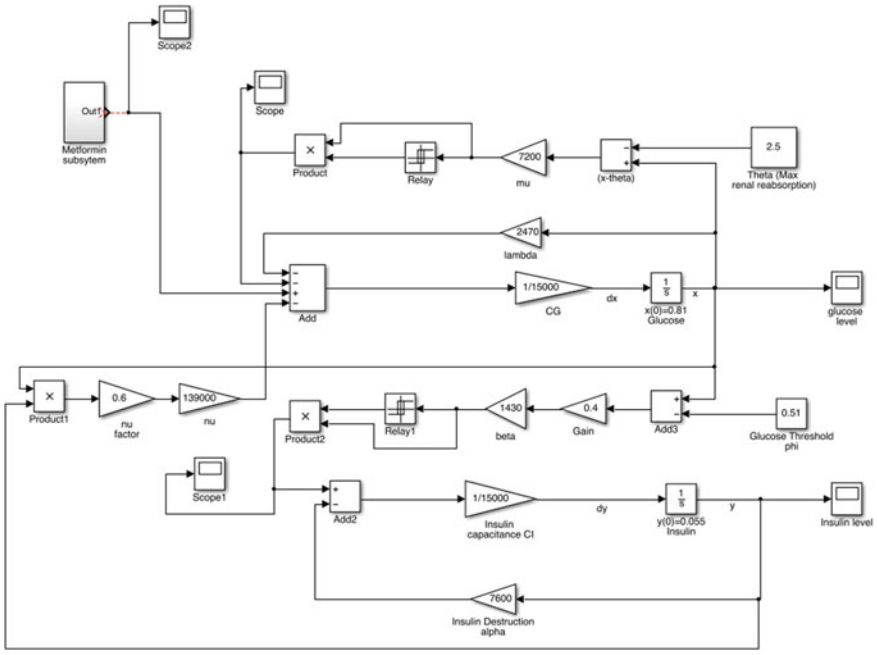


Fig. 6.58 Glucose-Insulin model with Metformin input subsystem added

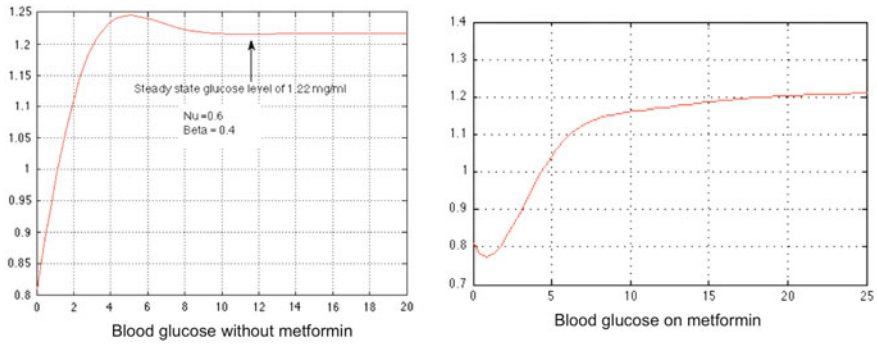


Fig. 6.59 Comparison of blood glucose with and without Metformin medication

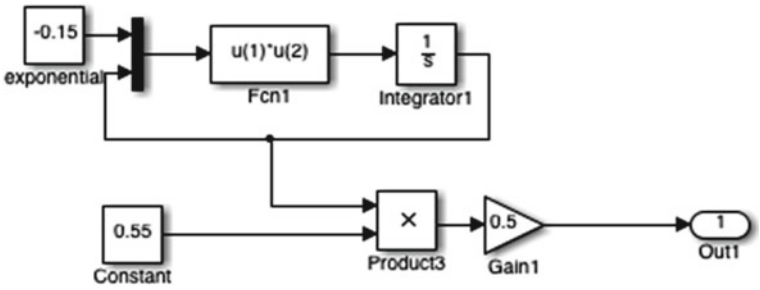


Fig. 6.60 Modeling of the administration of Sulfonourea as part of the insulin input

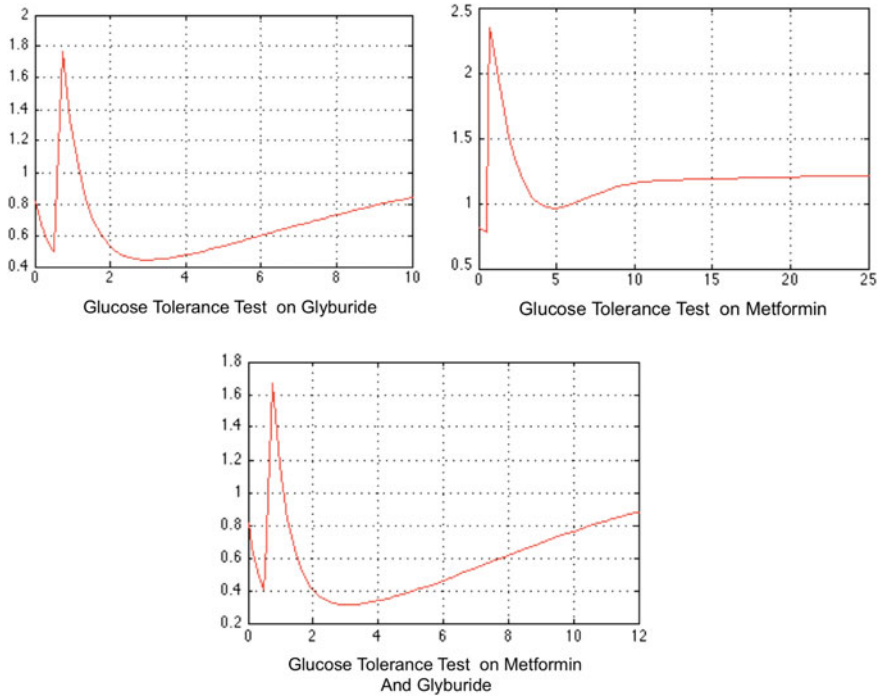


Fig. 6.61 Glucose test comparison. Treatment with insulin (or insulin enhancement) may create too drastic reduction in the glucose level

6.4 Simple Relay-Based Control Systems

Relays are two-position or three-position devices. Considered efficient power amplifiers, capable of transferring large actuation signals based on low-power command signals, such devices often form the actuation part of feedback control systems.

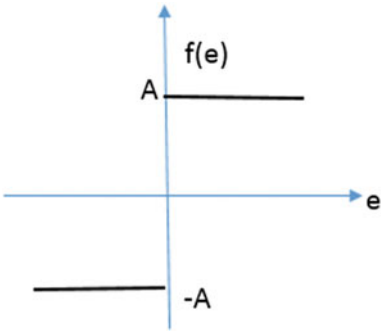


Fig. 6.62 Two-position relay input–output characteristics

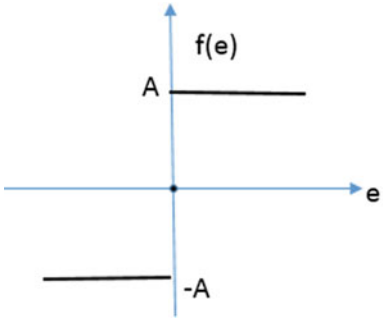


Fig. 6.63 Input–output characteristics of a three-position relay

In the following symmetric two-position relay shown in Fig. 6.62, the output signal is constant and positive if the input's sign is positive, and symmetrically negative if the input $e(t)$ is negative, where $e(t)$ could be a certain control error signal.

A three-position relay has a third output level of zero if the input is exactly zero (Fig. 6.63).

In the early days of feedback control, many practical relays suffered from nonidealities, such as dead zone (a finite activation level) and hysteresis, as in the examples shown in Fig. 6.64.

Dead zone in a feedback loop that causes a finite steady-state error. Hysteresis (switching upwards and downwards at non-equal input values) in a feedback loop has a destabilizing effect. Both effects may sometime be intentional as a way of enhancing the system's noise immunity. For example, most air conditioning units feature activation levels that have hysteresis in order to avoid undesired high-frequency on–off switching due to noisy readings by the temperature sensors.

In modern relay control systems, the relay may be implemented in software being part of the system's digital control system. Very often, the use of say a three-position relay is conceptual in order to better define the control strategy, whereas at the implementation phase, the relay is replaced by a high-gain saturation amplifier.

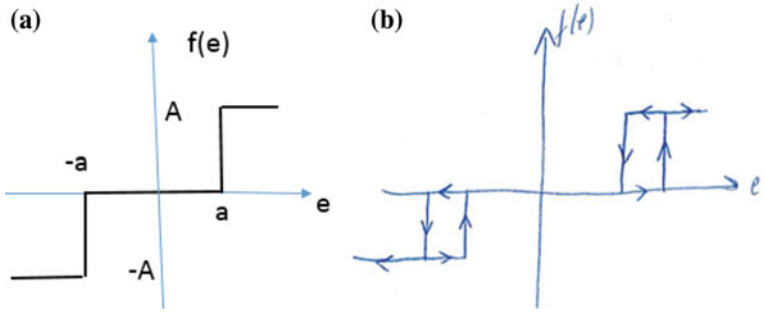


Fig. 6.64 Three-position relay with a dead zone (**a**) and three-position relay with a dead zone and hysteresis (**b**)

This section introduces several control concepts such as "sliding mode" and "bang–bang" by means of simple second-order feedback systems.

6.4.1 How Not to Do Relay Control

Let us consider an example of a feedback control loop, with unity negative feedback, driven by a step commands $r(t)$ of size A. The controlled process is a linear second-order type-1 system $G(s) = Y(s)/U(s)$ with no zeros, where $y(t)$ is the process output and $u(t)$ is the control signal. Let us assume that a three-position relay acts as a controller and actuator, accepting the error signal $e(t) = r(t) - y(t)$ as its input and generating the control signal $u(t)$, which can be at one of three possible levels: A, $-A$ and 0 (Fig. 6.65).

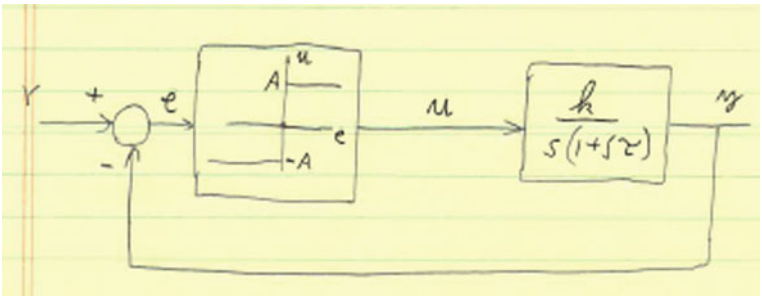


Fig. 6.65 Relay control second-order system where the relay is driven directly by the error signal

$$\frac{Y(s)}{U(s)} = \frac{k}{s(1+s\tau)} \quad u(t) = \begin{cases} A & e > 0 \\ 0 & e = 0 \\ -A & e < 0 \end{cases} \quad e(t) = r(t) - y(t) \tag{6.29}$$

The system may be analyzed in the phase plane plotting dy/dt against y (as done in the simulations), or de/dt against e as is often done in mathematical analysis. In the latter, the size of a step input $r(t)$ becomes part of the initial conditions of $e(0)$. For type-1 systems (like the process $G(s)$ of this example), one may even analyze plane the effects of ramp inputs.

The three parts of (6.29) may be combined into a single time-domain piecewise linear model:

$$-\tau\ddot{e} - \dot{e} = \begin{cases} kA & e > 0 \\ 0 & e = 0 \\ -kA & e < 0 \end{cases} \tag{6.30}$$

The phase plane $(x_1, x_2) = (e, \mathrm{d}e/\mathrm{d}t)$ is divided by the model of (6.30) into three parts. The model valid in the half plane $x_1 > 0$, for instance, is

$$\dot{x}_1 = x_2 \quad \dot{x}_2 = -\frac{1}{\tau}x_2 - \frac{kA}{\tau} \quad \text{if } x_1 > 0 \tag{6.31}$$

Trajectories in the phase plane represent solutions of (6.31) for various initial states $(e(0), \mathrm{d}e/\mathrm{d}t(0))$. At any arbitrary point (x_1, x_2) in the half plane $x_1 > 0$, the slope m of the trajectory that passes through such a point is

$$\frac{\mathrm{d}x_2}{\mathrm{d}x_1} = -\frac{1}{\tau} - \frac{kA}{\tau x_2} = m \tag{6.32}$$

We see that $m = \infty$ for $x_2 = 0$ so the trajectories intersect the x_1 axis vertically. For large value of $|x_2|$, the value of m approaches $-1/\tau$. As x_2 decreases, the slope becomes larger. For $x_2 = -\mathrm{kA}$, the slope $m = 0$. All this easy to obtain information allows us to sketch manually solution trajectories in the half plane $x_1 > 0$ (Fig. 6.66). The trajectories in the half plane $x_1 < 0$ are obtained in a similar manner.

For any step input and any set of initial conditions, the control system, as the output $y(t)$ converges slowly to the value of the reference input, switches infinitely many times between one relay level to its opposite. This is confirmed by the Simulink model in Fig. 6.67 and its outputs shown in Figs. 6.68 and 6.69.

The example above is representative of all control systems that utilize an oversimplistic control of relays. The switching curve of the relay was the line $e = 0$ in the de/dt versus e phase plane. Relays can be utilized more effectively but, as it will be seen, it requires a more careful programming of the input signal to the relay.

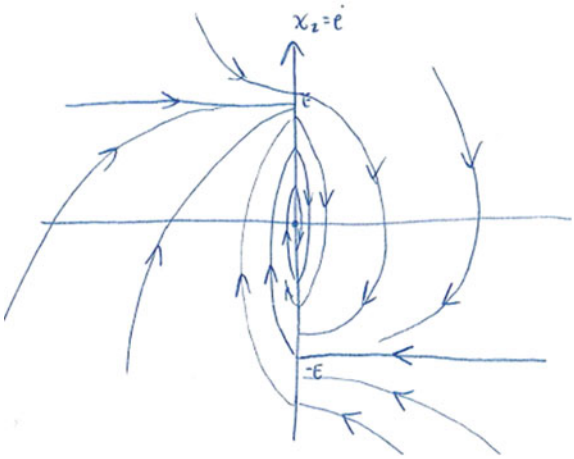


Fig. 6.66 Sketches of solution trajectories in the phase plane of a linear second-order process with relay control based on the error signal

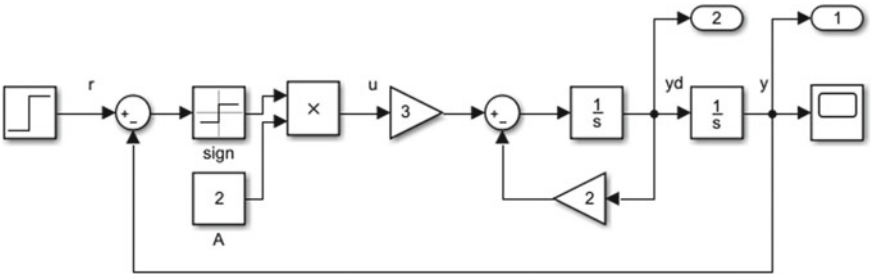


Fig. 6.67 Simulink model of type-1 linear process driven by a relay that is driven by the error signal in a tracking control system, for $k = 1.5$, $A = 2$, and $\tau = 0.5$

6.4.2 *Relay Control with Position and Velocity Feedback—The Sliding Phenomenon*

Example 6.4.1: Relay Control of a Second-Order Type-1 Linear Process with a Position and Velocity Feedback Control of the Relay Switching Line

The first logical attempt to extend the relay driving method, based on the error signal, is to tilt the switching line. In other words, the switching of the relay is to be done based on a combination of $e(t)$ and $de/dt(t)$, as illustrated in Fig. 6.70 using a PD controller on the error signal $e(t)$, or equivalently—combine the error signal with a scaled version of the error velocity and use this signal to drive the relay.

In order to understand the potential advantages of this modified control strategy, let us observe (6.33), which is the modified Eq. (6.30):

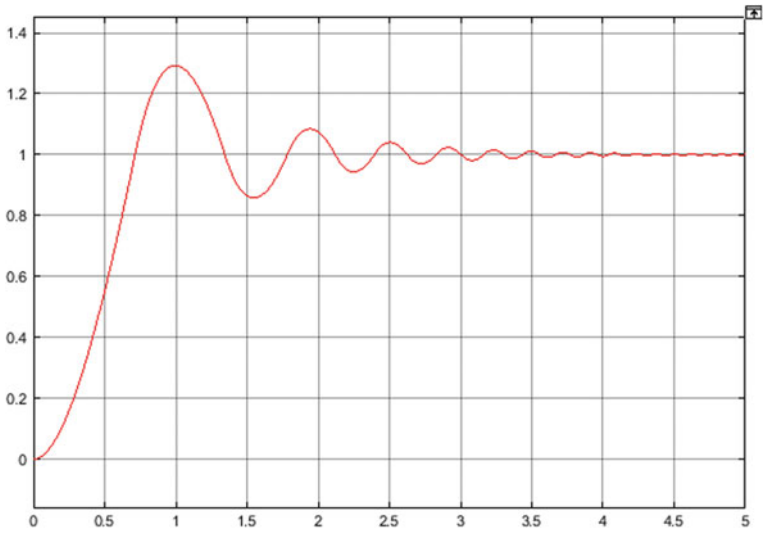


Fig. 6.68 Relay control that is acting directly on the error signal leads to a very oscillatory response

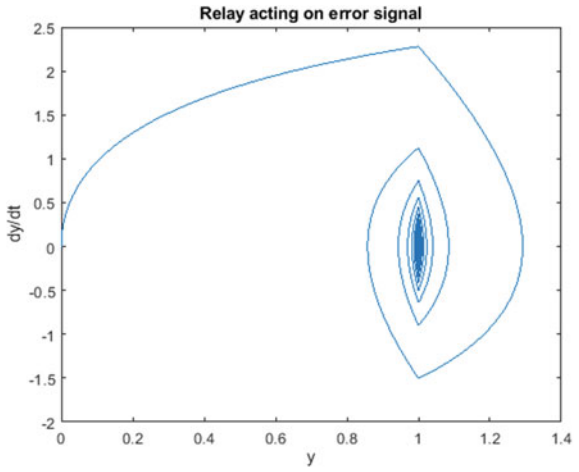


Fig. 6.69 An equivalent phase plane solution (y vs. dy/dt) to that shown in Fig. 6.66

$$-\tau_m \ddot{e} - \dot{e} = \begin{cases} K_m A & e + T_L \dot{e} > 0 \\ 0 & e + T_L \dot{e} = 0 \\ -K_m A & e + T_L \dot{e} < 0 \end{cases} \tag{6.33}$$

Again the phase plane (e, de/dt) is divided into three parts—the half plane that lies to the right of the switching line $e + T_L \dot{e} = 0$, the half plane that lies to the left of the switching line and the switching line itself. The model equations do not

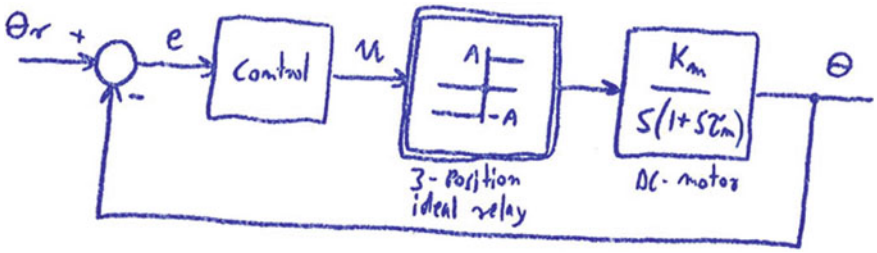


Fig. 6.70 A type-1 second-order linear process (such as a DC motor in position control) driven by a three-position relay actuator driven by a PD controller $u(t) = e(t) + T_L de(t)/dt$

change the switching rule of the relay. The new version of (6.31) (that represented earlier the state equations that are valid at $x_1 > 0$) is:

$$\dot{x}_1 = x_2 \quad \dot{x}_2 = -\frac{1}{\tau_m}x_2 - \frac{K_m A}{\tau_m} \quad \text{if } x_1 + T_L x_2 > 0 \tag{6.34}$$

The system trajectories sketched in Fig. 6.66 are exactly the same trajectories sketched in Fig. 6.71, except that they are intercepted by the switching line earlier than the line $x_1 = 0$.

As a side comment let us observe that for the positive velocity feedback $T_L < 0$, the switching line tilts to the right which causes the trajectories to be intercepted farther than the line $x_1 = 0$. As a result, the trajectories diverge further and further away from the origin and the closed-loop system becomes unstable.

Figure 6.71 is deceptive in creating the impression that for every trajectory that hits the switching line, there exists another trajectory on the other side of the switching line that continues the solution beyond the switching line until that new

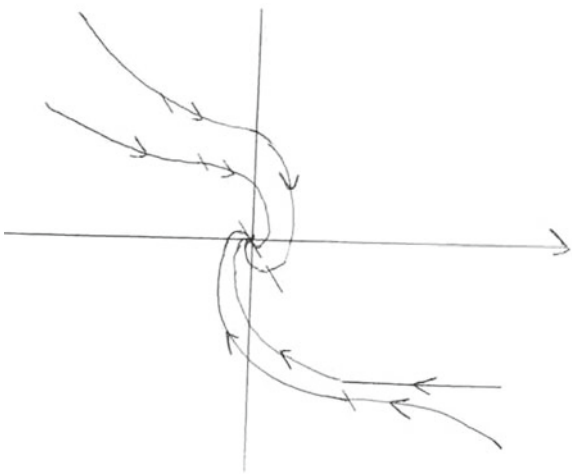


Fig. 6.71 The tilted-to-the-left switching line $x_2 = -(1/T_L)x_1$ (with $T_L > 0$) appears to bring the solution trajectories to a faster convergence to (0,0)

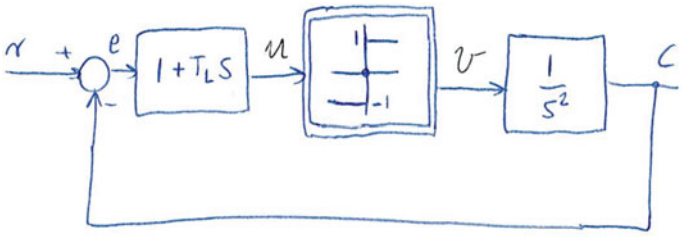


Fig. 6.72 A double integrator process driven by relay with unity levels driven by a PD controller

trajectory again hits the switching line, this time at a point closer to the origin and eventually very few switchings suffice to bring the system arbitrarily close to a steady-state position in which the error in tracking any step input is zero.

What really happens is that two trajectories (each belongs to a different model due to the respective relay output level) may meet at a point on the switching line with opposing motions. We shall come back to the above example after studying another, simpler to analyze, control example.

Example 6.4.2: Relay Control of a Double Integrator Process Using Position and Velocity Feedback to Control the Relay Switching Line

The only reason that the double integrator with PD control relay input example (shown in Fig. 6.72) is simpler than the one that has a process with one integrator and one pole (Fig. 6.70) is that for the former, the phase plane trajectories have simpler equations. The solution approach of both examples is the same. Again, we define the error signal $e = c - r$ and its derivative as the state variables of the system.

The analysis of Example 6.4.2 can be done for parabolic inputs $r(t) = \alpha + t + \gamma t^2$ because due to the double integrator, the time variable t does not appear explicitly in the state equations. However, let us consider only step inputs. The system equations are

$$-\ddot{e} = \begin{cases} 1 & e + T_L\dot{e} > 0 \\ 0 & e + T_L\dot{e} = 0 \\ -1 & e + T_L\dot{e} < 0 \end{cases} \tag{6.35}$$

The corresponding state-variable model is

$$\begin{aligned} \frac{dx_1}{dt} &= x_2 \\ \frac{dx_2}{dt} &= \begin{cases} -1 & x_1 + T_L x_2 > 0 \\ 0 & x_1 + T_L x_2 = 0 \\ 1 & x_1 + T_L x_2 < 0 \end{cases} \end{aligned} \tag{6.36}$$

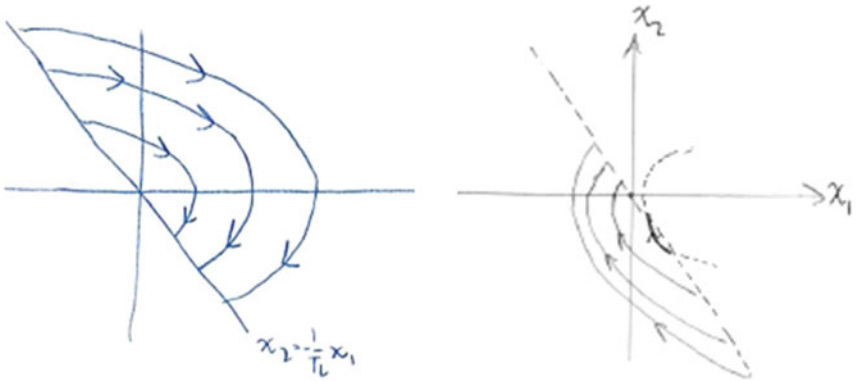


Fig. 6.73 System trajectories when relay output is positive

For the half plane that lies to the right of the switching line $x_2 = -\frac{1}{T_L}x_1$, the solution trajectories are

$$\frac{dx_2}{dx_1} = -\frac{1}{x_2} \Rightarrow \frac{1}{2}x_2^2 - \frac{1}{2}x_2(0)^2 = -x_1 + x_1(0) \quad \text{for} \quad x_1 + T_L x_2 > 0 \tag{6.37}$$

A sketch of these parabolas is given in Fig. 6.73 (Left). Every such trajectory intersects the switching line twice—above $x_2 = 0$ and below that line. Same situation is on the other side of the switching line (Fig. 6.73 Right). There clearly exist points along the switching line where trajectories meet head to head, for instance in cases where a peak of a parabola lies outside the domain of validity of such a parabola.

Let us "stitch together" the trajectories that belong to each side, as shown sketched in Fig. 6.74. We see that farther away from (0,0) the trajectories cross continuously from side to side (of the switching line), whereas near the origin (0,0) trajectories appear unable to continue to the other side. So what happens to such a solution? Where does it go from that collision point?

With an ideal relay (that reacts instantly to input changes), the solution will indeed get stuck at a certain point located on the switching line.

As engineers we know that we can safely assume that the relay is never ideal. It must have some big or small imperfections, for instance, a small delayed response. Such a tiny imperfection allows each trajectory to progress infinitesimally small distance into the other side. This results in a strange sliding motion along the switching line towards the origin (0,0). An exaggerated illustration is shown in Fig. 6.75. The sliding motion consists of high-frequency switchings of the relay as the phase plane solution converges to (0,0) along the switching line.

In general, a phase plane solution that moves along a straight line to the origin is equivalent in the time domain to a first-order linear exponential decay with a time constant that is equal to the coefficient T_L determined by the slope of that line.

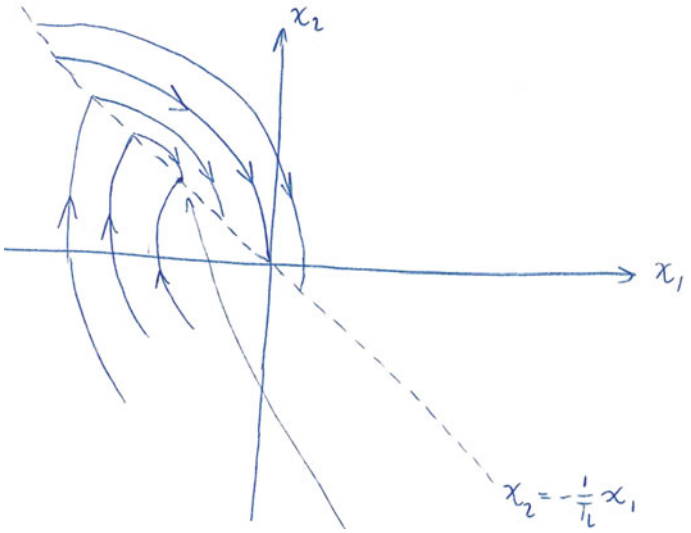


Fig. 6.74 Switched double integrator phase plane trajectories, up to a time where these get stuck at a finite distance from (0,0)

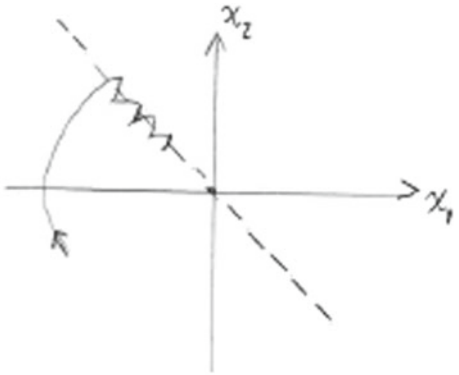


Fig. 6.75 Illustration of a sliding motion along a switching line

Another way of looking at such a sliding convergence of the solution to steady state is to observe the output $c(t)$ which converges with a time constant T_L towards the step input value. The relay high-frequency switchings appear as high-frequency noise on top of the exponential convergence of $c(t)$. The closer to perfection the relay the smaller is the noise amplitude. However, a more alarming node (in the block diagram) to observe the high-frequency relay switching is the process control signal $v(t)$ that exhibits high-frequency large-amplitude noise-looking behavior. Indeed, in the early days of servomechanisms, the sliding phenomenon caused fatigue collapse of many expensive relays, and it was considered a severe blunder by the control engineering team.

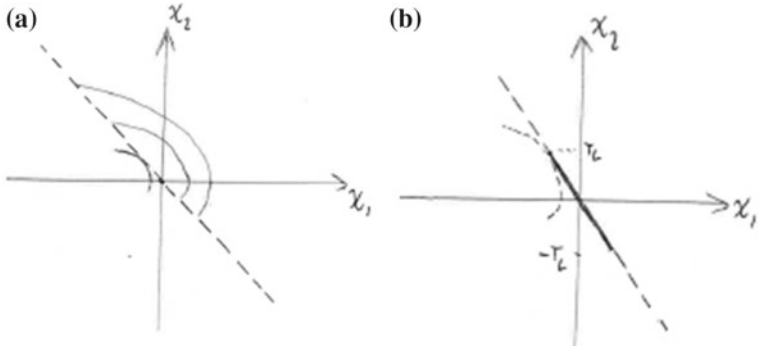


Fig. 6.76 **a** The onset of a sliding phenomenon is at a point where the solution trajectory becomes tangent to the switching line, **b** the resulting "sliding regime"

The switching line segment along which the system's solution slides is denoted "the sliding regime". The onset of the sliding motion occurs whenever a parabola becomes tangent to the switching line. This allows the determination of the size of the sliding regime.

The tangency of the parabolic trajectory to the switching line (Fig. 6.76a) is expressed as

$$\frac{dx_2}{dx_1} = -\frac{1}{x_2} = -\frac{1}{T_L} \tag{6.38}$$

The resulting sliding regime is therefore

$$-T_L \leq x_2 \leq T_L \tag{6.39}$$

A Simulink model for Example 6.4.2 is shown in Fig. 6.77 for $T_L = 0.5$, a step command of 2.5 and zero initial conditions.

In Fig. 6.78a: As trajectories intersect the switching line outside $-0.5 \leq x_2 \leq 0.5$, there is a continuous transition to trajectories on the other side of the

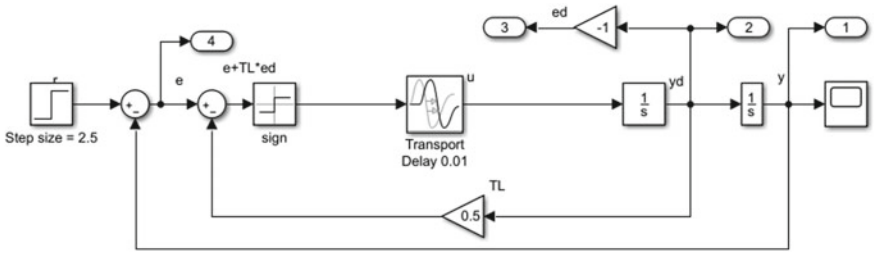


Fig. 6.77 Simulink model for the double integrator servomechanisms that exhibits a finite sliding regime phenomenon

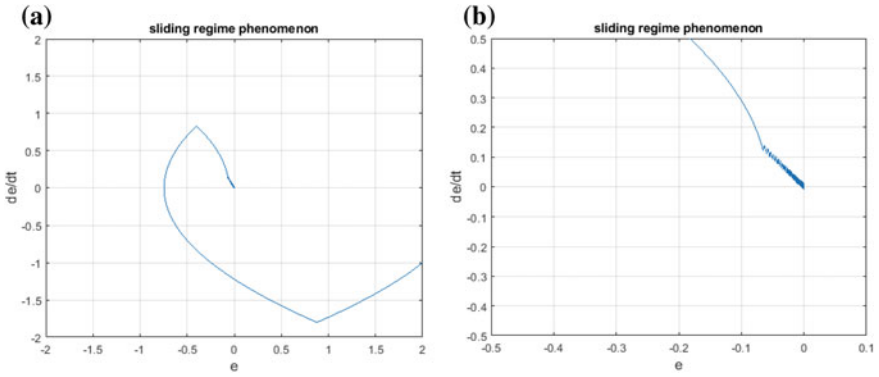


Fig. 6.78 Simulation results of the model of Fig. 6.77

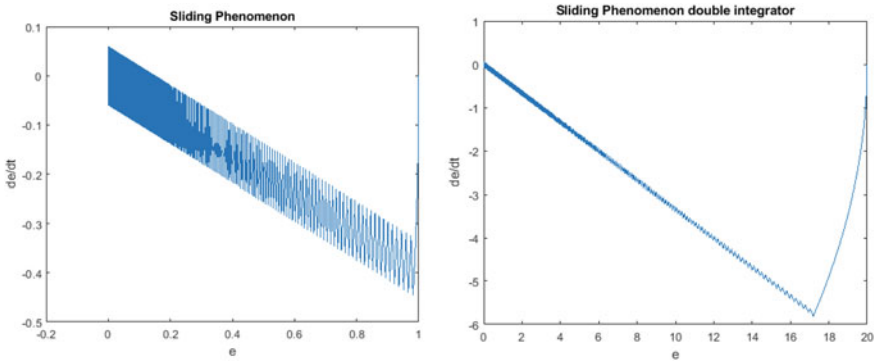


Fig. 6.79 More zoomed-in sliding motions

switching line; (b): Zoomed-in convergence to (0,0). As a trajectory intersects the switching line inside the range $-0.5 \leq x_2 \leq 0.5$, the remaining convergence is by sliding motion along the switching line.

Figure 6.79 demonstrates more zoomed-in sliding motions obtained from other models under other conditions. The motion consists of high-frequency switching actions, depending on the level of relay imperfection. The closer to ideal the relay the higher is the frequency of switchings and the lower the amplitude of output oscillations along the exponential convergence time trajectory.

Back to the Solution of Example 6.4.1

Let us analyze first Eq. (6.34) that is valid over the half plane $x_1 + T_L x_2 > 0$. By symmetry of the problem, the analysis for the half plane $x_1 + T_L x_2 < 0$ is analogous. By (6.34), the trajectory slope at any point (x_1, x_2) in the half plane is:

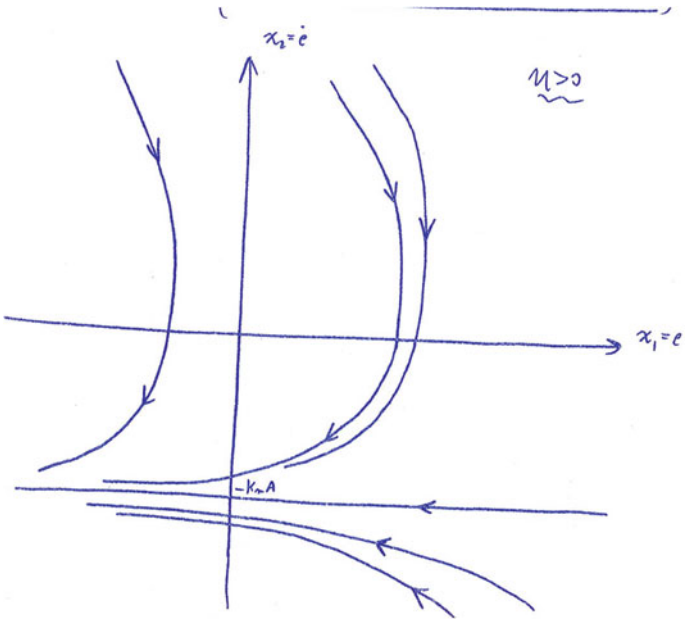


Fig. 6.80 Relay servomechanism trajectories if the relay is stuck at its positive level

$$\frac{\mathrm{d}x_2}{\mathrm{d}x_1} = \frac{-\frac{1}{\tau_m}x_2 - \frac{K_m A}{\tau_m}}{x_2} \qquad x_1 + T_L x_2 > 0 \tag{6.40}$$

Let us ignore (temporarily) the switching line $x_2 = -\frac{1}{T_L}x_1$ and observe the entire family of trajectories that correspond to a positive control signal $u > 0$. See Fig. 6.80.

In particular, we note the fact that $\frac{\mathrm{d}x_2}{\mathrm{d}x_1} \to -\frac{1}{\tau_m}$ if $|x_2|$ becomes large.

Let us assume first that $T_L > 0$ is small compared to the servomechanism time constant τ_m. The sketched trajectories illustrated in Fig. 6.81 clearly indicate the existence of some sliding motion along the switching line. In order to find the size of the sliding regime (for the case of $\tau_m > T_L > 0$), we need to look for trajectories that become tangent to the switching line. From (6.40), we observe that

$$\frac{\mathrm{d}x_2}{\mathrm{d}x_1} = \frac{-\frac{1}{\tau_m}x_2 - \frac{K_m A}{\tau_m}}{x_2} = -\frac{1}{T_L} \qquad x_2 > 0 \tag{6.41}$$

The solution of (6.41) is

$$x_2 = \frac{\frac{K_m A}{\tau_m}}{\frac{1}{T_L} - \frac{1}{\tau_m}} = \frac{K_m A T_L}{\tau_m - T_L} \tag{6.42}$$

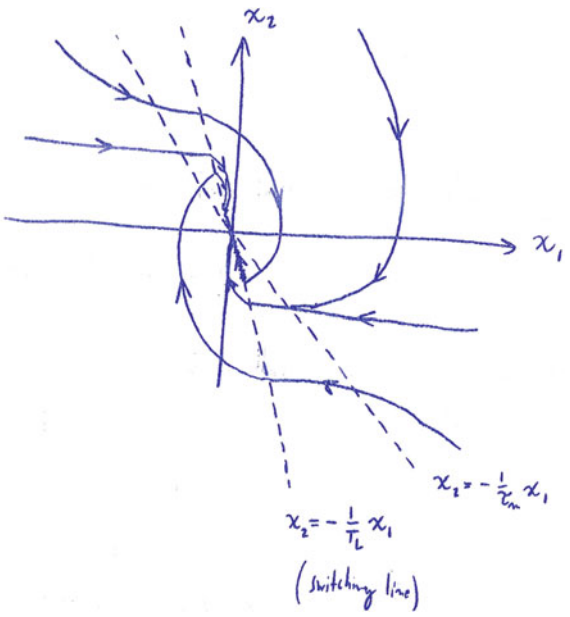


Fig. 6.81 Trajectories for the case where the switching line time constant is smaller than that of the process

We see that whenever the switching line control parameter $T_L \ll \tau_m$, the sliding regime is very small but the system response is very underdamped. As T_L is increased, the sliding regime becomes larger and starting at $T_L = \tau_m$ and beyond it becomes infinite, as shown in Fig. 6.82.

For the case $T_L > \tau_m$, the calculation done at (6.42) is of course no longer valid. Recalling the fact that for large absolute values of x_2, the slope of the trajectories approaching $-1/\tau_m$ establishes the result that each trajectory ends up in sliding.

What is the significance of this solution? Why is it that Example 6.4.1 inspired the development of one of the most powerful nonlinear control techniques called sliding mode control (to be studied later in this chapter)?

We note that the servomechanism converges to zero steady-state error along a known time constant T_L despite potential large modeling errors in K_m and τ_m. The creation of a sliding line is a form of robust control but what do we do about the high-frequency relay oscillations?

Figure 6.83 shows a Simulink diagram of the servomechanism for the parameters $A = 2$, $K_m = 3$ and $\tau_m = 0.5$. The control parameter $T_L = 1$. The relay nonideality is modeled by means of a small pure time delay (of 0.01). The resulting sliding motion (shown in Fig. 6.84) is indeed unpleasantly oscillatory/noisy.

What ended up making sliding mode solutions attractive and acceptable was the important engineering observation that high-gain saturation amplifiers can always replace the nonideal relays. By doing so, the overall robust control solution intent is preserved whereas the control signal becomes less noisy and oscillatory.

Figure 6.85 demonstrates how to create a simulation diagram of a high-gain saturating amplifier. The starting point is a unity saturation block (with output levels

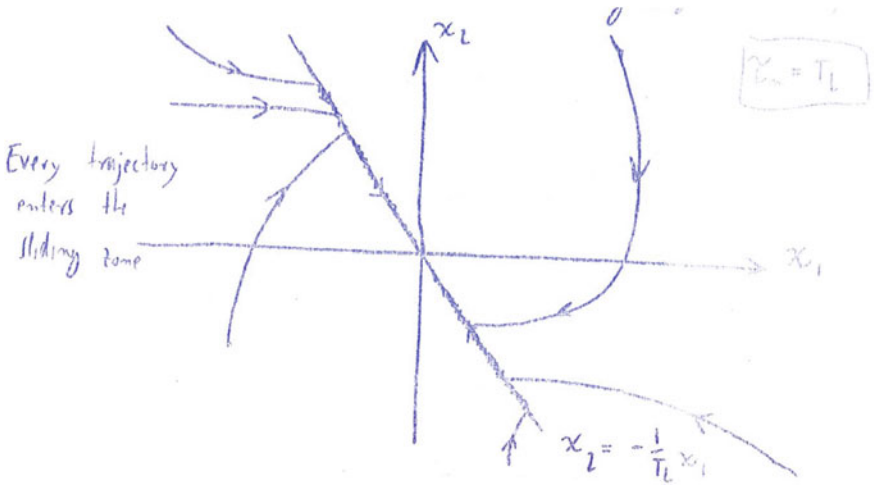


Fig. 6.82 Infinite sliding regime when the control time constant exceeds the process time constant

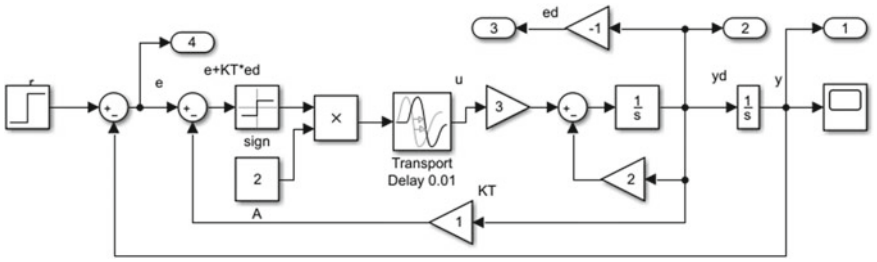


Fig. 6.83 Simulation diagram of the PD-controlled relay servomechanism in which the system enters a sliding mode motion for any step input and any initial conditions

of +1 and −1, and a linear region that extends between the input values of −1 and +1). Post-multiplying the unity saturation block by a gain K sets the saturation output levels to $+K$ and $-K$. Pre-multiplying the saturation block by a gain B sets the linear region limits at $-1/B$ and $+1/B$, as shown in Fig. 6.86 for $K = 5$ and $B = 4$.

Figure 6.87 is the simulation diagram of the servomechanism shown in Fig. 6.83 when the ideal relay and the time delay block were replaced by a high-gain amplifier with a gain of 100.

The resulting phase plane trajectory (Fig. 6.88) retains the sliding line without high-frequency oscillations. Figure 6.89 shows the control loop output signal approaching steady state along a linear first-order time constant created by the gain T_L and the control signal that has no oscillations at all.

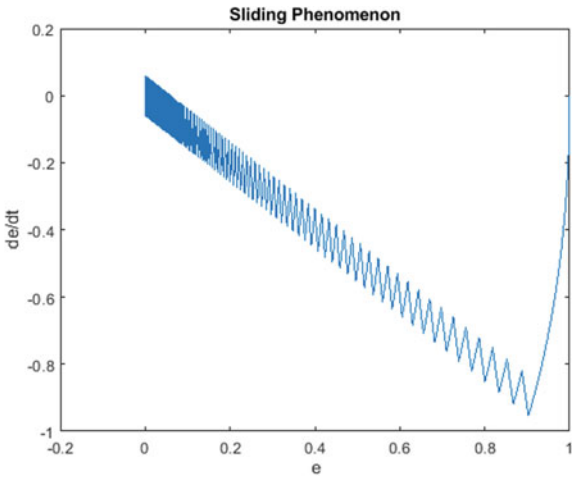


Fig. 6.84 Sliding motion of the servomechanism system of Example 6.4.1

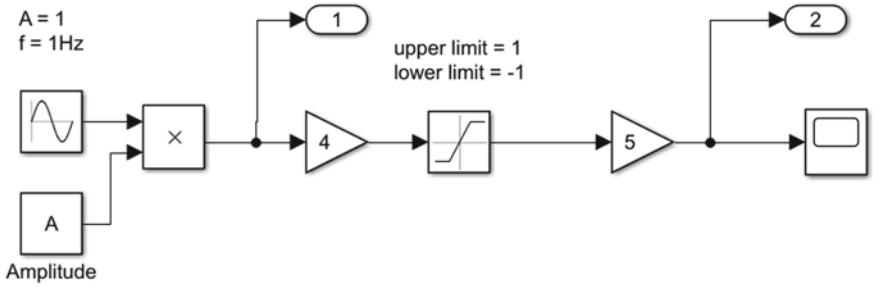
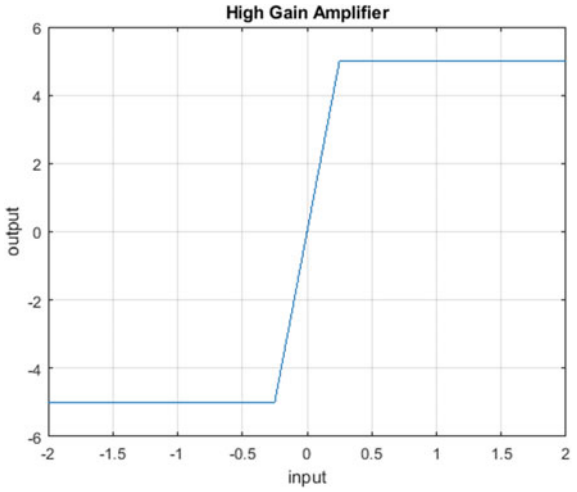


Fig. 6.85 Simulation of a high-gain saturation amplifier

Fig. 6.86 The resulting input–output curve of the simulation diagram of Fig. 6.85



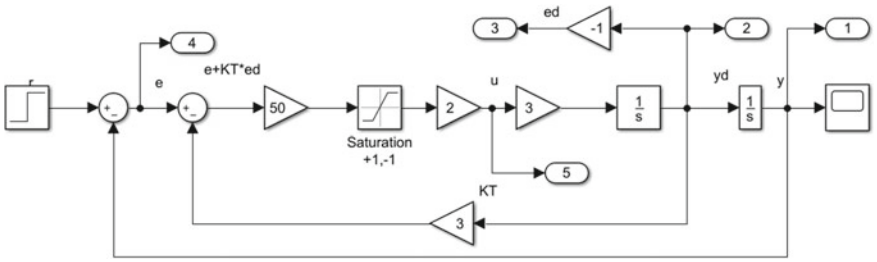


Fig. 6.87 Replacing the relay by a high-gain amplifier

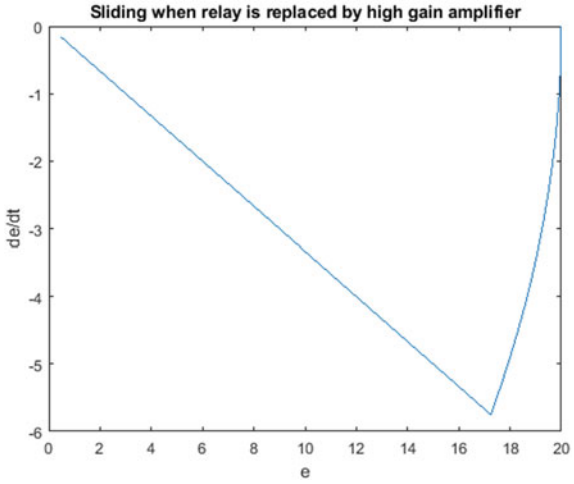


Fig. 6.88 Solution trajectory whenever the relay is replaced by a high-gain amplifier

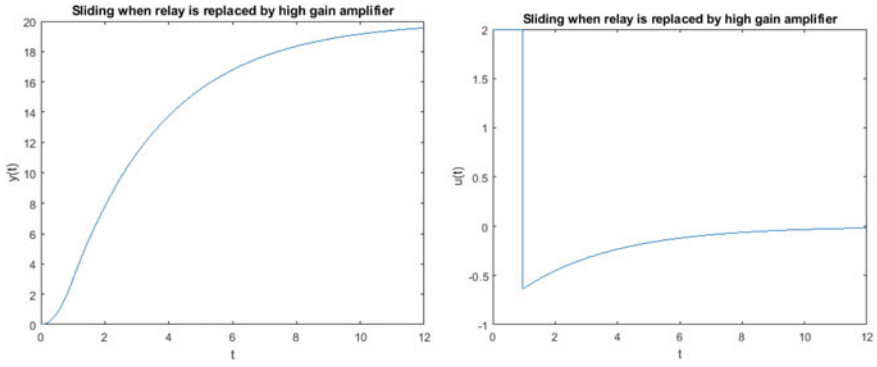


Fig. 6.89 The output $y(t)$ (left) and the control signal $u(t)$ for the sliding servomechanism. All that was needed was to replace the relay by a saturating amplifier

6.4.3 Minimum-Time Relay Control—Bang–Bang Control

So far, we have seen two rather simplistic ways of creating input signals to the relay actuator that is in the servomechanism control loop. The first was a proportional control by which the error signal drives the relay, and we say that the relay has a switching line $x_1 = 0$, where x_1 is the error signal. The second is a PD control by which a combination of the error signal and its derivative drive the relay. The relay still has a switching line but the line is inclined. Can we replace the switching line of the relay by a switching curve? If so, for what purpose?

A well-known relay control method is called "Bang–Bang" control. In such a control, the goal is to bring the state of the system from an arbitrary initial condition to a zero state in minimum time. In order to illustrate the idea in its basic form, let us assume that a car must go from point A to point B in minimum time, along a straight line. Once the car reaches the target point B, it must stop there. We further assume that the car has a known maximum acceleration capability. It also has a known maximum deceleration. If both acceleration and deceleration are equal, then the optimal control strategy for the car would be to maximally accelerate half of the way. Then exactly at the road half-point, to maximally decelerate, and finally at the target point to apply zero control, in order to stay at the target point.

If the car's deceleration ability is twice larger than the acceleration ability, then the optimal minimum time control strategy is to maximally accelerate 2/3 of the way, and at that point to switch to maximum deceleration, with a final stopping at the target.

We see that every car's minimum time control scenario involves only three levels of the car control signal, something that can be provided by a three-position relay. The hardest part of the control is to create a programmable sensory-driven switching strategy.

Example 6.4.3: Bang–Bang Control of the double integrator process
Let us again consider the double integrator servomechanism discussed in Example 6.4.2 [8]. The control system shown in Fig. 6.90 has a goal of a minimum time closed-loop step response.

We recall from Example 6.4.2 that the phase plane solution trajectories for the case $v = 1$ are a set of parabolas, and similarly are the trajectories for $v = -1$.

Whenever $u > 0$ or $u < 0$ the control system has no equilibrium (Fig. 6.91). Yet, the goal is to bring the system to (0,0) and stay there. Are there any trajectories of the $u > 0$ case that pass through the origin? Yes, just one. Likewise, are there trajectories of the $u < 0$ case that pass through (0,0)? Again, yes but just one.

These solution trajectories that pass through the target state give rise to the programmable switching curve that is needed for minimum time control (Fig. 6.92).

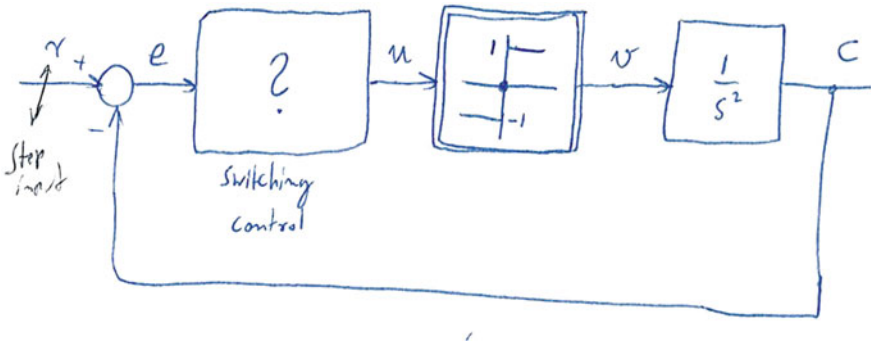


Fig. 6.90 Minimum step response time closed-loop control of a double integrator process. What should be the switching algorithm applied to the error signal $e(t)$?

The mathematical expression of the optimal (minimum-time) switching curve is

$$2x_1 = -x_2|x_2| \quad \text{or} \quad 2e = -\dot{e}|\dot{e}| \tag{6.43}$$

Figure 6.93 illustrates how the switching curve works for any arbitrary initial state of the system: Depending on the specific half plane that the initial state is a correct relay output signal is applied. The switching to the opposite relay output level is initiated upon sensing of an intersection with the switching curve. Thereafter, the system converges to the (0,0) state along a solution that passes through that state. A final switching to the relay zero output happens upon arrival at the target state.

The control $f(e)$ may be implemented either as a step-by-step sensory-driven algorithm, or may be automated by a single nonlinear control formula $u = \text{sign}(e + \frac{1}{2}\dot{e}|\dot{e}|)$ as was done in the Simulink diagram given in Fig. 6.94.

The simulation is stopped upon arrival at the destination.

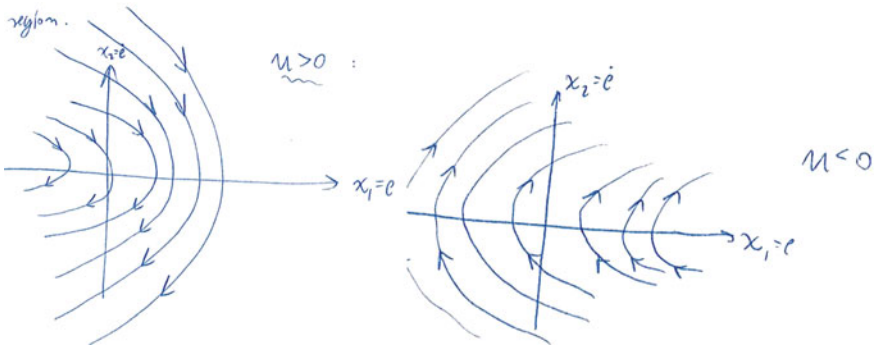


Fig. 6.91 Family of trajectories for each constant level of the relay output

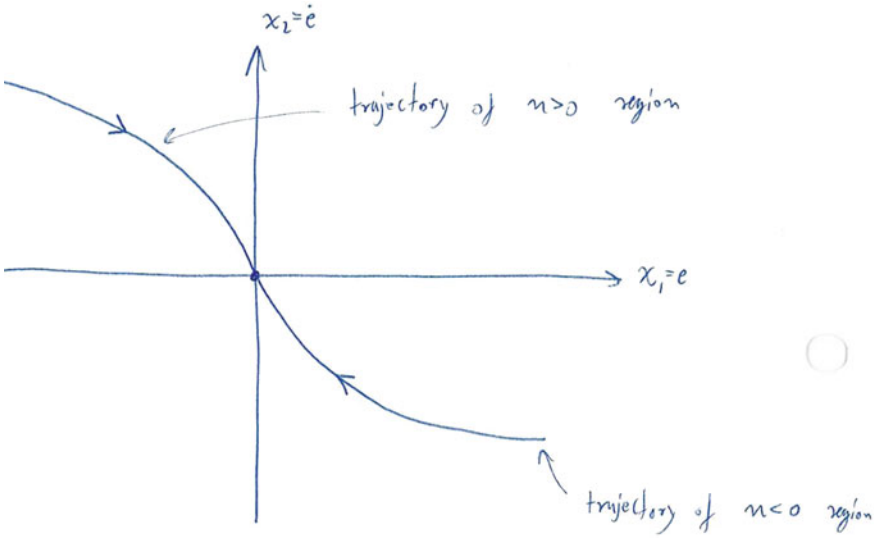


Fig. 6.92 The optimal switching curve consists of solution trajectories that pass through the final state

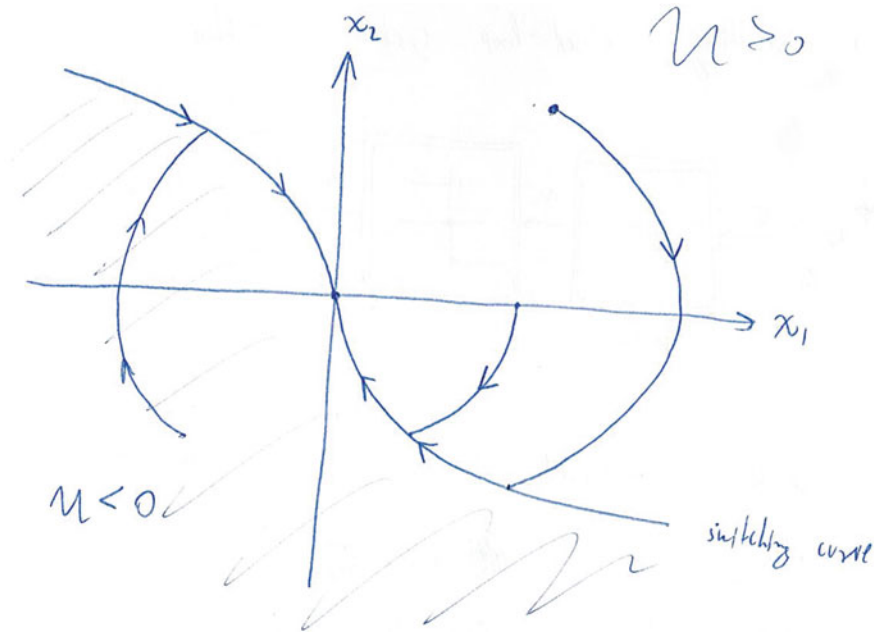


Fig. 6.93 Optimal solution trajectories that utilize the optimal switching curve

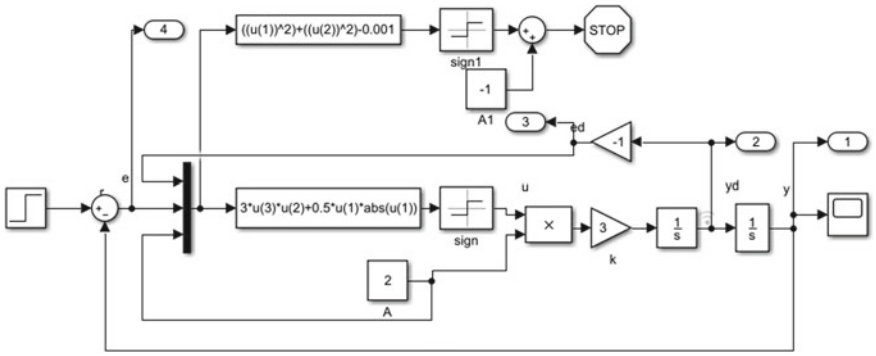


Fig. 6.94 Simulink diagram of the double integrator bang–bang control system

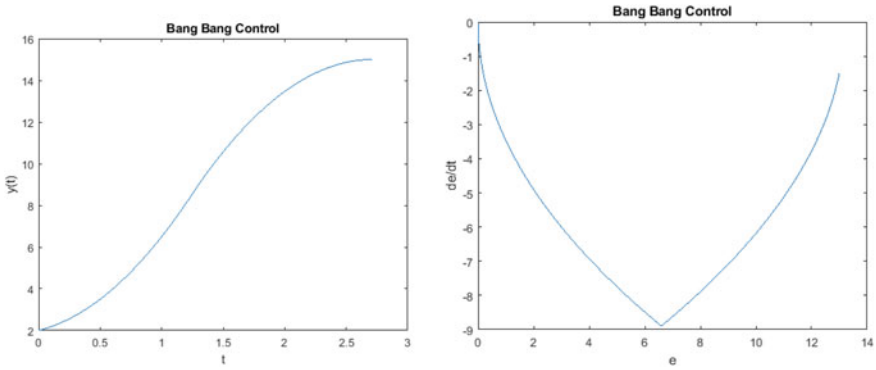


Fig. 6.95 Fast output convergence as the control algorithms is implemented perfectly

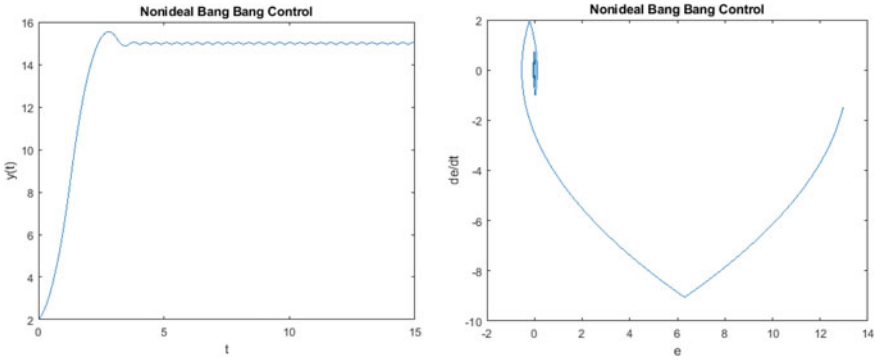


Fig. 6.96 Bang–Bang performance when the relay has a time delay of 0.03 s

Obviously, this control strategy requires extreme timing precision. Figure 6.95 demonstrates a good and precise performance, whereas Fig. 6.96 shows what happens if the relay has a small delay.

The control method has been extended to nth-order control systems. It should be cautioned that a minimum-time control strategy is tailored for a specific command input. It may produce horrible results for a different input signal.

6.5 Sliding Mode Control

The sliding phenomena discovered in certain relay servomechanisms became an inspiration for one of the more powerful practical nonlinear control techniques named sliding mode control (SMC) [9, 10]. SMC is a feedback control strategy for systems of any order that may be nonlinear, time varying, and may have un-modeled dynamics. The goal of SMC is to obtain accurate output tracking of specific input command signals, despite the presence of unknown disturbances with known limits and model parameter variations with known limits.

We use relays during the conceptual control design (and we later replace every relay by a high-gain saturating amplifier for the actual implementation) in order to force the system to slide along a prescribed surface of the state space.

Say that the state vector X lies in an n-dimensional vector space. A **sliding surface** $\{X: S(X; t) = 0\}$ is a manifold of dimension $n - 1$ that acts as a switching boundary for the relays needed for the control. Figure 6.97 illustrates a sliding surface in 3D state space. Sliding surfaces are time varying and keep changing position based on the input command signals.

First, the control strategy causes the system's solution trajectories to converge towards the sliding surface. Next, the control acts to keep the system solution trajectories at the sliding surface at all times. "Being on the sliding surface" amounts to a perfect tracking of the command signal.

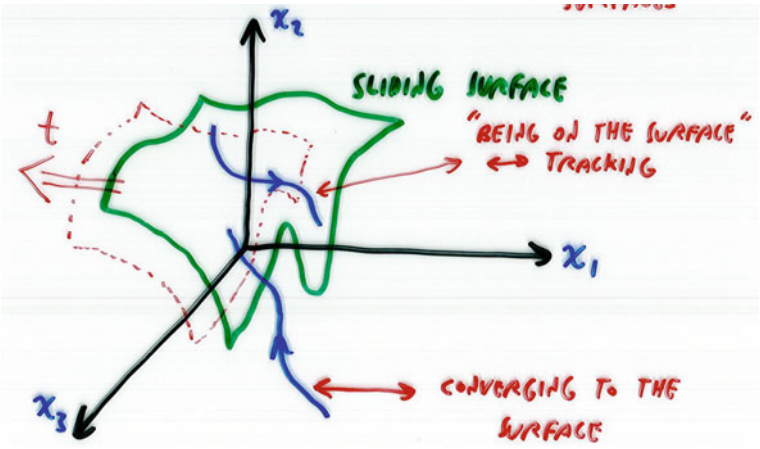


Fig. 6.97 An illustration of a sliding surface

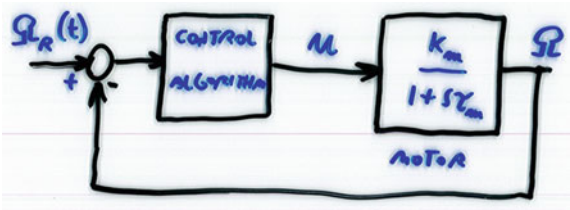


Fig. 6.98 Speed control of a motor

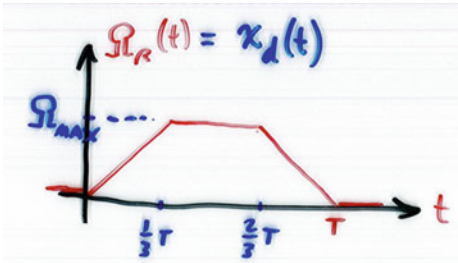


Fig. 6.99 Trapezoidal velocity profile reference command to the motor velocity control

Let us demonstrate the sliding surface idea by means of a DC motor speed control system shown in Fig. 6.98.

Say that we want the motor's velocity $\Omega(t)$ to track a trapezoidal velocity profile specified by the input command $\Omega_R(t)$ shown in Fig. 6.99.

The sliding surface $S(x; t)$ is defined as

$$\begin{aligned} S(x;t) &= x(t) - x_d(t) = 0 \\ x(t) &= \Omega(t) \quad x_d(t) = \Omega_R(t) \end{aligned} \tag{6.44}$$

In this control problem, the system is one-dimensional. Being on $S(x; t)$ means a perfect tracking of the reference signal $x_d(t)$.

We shall come back to the motor speed control example after discussing several preliminary theoretical concepts related to a simpler first-order system. "Simpler" in the sense that the additive control signal $u(t)$ is not multiplied by any system parameter.

6.5.1 *Tracking Control of First-Order Systems with Modeling Uncertainties*

Let us consider the first-order process (6.45) together with the sliding surface (6.44). We shall consider three cases of the coefficient α: constant known parameter, unknown but with known limits and a known nonlinear function of x.

$$\frac{dx}{dt} + \alpha x = u \tag{6.45}$$

First let us understand what it means "to stay at the sliding surface". The control $u(t)$ needs to be such that it forces the proper corrective actions whenever the state $x(t)$ attempts to deviate from $x_d(t)$. If the system solution is such that $S(x; t)$ is positive, there should be a corrective force that takes the solution back to the surface $S(x; t) = 0$. The following conditions must hold locally near the sliding surface:

$$\begin{aligned} \frac{dS(x;t)}{dt} &< 0 \quad \text{if } S(x;t) > 0 \\ \frac{dS(x;t)}{dt} &> 0 \quad \text{if } S(x;t) < 0 \end{aligned} \tag{6.46}$$

An equivalent condition (6.46′) combines the two inequalities into a single inequality:

$$\frac{dS^2(x;t)}{dt} = 2S(x;t)\frac{dS(x;t)}{dt} < 0 \tag{6.46′}$$

We observe that

$$\frac{dS(x;t)}{dt} = \frac{dx}{dt} - \frac{dx_d}{dt} = -\alpha x(t) + u(t) - \frac{dx_d}{dt} \tag{6.47}$$

That is, the control signal $u(t)$ influences directly $dS(x; t)/dt$ and hence it is capable (if chosen correctly) to assure that condition (6.46′) is met.

Condition (6.46′) is only a local condition. That is, it is a necessary condition for the system to stay on the sliding surface, but it does not assure that the system is capable of converging to the sliding surface from "far away" (that is, from an initial state $x(0)$ that may deviate greatly from the reference $x_d(0)$.) A stronger condition that is sufficient for the first-order system (6.45) to converge to the sliding surface from any initial state is known as the "Global Convergence Condition":

$$\frac{1}{2}\frac{dS^2(x;t)}{dt} = S(x;t)\frac{dS(x;t)}{dt} < -A|S(x;t)| \tag{6.48}$$

The positive constant $A > 0$ is related to the speed of convergence to the sliding surface. The larger A the faster the convergence.

Let us now explore control signal $u(t)$ candidates in order to meet (6.48). We already know from our previous discussion of relay control servomechanisms that a relay is a useful device to monitor whether or not a certain switching condition is met. We also wish that the control $u(t)$ is a feedback control law. That is, u should depend on the state x (possibly nonlinearly). The following control law, with an undetermined nonlinear state feedback gain $\beta(x)$ and an undetermined relay symmetric output levels k, is to be found by substitution of (6.49) into (6.47) ~ (6.48) [9].

$$u(t) = \beta(x(t))x(t) - k \cdot \text{sign}(S(x;t)) \tag{6.49}$$

The selection process for $\beta(x)$ and k, of the control signal $u(t)$, is shown below

$$\begin{aligned}\frac{1}{2}\frac{dS^2(x;t)}{dt} &= S(x;t)\frac{dS(x;t)}{dt} = S(x;t)\left(\frac{dx}{dt} - \frac{dx_d}{dt}\right)\\ &= S(x;t)\left(-\alpha x(t) + u(t) - \frac{dx_d}{dt}\right)\\ &= S(x;t)\left(-\alpha x(t) + \beta(x)x(t) - k \cdot \text{sign}(S(x;t)) - \frac{dx_d}{dt}\right)\\ &= (-\alpha + \beta(x))S(x;t)x(t) - k|S(x;t)| - S(x;t)\frac{dx_d}{dt} < -A|S(x;t)|\end{aligned} \tag{6.50}$$

The first term (that involves $\beta(x)$) can be made negative if created by a relay that is activated by the sign of $S(x; t)x(t)$ in the following way:

$$\begin{aligned}-\alpha + \beta(x) < 0 \quad &\text{if} \quad S(x;t)x(t) > 0\\ -\alpha + \beta(x) > 0 \quad &\text{if} \quad S(x;t)x(t) < 0\end{aligned} \tag{6.51}$$

In other words, the state feedback gain determines the output levels of the relay that is activated by the signal $S{\cdot}x$.

The second term on the left side of inequality (6.50) involves the output level k of another relay, one that is activated by the signal S alone. This second term together with the third term (that depends on the derivative of the reference input signal) can become jointly negative if k is made larger than the maximum absolute level of the derivative of the input.

$$k > \left|\frac{dx_d}{dt}(t)\right|_{\max} \tag{6.52}$$

For example, the trapezoidal signal shown in Fig. 6.99 has an absolute maximum derivative of $3\Omega_{MAX}/T$.

Conditions (6.51)∼(6.52) assure that the global convergence condition is met.

Let us now study the SMC implementations for three cases of the system coefficient α.

Case 1: α is Constant and Fully Known

We set the state feedback gain $\beta(x)$ to a constant equal to α. This eliminates the unnecessary second relay brought in (6.51). The theoretical SMC control signal is therefore (by 6.49):

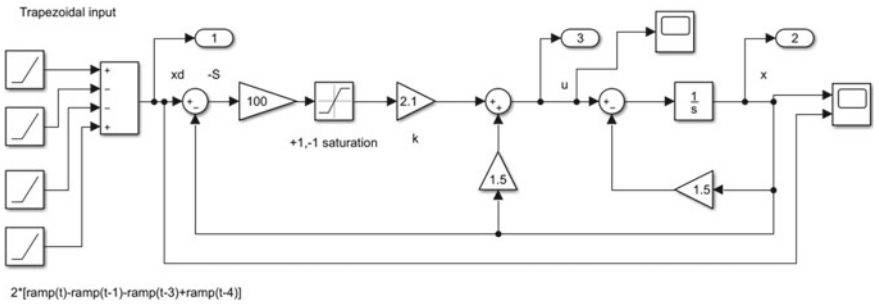


Fig. 6.100 Sliding mode control of first-order linear process with a known time constant

$$u(t) = \alpha x(t) - k \cdot \text{sign}(S(x;t)) \quad \text{where} \quad k > \left|\frac{dx_d}{dt}\right|_{\max} \tag{6.53}$$

A Simulink diagram for Case 1 is shown in Fig. 6.100.

In the example $\alpha = 1.5$. The trapezoidal input is obtained by combining four ramp signals with a slope of 2 (three of them are delayed). The input signal has a maximum derivative of 2 and therefore the relay level k was taken as 2.1. The relay has been replaced by an amplifier with a gain of 210 (and a linear zone width of 0.01) and a saturation level of 2.1. Figure 6.101 shows how $x(t)$ tracks $x_d(t)$ note that $x(0)$ differs from $x_d(0)$.

Replacing the relay by a high-gain saturating amplifier removed all the high-frequency chattering from the control signal $u(t)$ (shown in Fig. 6.102a).

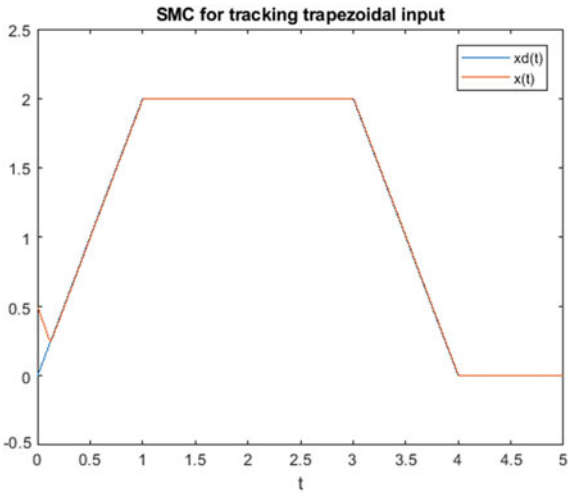


Fig. 6.101 Tracking performance of the system of Fig. 6.100

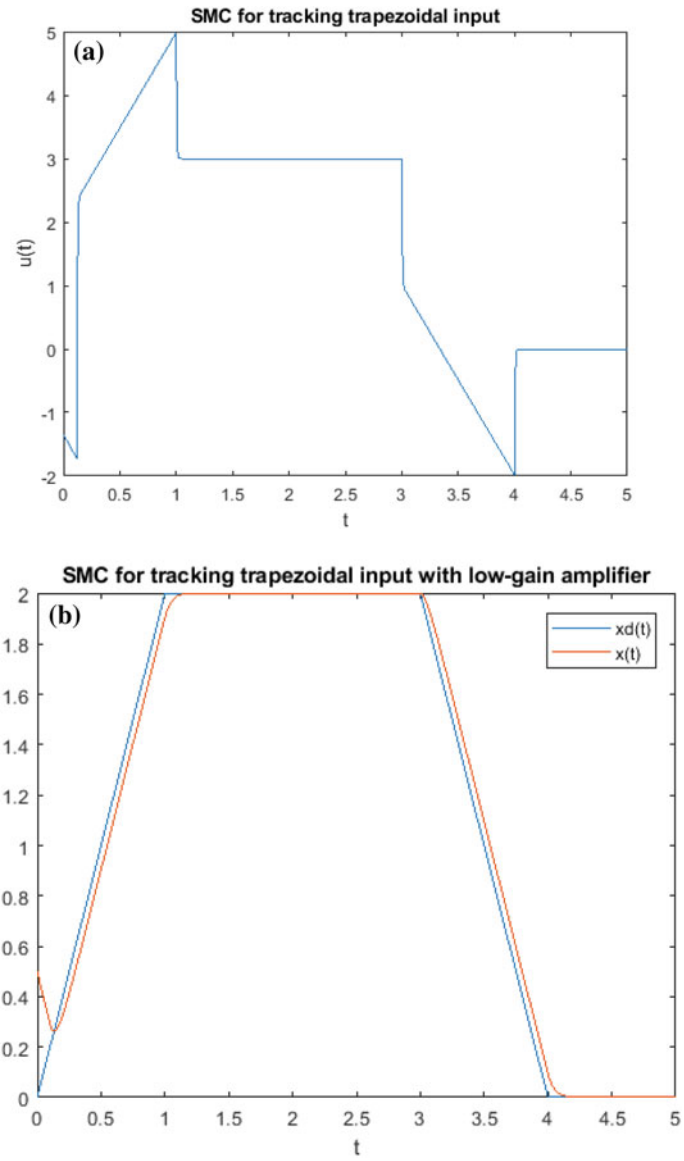


Fig. 6.102 **a** The control signal $u(t)$ of the system of Fig. 6.100. **b** Degraded tracking performance as the linear region of the high-gain amplifier is widened by a factor of 10

How large should the gain of the high-gain amplifier be? There is a tradeoff between tracking performance accuracy and the amount of control effort. Figure 6.102b shows the tracking performance as the saturating device's preamplifier gain is reduced by a factor of 10.

Case 2: α is Fixed and Unknown, with Known Upper and Lower Limits

The first-order system's coefficient α is assumed to be fixed and unknown, having known upper and lower limits:

$$\alpha_{\min} \leq \alpha \leq \alpha_{\max} \tag{6.54}$$

Now we to go back to the derivation of the convergence to the sliding surface condition to observe the effect on the control law design.

$$\frac{1}{2}\frac{\mathrm{d}S^2(x;t)}{\mathrm{d}t} = (-\alpha + \beta(x))S(x;t)x(t) - k|S(x;t)| - S(x;t)\frac{\mathrm{d}x_d}{\mathrm{d}t} < -A|S(x;t)| \tag{6.50}$$

Previously, when α was constant, we simply chose $\beta(x) = \alpha$ to make the first term zero. Now, in order to make the first term negative, a relay action is needed:

$$\begin{cases} \beta(x) = \alpha_{\min} & \text{if } S(x;t)x(t) > 0 \\ \beta(x) = \alpha_{\max} & \text{if } S(x;t)x(t) < 0 \end{cases} \tag{6.55}$$

We recall that the second and third terms of (6.50) are taken care of by a relay that is activated by an input signal $S(x; t)$. The second relay of (6.55) is activated by another signal, $S(x; t)x(t)$. If the amount of any "relay action" may be quantified by the difference between the relay's upper and lower limits, we see that the "relay action" of (6.55) increases with the amount of parameter uncertainty. Figure 6.103 presents the Simulink diagram for the SMC control when the parameter α varies between 3 and 5. The tracking performance is shown in Fig. 6.104.

How precisely do we need to know the values of the upper and lower bounds of the unknown parameter? Figures 6.105 and 6.106 demonstrate the degraded performance if the upper and lower bounds may be in error. In Fig. 6.105, the actual

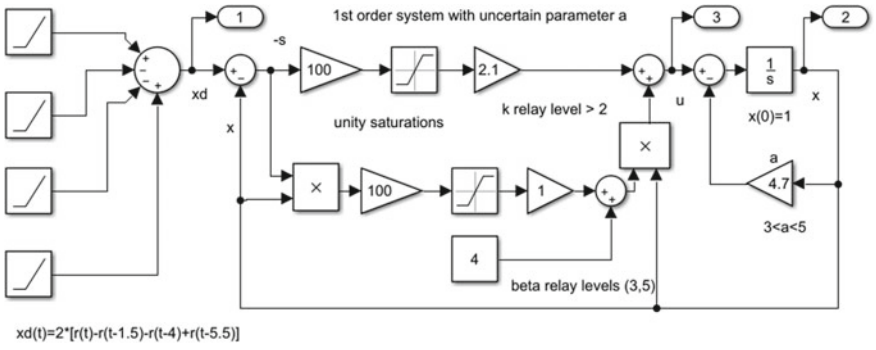


Fig. 6.103 Simulink diagram for SMC control of the first-order system with an unknown parameter of value 4.7

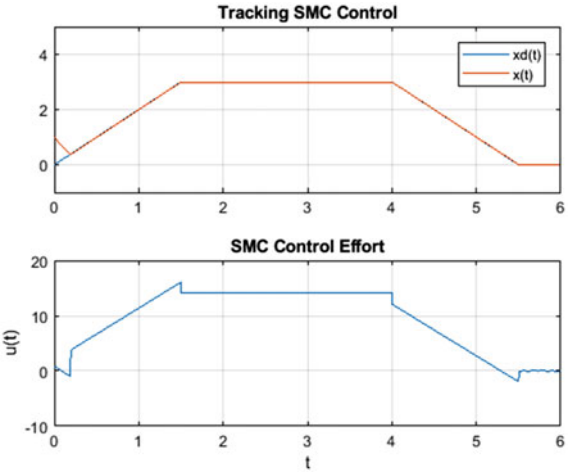


Fig. 6.104 Tracking performance of SMC control of system with unknown parameter

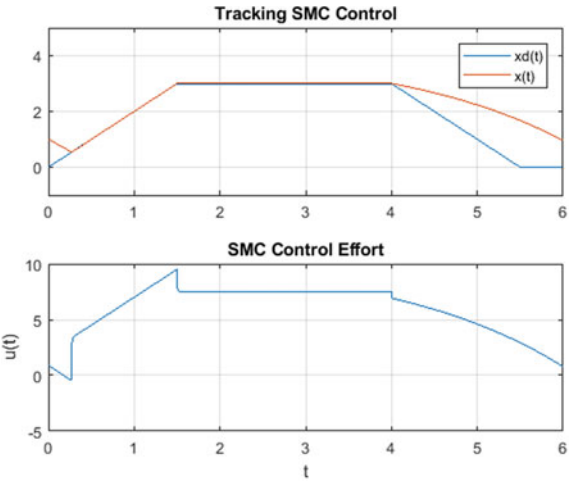


Fig. 6.105 Degraded SMC tracking performance when estimating incorrectly the system parameter's lower value

value of the parameter was set to $\alpha = 2.5$, being outside of the range set by the lower value of 3.

We recall that for this example, as the maximum slope of the trapezoidal input signal was 2, the first relay (the one activated by S) level k had to be larger than that slope of 2. The tracking performance degrades if this condition is violated, as seen in Fig. 6.107, which shows the tracking performance when the relay level is 1.5, smaller than the needed threshold of 2.

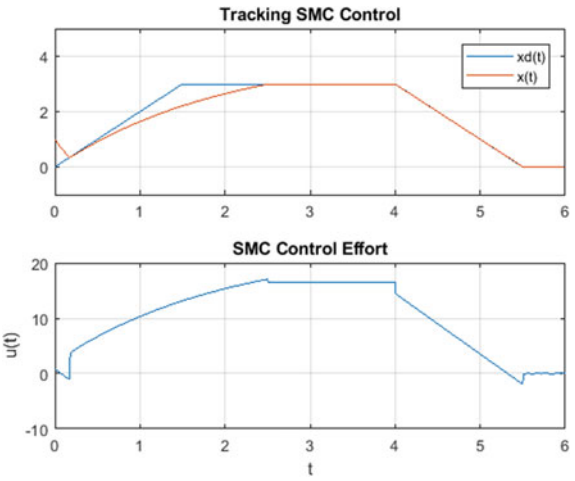


Fig. 6.106 Degraded tracking performance of SMC due to error in estimating the unknown parameter's upper bound. Actual parameter was 5.5 exceeding the design bound of 5

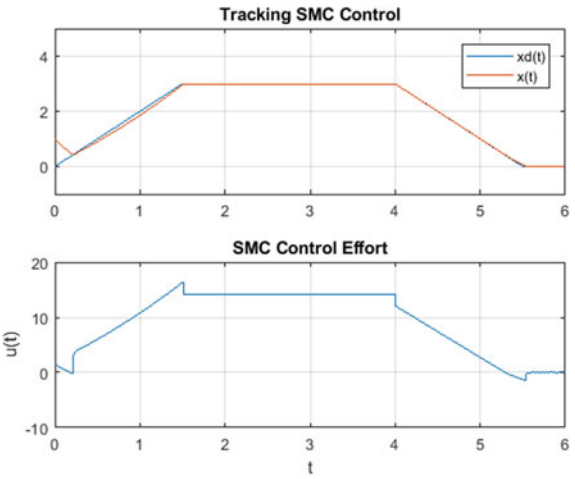


Fig. 6.107 Degraded tracking performance of SMC when a relay level is insufficiently small

Case 3: $\alpha = \alpha(x)$, a Known Nonlinear Function of the State x

For example, let us consider the nonlinear system:

$$\frac{dx}{dt} = (\cos(x))x + u \tag{6.56}$$

One may follow Case 2, observing that the nonlinear state-dependent coefficient can be viewed as an unknown parameter spread between the limits of −1 and +1. A simpler solution, avoiding the use of a second relay, is to select the feedback gain $\beta(x)$ (of 6.50) according to the known nonlinearity.

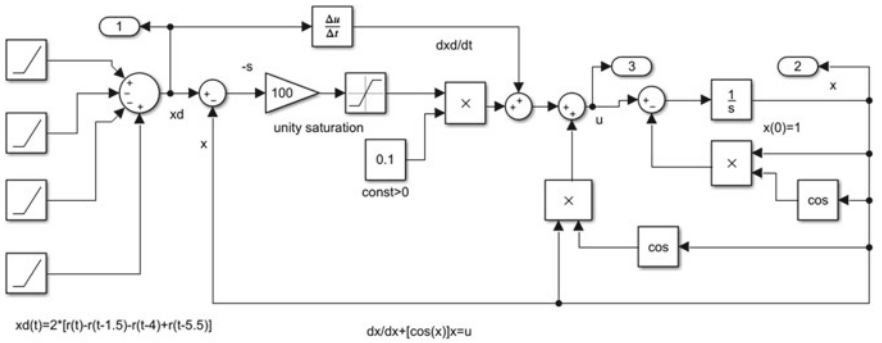


Fig. 6.108 SMC control of the nonlinear model (6.56). The relay level must be changed from 0.1 to a value larger than 2 (due to the slope of the reference signal)

$$u(t) = (\cos(x(t)))x(t) - k \cdot \text{sign}(S(x;t))$$
$$k > \left|\frac{dx_d(t)}{dt}\right|_{\max} \tag{6.57}$$

The Simulink diagram for the SMC control for Case 3 is shown in Fig. 6.108.

A good tracking performance is shown in Fig. 6.109, for relay levels $k = 2.1$. A degraded tracking performance is shown in Fig. 6.110 for $k = 0.5$.

Case 4: Back to Case 2 (of the Unknown Parameter), Using a Single Relay

As may be implied correctly from the simple example of Case 2 to the more general SMC control problem, every unknown parameter in the system model needs its own

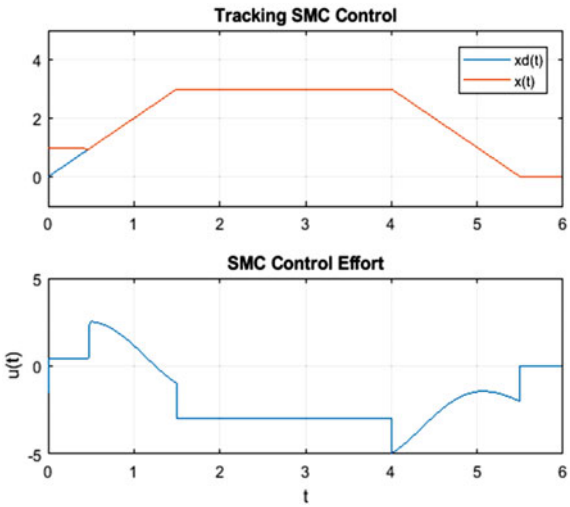


Fig. 6.109 Correct SMC tracking performance of the system of Fig. 6.105

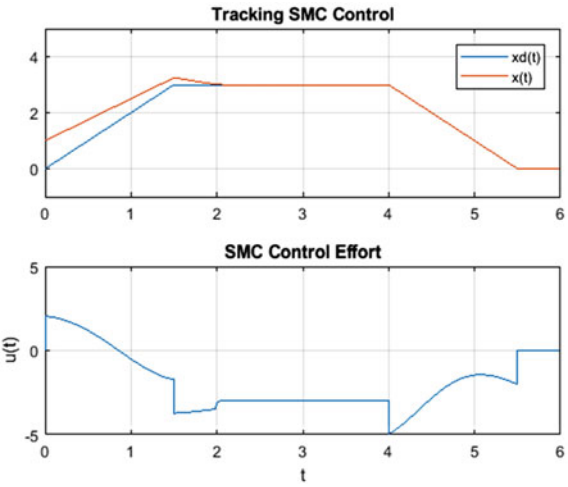


Fig. 6.110 Incorrect SMC tracking performance for insufficient relay action

special relay in order to assure that its effect on the convergence condition is negative, as desired. For instance, a system that has 3 unknown parameters, with known limits, may require 4 relays in order to implement an SMC control. Of course, each of the relays is later implemented by means of a suitable high-gain saturating amplifier.

It is possible to limit the number of relays to one. However, such a relay may need to have state-dependent levels [10]. Practically speaking, all relays (and high-gain amplifiers) used in modern SMC digital control are just digital control software lines of code. State-dependent relay levels are not exceedingly harder to implement than constant relay levels.

The idea of a single state-dependent control relay is illustrated on the model of (6.45) combined with (6.54). Let

$$\Delta\alpha = \alpha_{\max} - \alpha_{\min} \quad \alpha_{\text{average}} = \alpha_{\min} + \frac{\Delta\alpha}{2} \tag{6.58}$$

We define a control signal $u_0(x; t)$, a part of the total control signal $u(x; t)$, to be *our "best approximation" of a control signal that tries to make dS(x; t)/dt = 0*. The other part of $u(x; t)$ involves a relay, activated by $S(x; t)$, with state-dependent levels $k = k(x; t)$ that accomplishes a convergence to the sliding surface.

$$u(x;t) = u_0(x;t) - k(x;t) \cdot \text{sign}(S(x;t)) \tag{6.59}$$

Using again (6.50), let us first choose $u_0(x; t)$ and later see how to select the relay level $k(x; t)$:

$$\begin{aligned}\frac{1}{2}\frac{\mathrm{d}S^2(x;t)}{\mathrm{d}t} &= S(x;t)\frac{\mathrm{d}S(x;t)}{\mathrm{d}t} = S(x;t)\left(\frac{\mathrm{d}x}{\mathrm{d}t} - \frac{\mathrm{d}x_d}{\mathrm{d}t}\right)\\ &= S(x;t)\left(-\alpha x(t) + u(t) - \frac{\mathrm{d}x_d}{\mathrm{d}t}\right)\\ &= S(x;t)\left(-\alpha x(t) + u_0(x;t) - k(x;t)\cdot\mathrm{sign}(S(x;t)) - \frac{\mathrm{d}x_d}{\mathrm{d}t}\right)\\ &\Rightarrow u_0(x;t) = \alpha_{\text{average}}x(t) + \frac{\mathrm{d}x_d}{\mathrm{d}t}\end{aligned} \tag{6.60}$$

Let us substitute the above selection of $u_0(x;\, t)$ into the convergence condition, and find the needed levels of the relay:

$$\begin{aligned}\frac{1}{2}\frac{\mathrm{d}S^2(x;t)}{\mathrm{d}t} &= S(x;t)\left(-\alpha x(t) + u_0(x;t) - k(x;t)\cdot\mathrm{sign}(S(x;t)) - \frac{\mathrm{d}x_d}{\mathrm{d}t}\right)\\ &= S(x;t)\left(-\alpha x(t) + \alpha_{\text{average}}x(t) + \frac{\mathrm{d}x_d}{\mathrm{d}t} - k(x;t)\cdot\mathrm{sign}(S(x;t)) - \frac{\mathrm{d}x_d}{\mathrm{d}t}\right)\\ &= S(x;t)(-\alpha + \alpha_{\text{average}})x(t) - k(x;t)|S(x;t)|\end{aligned} \tag{6.61}$$

We observe, based on (6.58) and (6.61), that

$$(-\alpha + \alpha_{\text{average}})x(t) \le \frac{\Delta\alpha}{2}|x(t)| \tag{6.62}$$

Therefore, the selection of the symmetric state-dependent relay levels $k(x;\, t)$, for the relay that is activated by $S(x;\, t)$, is done as follows:

$$\begin{aligned}&k(x;t) = \frac{\Delta\alpha}{2}|x(t)| + \eta \quad \text{where} \quad \eta > 0\\ &\frac{1}{2}\frac{\mathrm{d}S^2(x;t)}{\mathrm{d}t} = S(x;t)(-\alpha + \alpha_{\text{average}})x(t) - k(x;t)|S(x;t)| < -\eta|S(x;t)|\end{aligned} \tag{6.63}$$

A Simulink diagram of the single-relay SMC control of the first-order system with unknown coefficient is shown in Fig. 6.111, for α that is somewhere between 3 and 5 (actual value was taken to be 4.7). The selection of η affects the speed of convergence from the initial state to the sliding surface. The tracking results for $\eta = 0.1$ are shown in Fig. 6.112, and for $\eta = 0.5$ in Fig. 6.113.

6.5.2 SMC Control Going Beyond First-Order Systems

The previous section covered many of the fundamental principles of sliding mode control, as a tracking control of a first-order control system, that may be nonlinear,

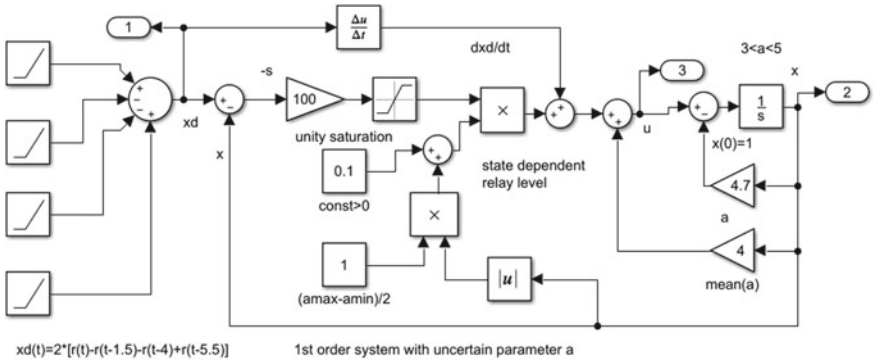


Fig. 6.111 Simulink diagram of a single relay with state-dependent levels SMC control

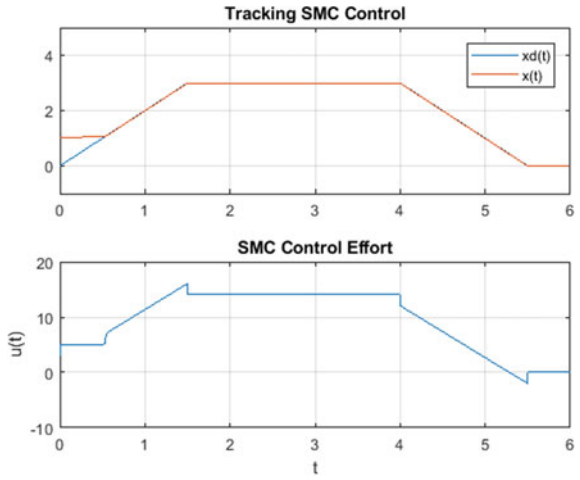


Fig. 6.112 Tracking performance of a single-relay SMC control using small excess level η

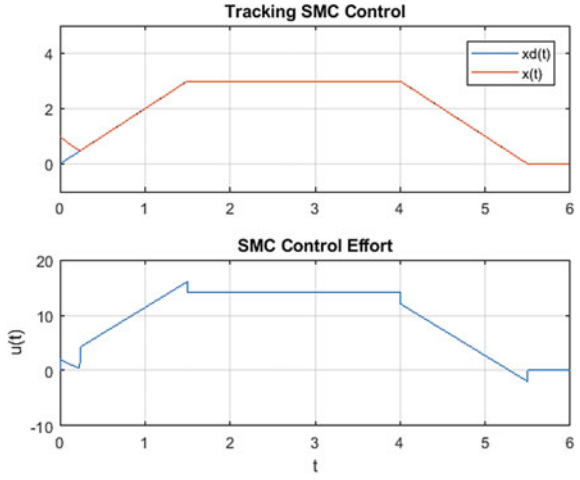


Fig. 6.113 SMC control performance with a single relay, using larger value of η

time varying and may have un-modeled dynamics. There are many follow-up questions that some are briefly discussed in this follow-up section:

(a) How is the SMC control methodology extended to SISO systems of any order?
(b) How is a sliding surface defined for an *n*th-order system? How do we make sure that such a surface is attracting and does not cause system instability?
(c) How is SMC control extended to Multi-Input Multi-Output (MIMO) systems? Can there be multiple sliding surfaces operating simultaneously and synchronously?

Starting with the last question: Yes, as was studied in [9], there can be multiple sliding surfaces operating together. This is important, for example, in many robotic applications. A robot manipulator with N degrees of freedom may be programmed to move its end effector in a prescribed trajectory in *N*-dimensional space. This task, using the robot inverse kinematic model, is broken down into specifying a trajectory for each one of the N robot joints. At each joint level, a controller may be designed so that the joint slides along a specific sliding surface. As shown by Slotine and Li [10] a stable sliding surface may be defined using a single parameter. The extension of the SMC method to an *n*th-order SISO control system may follow the ideas of either the single-relay SMC implementation, or the multi-relay SMC implementation.

Let us limit the discussion of *n*-order SMC control to systems in which the state vector X may be constructed using a scalar output $x(t)$ and its $n-1$ derivatives. We assume that the signal $x_d(t)$ that the system output $x(t)$ is supposed to track is differentiable n times. We then define as ΔX the tracking error vector, whose elements are the output tracking error and the first $n-1$ derivatives of the tracking error.

$$\Delta X = \begin{pmatrix} x \\ \frac{\mathrm{d}x}{\mathrm{d}t} \\ \vdots \\ \frac{\mathrm{d}^{n-1}x}{\mathrm{d}t^{n-1}} \end{pmatrix} - \begin{pmatrix} x_d \\ \frac{\mathrm{d}x_d}{\mathrm{d}t} \\ \vdots \\ \frac{\mathrm{d}^{n-1}x_d}{\mathrm{d}t^{n-1}} \end{pmatrix} = \begin{pmatrix} \Delta x \\ \frac{\mathrm{d}\Delta x}{\mathrm{d}t} \\ \vdots \\ \frac{\mathrm{d}^{n-1}\Delta x}{\mathrm{d}t^{n-1}} \end{pmatrix} \tag{6.64}$$

If $x(t)$ tracks perfectly the reference signal $x_d(t)$ then obviously the derivatives of $x(t)$ track perfectly their counterpart $x_d(t)$ derivatives. In the case of a suboptimal tracking performance, one may set different weighting coefficients to the various tracking error derivatives. A sliding surface for an *n*th-order system may be defined as

$$S(X;t) = c_0\Delta x(t) + c_1\frac{\mathrm{d}\Delta x(t)}{\mathrm{d}t} + \cdots + c_{n-1}\frac{\mathrm{d}^{n-1}\Delta x(t)}{\mathrm{d}t^{n-1}} = 0 \tag{6.65}$$

Obviously, the surface $S(X;\ t) = 0$ must not be inherently unstable. The weighting coefficients $\{c_0, \ldots, c_{n-1}\}$ should be selected to keep system (6.65) stable. A set of acceptable coefficients must satisfy the Routh–Hurwitz stability criterion. One such selection [Slotine-Li] is to take all the roots of the Laplace transform of (6.65) to be identical and equal to a pole located anywhere in the LHP of the s-plane. Let us then define the sliding surface $S(X; t)$ as follows to have all the poles be at $s = -\lambda$, where $\lambda > 0$

$$S(X;t) = \left(\frac{\mathrm{d}}{\mathrm{d}t} + \lambda\right)^{n-1} \Delta x = 0 \tag{6.66}$$

Let us illustrate (6.66) by expanding it for several low values of n:

$$\begin{aligned} n = 1: &\quad S(X;t) = \Delta x = 0 \\ n = 2: &\quad S(X;t) = \frac{\mathrm{d}\Delta x}{\mathrm{d}t} + \lambda \Delta x = 0 \\ n = 3: &\quad S(X;t) = \frac{\mathrm{d}^2\Delta x}{\mathrm{d}t^2} + 2\lambda \frac{\mathrm{d}\Delta x}{\mathrm{d}t} + \lambda^2 \Delta x = 0 \end{aligned} \tag{6.67}$$

We see that for $n = 1$, the definition (6.66) yields the familiar tracking error signal used in the previous section. For higher order systems, there is no loss of generality defining $S(X; t)$ in terms of a single parameter.

6.5.2.1 Slotine-Li's First Generic Example [10]

What we mean by the word "generic" is that the method presented in this example is general enough to easily extend to systems of higher order. Let us consider the following second-order nonlinear control system:

$$\frac{\mathrm{d}^2 x}{\mathrm{d}t^2} = f\left(x, \frac{\mathrm{d}x}{\mathrm{d}t}\right) + u \tag{6.68}$$

The control problem is to design SMC control so that $x(t)$ tracks the signal $x_d(t)$ on a sliding surface defined in (6.67) for the case $n = 2$. The nonlinear part of the model $f(x, \mathrm{d}x/\mathrm{d}t)$ is known imperfectly. That is, we know only some nominal model $f_0(x, \mathrm{d}x/\mathrm{d}t)$ that approximates the actual nonlinear model f, and we also know the bounds of our model uncertainty:

$$\left| f_0\left(x, \frac{\mathrm{d}x}{\mathrm{d}t}\right) - f\left(x, \frac{\mathrm{d}x}{\mathrm{d}t}\right) \right| \le F\left(x, \frac{\mathrm{d}x}{\mathrm{d}t}\right) \tag{6.69}$$

Both $f_0(x, \mathrm{d}x/\mathrm{d}t)$ and $F(x, \mathrm{d}x/\mathrm{d}t)$ are assumed known. The SMC control design may follow the single-relay design strategy introduced earlier. That is, the control

signal $u(X; t)$ is partitioned into to $u_0(X, t)$, designed to approximate $\mathrm{d}S(X; t)/\mathrm{d}t \approx 0$, as best as possible, and a relay action part that is activated by $S(X; t)$:

$$u(X;t) = u_0(X;t) - k(X;t) \cdot \mathrm{sign}(S(X;t)) \tag{6.70}$$

The derivation of $\mathrm{d}S(X; t)/\mathrm{d}t$ is as follows:

$$\begin{aligned} \frac{\mathrm{d}S(X;t)}{\mathrm{d}t} &= \frac{\mathrm{d}^2\Delta x}{\mathrm{d}t^2} + \lambda\frac{\mathrm{d}\Delta x}{\mathrm{d}t} = \frac{\mathrm{d}^2 x}{\mathrm{d}t^2} - \frac{\mathrm{d}^2 x_d}{\mathrm{d}t^2} + \lambda\frac{\mathrm{d}\Delta x}{\mathrm{d}t} \\ &= f\left(x, \frac{\mathrm{d}x}{\mathrm{d}t}\right) + u - \frac{\mathrm{d}^2 x_d}{\mathrm{d}t^2} + \lambda\frac{\mathrm{d}\Delta x}{\mathrm{d}t} \approx 0 \end{aligned} \tag{6.71}$$

The selection of $u_0(X; t)$ is therefore made to be

$$\begin{aligned} &\frac{\mathrm{d}S(X;t)}{\mathrm{d}t} = f\left(x, \frac{\mathrm{d}x}{\mathrm{d}t}\right) + u_0(X;t) - \frac{\mathrm{d}^2 x_d}{\mathrm{d}t^2} + \lambda\frac{\mathrm{d}\Delta x}{\mathrm{d}t} \approx 0 \\ &\Rightarrow u_0(X;t) = -f_0\left(x, \frac{\mathrm{d}x}{\mathrm{d}t}\right) + \frac{\mathrm{d}^2 x_d}{\mathrm{d}t^2} - \lambda\frac{\mathrm{d}\Delta x}{\mathrm{d}t} \end{aligned} \tag{6.72}$$

Finally, it is left to find the state-dependent relay levels $k(X; t)$ by substituting (6.72) into the derivative of $S^2(X; t)$:

$$\begin{aligned} \frac{1}{2}\frac{\mathrm{d}S^2(X;t)}{\mathrm{d}t} &= S(X;t)\frac{\mathrm{d}S(X;t)}{\mathrm{d}t} \\ &= S(X;t)\left[f\left(x, \frac{\mathrm{d}x}{\mathrm{d}t}\right) - f_0\left(x, \frac{\mathrm{d}x}{\mathrm{d}t}\right) - k(X;t)\mathrm{sign}(S(X;t))\right] \\ &= \left[f\left(x, \frac{\mathrm{d}x}{\mathrm{d}t}\right) - f_0\left(x, \frac{\mathrm{d}x}{\mathrm{d}t}\right)\right]S(X;t) - k(X;t)|S(X;t)| \end{aligned} \tag{6.73}$$

Hence, the state-dependent relay levels $k(X; t)$ should be taken according to

$$k(X;t) = F\left(x, \frac{\mathrm{d}x}{\mathrm{d}t}\right) + \eta \quad \eta > 0 \tag{6.74}$$

The model uncertainty envelope $F(x, \mathrm{d}x/\mathrm{d}t)$ may also include time-varying parameters and time-varying disturbance signals. Again, we see that the larger the uncertainty the larger the amplitude of relay control action.

Illustrative Simulation Example of Slotine-Li's First Generic Example

Let the control system that is to be controlled by SMC be

$$\begin{aligned}&\frac{\mathrm{d}^2x}{\mathrm{d}t^2}+a(t)\left(\frac{\mathrm{d}x}{\mathrm{d}t}\right)^2\cos(3x)=u\\&1\le a(t)\le 2\\&x_d(t)=\sin\left(\frac{\pi t}{2}\right)\end{aligned}\tag{6.75}$$

The time-varying coefficient $a(t)$ can be anything. In the following simulation, $a(t)$ was taken as $a(t)=1.5+0.5\cos(2\pi t)$. We have $f(x,\mathrm{d}x/\mathrm{d}t,t)=-a(t)(\mathrm{d}x/\mathrm{d}t)^2\cos(3x)$ and its best estimate is $f_0(x,\mathrm{d}x/\mathrm{d}t,t)=-1.5(\mathrm{d}x/\mathrm{d}t)^2\cos(3x)$. Let us calculate the envelope of uncertainty $F(x, \mathrm{d}x/\mathrm{d}t, t)$:

$$|f_0(x,\mathrm{d}x/\mathrm{d}t,t)-f(x,\mathrm{d}x/\mathrm{d}t,t)|\le 0.5(\mathrm{d}x/\mathrm{d}t)^2|\cos(3x)|=F(x,\mathrm{d}x/\mathrm{d}t,t)\tag{6.76}$$

Let us select $\lambda = 20$ and $\eta = 1$. The sliding surface is therefore $S(X;t)=\frac{\mathrm{d}\Delta x}{\mathrm{d}t}+20\Delta x=0$. The control signal $u(X; t)$ is taken as

$$\begin{aligned}u(X;t)=1.5\left(\frac{\mathrm{d}x}{\mathrm{d}t}\right)^2\cos(3x)+\frac{\mathrm{d}^2x_d}{\mathrm{d}t^2}-20\Delta x\\-\left(0.5\left(\frac{\mathrm{d}x}{\mathrm{d}t}\right)^2|\cos(3x)|+1\right)\mathrm{sign}\left(\frac{\mathrm{d}\Delta x}{\mathrm{d}t}+20\Delta x\right)\end{aligned}\tag{6.77}$$

The Simulink diagram of the system and its SMC controller is shown in Fig. 6.114.

The tracking performance shown in Fig. 6.115 exhibits a long transient time till tracking is attained. As shown in Fig. 6.116 (for a value of η that is 20 times larger), the transient time slightly improves at the price of a larger control signal amplitude.

Generally speaking, the transient time prior to steady-state tracking consists of two component. The first one denoted as t_{reach} is the time that it takes to converge from $S(X; 0)$ to $S(X; t_{\mathrm{reach}}) = 0$. As we integrate $\frac{1}{2}\frac{\mathrm{d}(S^2(X;t))}{\mathrm{d}t}\le-\eta|S(X;t)|$ between $t = 0$ and $t = t_{\mathrm{reach}}$, we obtain the following estimate of t_{reach}:

$$t_{\mathrm{reach}}\le\frac{|S(X;0)|}{\eta}\tag{6.78}$$

The second component of the transient time is the time that it takes, while the system is already on the sliding surface, for the tracking error to tend exponentially to zero. It can be shown that such a time constant equals $(n-1)/\lambda$ and it is known that reaching close enough to a steady state may take $4\sim5$ time constants.

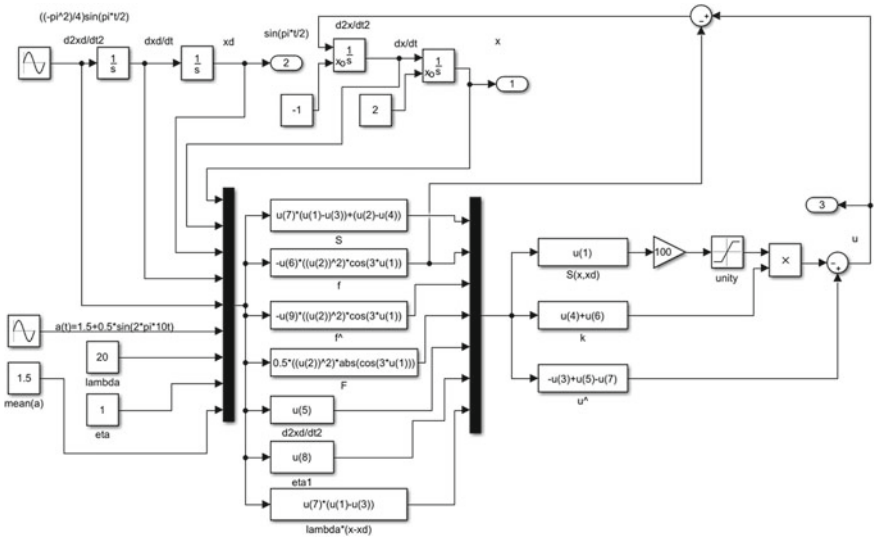


Fig. 6.114 Simulink diagram of SMC control of a second-order nonlinear and time-varying process

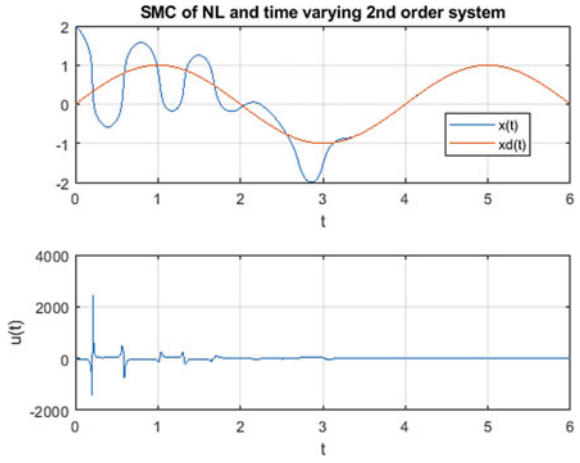


Fig. 6.115 Tracking performance for the illustrative example, featuring a long transient time

An additional interesting by-product of defining the sliding surface $S(X; t)$ to depend on a single parameter λ is a result (proved in [Slotine-Li]) that connects the width Φ of the high-gain amplifier's linear range (that replaces the three-position ideal relay) to upper bounds on the tracking quality, and the connection is made through the λ parameter.

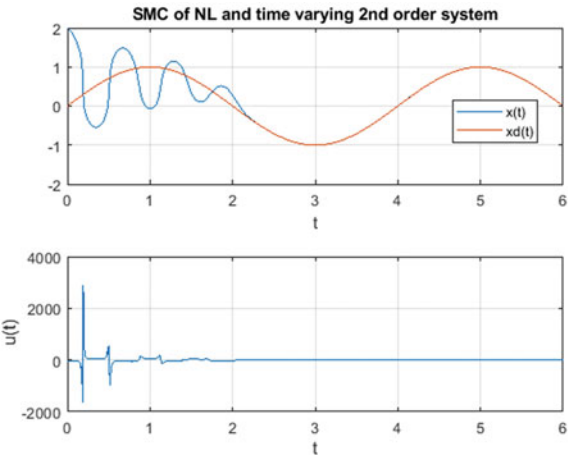


Fig. 6.116 A slightly improved transient time for a larger value of η

Result: How Suboptimal is the Tracking Performance Due to the Finite Linear Range of the High-Gain Saturating Amplifier that Replaces the Ideal Relay?

If $|S(X;t)| \leq \Phi$ for all $t \geq 0$, where, the linear range of the high-gain amplifier (that replaces the relay) goes between $-\Phi$ and $+\Phi$, then $\left|\frac{d^i \Delta x}{dt^i}(t)\right| \leq (2\lambda)^i \varepsilon \quad t \geq 0 \;\; i = 0, \ldots, n-1$ where $\varepsilon = \frac{\Phi}{\lambda^{n-1}}$.

The above result helps the control designer select the properties of the high-gain amplifier, depending on the desired tracking performance. The tradeoff is that the larger Φ, the larger the tracking error, but the smaller are the amplitude and oscillations of the control signal $u(t)$. This is the "smoothing" action caused by replacing the relay by a high-gain saturating amplifier.

So far every control system that was considered for SMC control had a purely additive control signal $u = u(X; t)$. That is, in the controlled process model, the input u had a unity coefficient. Slotine-Li's second generic example discusses SMC whenever the control signal u may have state-dependent, and possibly time-varying coefficient. We shall show the bottom line results skipping the proofs.

6.5.2.2 Slotine-Li's Second Generic Example [10]

Consider the process

$$\frac{d^2x}{dt^2} = f\left(x, \frac{dx}{dt}, t\right) + b\left(x, \frac{dx}{dt}, t\right)u \tag{6.79}$$

Both nonlinear and time-varying parts of the model, f and b, may have random or not fully known parts, of course, coming with some type of known bounds. In

addition, the coefficient b must satisfy one more condition—it must have a known sign. That is

$$0 < b_{\min} \le b \le b_{\max} \tag{6.80}$$

Each of the bounds $b_{\min}$ and $b_{\max}$ may be state-dependent. Let the best estimate of b, denoted as b_0, be obtained by taking a geometric mean of the upper and lower bounds of b:

$$b_0 = \sqrt{b_{\min} b_{\max}} \tag{6.81}$$

The estimated coefficient b_0 is in general state-dependent. As before, let us assume that f_0 is an estimate of f, and that F represents a model for the bounds on the deviations between f and f_0. As before, f_0 is constructed to bring $\mathrm{d}S(X; t)/\mathrm{d}t$ as close as possible to zero.

It is shown in [Slotine-Li] that the SMC control law now attains the following much more complicated form:

$$\begin{aligned} u &= b_0^{-1}(u_0 - k \cdot \mathrm{sign}(S)) \\ u_0 &= -f_0 + \frac{\mathrm{d}^2 x_d}{\mathrm{d}t^2} - \lambda \frac{\mathrm{d}\Delta x}{\mathrm{d}t} \\ k &\ge B(F + \eta) + (B - 1)|u_0| \\ B &= \sqrt{\frac{b_{\max}}{b_{\min}}} \quad F \ge |f - f_0| \end{aligned} \tag{6.82}$$

Using the above tools, it is now possible to address the simple motor speed control problem mentioned at the beginning of our discussion of the SMC methodology. In that problem, the motor approximated model was $\Omega(s)/\Omega_R(s) = K_m/(1 + s\tau_m)$ where both the motor's gain and time constant may be subject to some uncertainty with known bounds. The motor model can be written as $\mathrm{d}\Omega/\mathrm{d}t = f + bu$, where $f = -\Omega/\tau_m$ and $b = K_m/\tau_m$. A development of a SMC control for the motor speed control, following (6.82), is left as an easy exercise.

In some situations, the multi-relay SMC strategy that we have explored for first-order systems may be a more straightforward design tool. The idea is to "cover" every unknown coefficient with a simple relay. This strategy was explored by Sastry and Slotine [9]. Here is a summary of some of the key ideas.

Referring again to the definition of tracking error (6.64) and to the sliding surface definition (6.65), we recall that the discussion is limited to system in which the output signal and its first $n - 1$ derivatives qualify to be selected as the system state vector X. The nth-order nonlinear process model can be written as

$$\frac{\mathrm{d}^n x}{\mathrm{d}t^n} = f(X; t) + b(X; t)u \tag{6.83}$$

Again, the extent of the uncertainties in $f(X; t)$ must be modeled in terms of continuous functions of X. For $b(X; t)$, it should have the additional condition that $b(X; t)$ always has the same known sign.

The most general process models studied by Sastry and Slotine involved "square systems". These are systems consisting of m coupled differential equations (each with an order of n_j for $j = 1, \ldots, m$) and a model such as the one shown in (6.83), that each has its own additive control signal u_j. Such models may be common in robotics. In a motion control of an N degrees of freedom robot manipulator, the robot dynamic model may be described in terms of N coupled second-order dynamic equations. Such equations are typically highly nonlinear consisting of inertia terms (that correspond with second derivatives of the joint angles), centrifugal terms (that involve squares of joint angle derivatives), Coriolis terms (that involve products of joint velocities), and gravity terms (that just depend on the joint angles). In N degrees of freedom robot, it is common to find N parallel control systems, one for each robot joint. These N local controllers may each implement an SMC control for some sliding surface. The controllers communicate with a central control system that coordinates the overall SMC control effort. The following example is too simple to have come even from a two-link manipulator. It is a hypothetical example that just illustrates the concepts of multi-sliding-surfaces and multi-relays implementation of each.

6.5.2.3 Multi-sliding Surface SMC Example [9]

We consider the following "two-link manipulator" model, where the joint angles are denoted as θ_1 and θ_2. Each of the joints is to be controlled to track a parabolic signal. Some of coefficients are known exactly and for some there are known limits.

$$\begin{aligned}
&\frac{d^2\theta_1}{dt^2} = a_{11}\theta_1 + a_{12}\theta_2^2 + a_{13}\theta_1\frac{d\theta_2}{dt}\cos\theta_2 + u_1 \\
&\frac{d^2\theta_2}{dt^2} = a_{21}\theta_1^3 - a_{22}\theta_2\cos\theta_1 + u_2 \\
&x_{d1}(t) = \theta_1(t) = 2t^2 \quad x_{d2}(t) = \theta_2(t) = t^2 \quad \text{for} \quad t \geq 0 \\
&2 \leq a_{11} \leq 4 \quad a_{12} = 1 \quad a_{13} = 2 \quad a_{21} = 1 \quad -1.1 \leq a_{22} \leq -0.9
\end{aligned} \tag{6.84}$$

For each of the joints we define a sliding surface as follows:

$$\begin{aligned}
&\Theta_1 = \begin{pmatrix} \theta_1 \\ d\theta_1/dt \end{pmatrix} \quad \Theta_2 = \begin{pmatrix} \theta_2 \\ d\theta_2/Et \end{pmatrix} \\
&S_1(\Theta_1; t) = (\theta_1(t) - 2t^2) + (d\theta_1/dt - 4t) = 0 \\
&S_2(\Theta_2; t) = (\theta_2(t) - t^2) + (d\theta_2/dt - 2t) = 0
\end{aligned} \tag{6.85}$$

The development of SMC for each of the joint involves initially an allocation of a state-dependent gain to each term in the dynamic mode. Some of these gains end up being constant or mimics of the respective process nonlinearity. Other such gains may become relay actions. In each of the joints, there must be one relay control that is activated by the joint's sliding surface.

$$\begin{aligned} u_1 &= \beta_{11}\theta_1 + \beta_{12}\theta_2^2 + \beta_{13}\theta_1(\mathrm{d}\theta_2/\mathrm{d}t)\cos\theta_2 + k_{11}((\mathrm{d}\theta_1/\mathrm{d}t) - 4t) - k_{12}\mathrm{sign}(S_1) \\ u_2 &= \beta_{21}\theta_1^3 + \beta_{22}\theta_2 + k_{21}((\mathrm{d}\theta_2/\mathrm{d}t) - 2t) - k_{22}\mathrm{sign}(S_2) \end{aligned} \tag{6.86}$$

Let us now substitute u_1 of (6.86) into the derivative of S_1^2.

$$\begin{aligned} \frac{1}{2}\frac{\mathrm{d}S_1^2(\Theta_1;t)}{\mathrm{d}t} &= S_1\theta_1(\beta_{11} + a_{11}) + S_1\theta_2^2(\beta_{12} + a_{12}) \\ &\quad + S_1\theta_1(\mathrm{d}\theta_2/\mathrm{d}t)\cos\theta_2(\beta_{13} + a_{13}) \\ &\quad + S_1((\mathrm{d}\theta_1/\mathrm{d}t) - 4t)(k_{11} + 1) - k_{12}|S_1| - 4S_1 \end{aligned} \tag{6.87}$$

Because a_{12} and a_{13} are fixed, we simply choose $\beta_{12} = -a_{12} = -1$ and $\beta_{13} = -a_{13} = -2$ to nullify the second and third terms of the derivative of S_1^2. Because a_{11} is a random constant, lying between 2 and 4, we choose β_{11} to be a relay action activated by the sign of $S_1(\Theta_1; t)\theta_1$:

$$\begin{aligned} \beta_{11} &= -4 \quad \textit{if} \quad S_1(\Theta_1;t)\theta_1 > 0 \\ \beta_{11} &= -2 \quad \textit{if} \quad S_1(\Theta_1;t)\theta_1 < 0 \end{aligned} \tag{6.88}$$

The gain k_{11} is a relay action with small amplitude activated by the sign of $S_1((\mathrm{d}\theta_1/\mathrm{d}t) - 4t)$:

$$\begin{aligned} k_{11} &= -1.1 \quad \text{if } S_1((\mathrm{d}\theta_1/\mathrm{d}t) - 4t) > 0 \\ k_{11} &= -0.9 \quad \text{if } S_1((\mathrm{d}\theta_1/\mathrm{d}t) - 4t) < 0 \end{aligned} \tag{6.89}$$

Another way, probably better, is to simply choose $k_{11} = -1$ to cancel this relay altogether.

The gain k_{12} of the relay activated by the sign of S_1 creates symmetric levels determined by the maximum slope of $\theta_1(t)$ which is 4.

The control signal u_2 is designed in a similar manner. The two designs (of each local SMC control) are decoupled.

$$\begin{aligned} \frac{1}{2}\frac{\mathrm{d}S_2^2(\Theta_2;t)}{\mathrm{d}t} &= S_2\theta_1^3(\beta_{21} + a_{21}) + S_2\theta_2\cos\theta_1(\beta_{22} + a_{22}) \\ &\quad + S_2((\mathrm{d}\theta_2/\mathrm{d}t) - 2t)(k_{21} + 1) - k_{22}|S_2| - 2S_2 \end{aligned} \tag{6.90}$$

Because $a_{21} = 1$, we choose $\beta_{21} = -1$ to null the first term. Because of the randomness of a_{22}, the gain β_{22} represents a relay action activated by the sign of

$S_2\theta_2\cos\theta_1$. The relay levels are chosen similarly to (6.88), to (6.88), to be −1.1 when the sign of the activating signal is positive and −0.9 if the sign of the activating signal is negative. The gain k_{21} may be taken equal to −1 to null the third term. Finally, the relay levels k_{22} are chosen to be −2 and +2.

A simulation verification of this example is left as an exercise.

6.6 Control Design Based on Synthesis of Lyapunov Functions

Lyapunov stability is named after Aleksandr Mikhailovich Lyapunov, a Russian mathematician who defended the thesis *The General Problem of Stability of Motion* at Moscow University in 1892. A. M. Lyapunov was a pioneer in successful endeavoring to develop the global approach to the analysis of the stability of nonlinear dynamical systems by comparison with the widely spread local method of linearizing them about points of equilibrium. The mathematical theory of stability of motion, founded by A. M. Lyapunov, considerably anticipated the time for its implementation in science and technology.

6.6.1 Brief Review of Lyapunov Stability Theory

The basic idea of Lyapunov Stability Theory is that if one can show that some scalar generalized energy function of the system state becomes smaller and smaller as we move along any solution trajectory of the system, it indicates system stability in some sense. Furthermore, such calculation of the changes in the energy function does not require the knowledge of the system solution, only knowledge of the system model.

Example 6.6.1 (Illustration of Lyapunov Stability Theory Main Idea)

Let us consider the following second-order autonomous nonlinear system:

$$\begin{aligned}\frac{dx_1}{dt} &= -x_1 + x_2 \\ \frac{dx_2}{dt} &= -x_1 - x_2 - x_2^3\end{aligned} \tag{6.91}$$

The system is "autonomous" because it has no external input u. It is driven only by the initial conditions $x_1(0)$ and $x_2(0)$. We see that the system has only one equilibrium point $x_{1e} = 0$ and $x_{2e} = 0$.

Let us now consider the following function $V(x_1, x_2)$:

$$V(x_1, x_2) = x_1^2 + x_2^2 \tag{6.92}$$

We see that the arbitrary V is positive for all nonzero values of the state vector. We may consider it as some kind of a scalar energy function of the state. The larger the states the larger the system "energy", whereas at (0,0) the energy is zero. We could have chosen many other such V functions.

Let us now compute, using the differentiation chain rule, the rate of change of V as we move along the system's solution trajectories. This is done as follows:

$$\begin{aligned} \frac{\mathrm{d}V(x_1, x_2)}{\mathrm{d}t} &= \frac{\partial V}{\partial x_1}\frac{\mathrm{d}x_1}{\mathrm{d}t} + \frac{\partial V}{\partial x_2}\frac{\mathrm{d}x_2}{\mathrm{d}t} \\ &= 2x_1(-x_1 + x_2) + 2x_2(-x_1 - x_2 - x_2^3) \\ &= -2x_1^2 - 2x_2^2 - 2x_2^4 < 0 \end{aligned} \tag{6.93}$$

The computation of (6.93) required only the model (6.91) and the energy function (6.92). "To move along the solution" means that we choose values of the state that obey the system model. We see that the function V has a negative rate of change for any nonzero state vector. It is an indication that the system converges to (0,0) from any initial state.

Indeed the basic setup of Lyapunov's theory is the study of the stability of the zero state (0, …, 0) of an nth-order autonomous system.

$$X = \begin{pmatrix} x_1 \\ \vdots \\ x_n \end{pmatrix} \quad \frac{\mathrm{d}X}{\mathrm{d}t} = \begin{pmatrix} \mathrm{d}x_1/\mathrm{d}t \\ \vdots \\ \mathrm{d}x_n/\mathrm{d}t \end{pmatrix} = \begin{pmatrix} f_1(x_1, \ldots, x_n) \\ \vdots \\ f_n(x_1, \ldots, x_n) \end{pmatrix} \tag{6.94}$$

One may study the stability of any other (not necessarily zero) equilibrium state, by doing a simple change of variables of system (6.94). Therefore, there is no loss of generality discussing the zero state.

Lyapunov theory involves multiple definitions of stability and multiple types of the functions $V(X)$ to be candidates to become "Lyapunov Functions". Let us list them all, referring to [11], but then we shall focus on those definitions that are most relevant for practical control.

Definition 6.6.1 (Stability) The zero state is stable if for any arbitrary neighborhood of the zero state (no matter how small) there exist initial conditions that bring the system trajectories to this neighborhood.

In a stable equilibrium, there is no guarantee that the trajectories will actually converge to the zero state.

Definition 6.6.2 (Asymptotic Stability) The zero state is asymptotically stable if it is stable, and if there exist some neighborhood of the zero state such that from every initial state located within such neighborhood, the system trajectories converge to the zero state.

The set of all the initial states from which the system solutions actually converge to the zero state equilibrium is called *a region of attraction* of the zero state.

Definition 6.6.3 (Global Asymptotic Stability) The zero state is globally asymptotically stable if it is asymptotically stable and the region of attraction includes all possible initial states.

The "energy functions" $V(X)$ to be relevant for Lyapunov's stability theory are all continuously differentiable functions of the state elements $\{x_1, \ldots, x_n\}$, have zero value for the zero vector, and are nonnegative for any nonzero state vector. Let us list several such functions.

Definition 6.6.4 (Locally Positive Semi-Definite Function—l.p.s.d.f) A continuously differentiable function $V(X)$ from the n-dimensional vector space to the real numbers, such that $V(0) = 0$, is called l.p.s.d.f if and only if there exists a neighborhood of the zero state for which $V(X) \geq 0$ for any nonzero X that is contained in this neighborhood.

Positive semi-definiteness imply that the function may become equal to zero at states different than the zero state.

Definition 6.6.5 (Locally Positive Definite Function—l.p.d.f) A continuously differentiable function $V(X)$ from the n-dimensional vector space to the real numbers, such that $V(0) = 0$, is called l.p.d.f if and only if there exists a neighborhood of the zero state for which $V(X) > 0$ for any nonzero X that is contained in this neighborhood.

Definition 6.6.6 (Positive Definite Function—p.d.f) A continuously differentiable function $V(X)$ from the n-dimensional vector space to the real numbers, such that $V(0) = 0$, is called p.d.f if and only if $V(X) > 0$ for any nonzero X, and if it is radially unbounded. That is, as the state elements become larger, the function $V(X)$ becomes larger.

Example 6.6.2 (Illustration of Various Positive Definite and Similar Functions)

$V(x_1, x_2) = x_1^2 + x_2^2$ (used in Example 6.6.1) is a positive definite function.
$V(x_1, x_2) = x_1^2$ is a positive semi-definite function. It becomes zero for $x_1 = 0$ even if x_2 is nonzero.
$V(x_1, x_2) = (x_1 - x_2)^2$ is a positive semi-definite function. It becomes zero for any $x_1 = x_2$.
$V(x_1, x_2, x_3) = x_1^2 + x_2^2 + x_3^2\left(3 - x_3^2\right)$ is locally positive definite function. It is p.d.f only in the neighborhood {any x_1, any x_2, $|x_3| < \sqrt{3}$}.
$V(x_1, x_2, x_3) = x_1^2 + x_2^2 - x_3^2$ is an indefinite function.
$V(x_1, x_2) = x_1^2 + \left(1 - \exp\left(-x_2^2\right)\right)$ is only a locally positive definite function because it does not satisfy the radial unboundness property. For instance, if x_1 is kept finite and x_2 is driven to infinity, the value of V does not go to infinity.

The rate of change of a function $V(X)$ is computed as follows, simply extending the calculation done in (6.93) in Example 6.6.1:

$$\frac{\mathrm{d}V(X)}{\mathrm{d}t} = \frac{\partial V}{\partial x_1}\frac{\mathrm{d}x_1}{\mathrm{d}t} + \cdots + \frac{\partial V}{\partial x_n}\frac{\mathrm{d}x_n}{\mathrm{d}t} \tag{6.95}$$

We are ready now to list Lyapunov's stability theorems (without proof):

Theorem 6.6.1 (Lyapunov's Stability Theorem) The zero state of system (6.94) is stable if there exists a locally positive definite function $V(X)$ such that $-\mathrm{d}V(X)/\mathrm{d}t$ is locally positive semi-definite.

A function $V(X)$ that establishes stability (or other types of stability, as we shall see soon) is called a Lyapunov Function. Proving stability is done by guessing or by constructing more systematically a Lyapunov Function, which in general is not an easy task.

Let us illustrate Theorem 6.6.1 by going all the way back to the very first example of Chap. 6 (Example 6.1.1) of the rotating pendulum.

Example 6.6.3 (Stability of the Pendulum Zero State)
We recall the model of the frictionless pendulum

$$x_1 = \theta \quad x_2 = \frac{\mathrm{d}\theta}{\mathrm{d}t} \tag{6.4}$$

$$\frac{\mathrm{d}x_1}{\mathrm{d}t} = x_2 \quad \frac{\mathrm{d}x_2}{\mathrm{d}t} = -\frac{g}{L}\sin(x_1) \tag{6.5}$$

Let $V(x_1, x_2) = \frac{g}{L}(1 - \cos x_1) + \frac{x_2^2}{2}$. $V(X)$ is not exactly the energy of the system. The first term is related to the potential energy of the bob. The second term is related to the kinetic energy of the pendulum. $V(X)$ is indeed locally pdf for a neighborhood defined by $|x_1| < 2\pi$. We see that $\frac{\mathrm{d}V(x_1,x_2)}{\mathrm{d}t} = \frac{g}{L}(\sin x_1)x_2 - x_2\frac{g}{L}(\sin x_1) = 0$ showing that $-\mathrm{d}V(X)/\mathrm{d}t$ is positive semi-definite. Therefore the zero state (0,0) is stable. Recall the simulation results showing that the pendulum rotate around the low point at an amplitude that can be made very small by choosing the initial conditions appropriately.

Theorem 6.6.2 (Lyapunov's Asymptotic Stability Theorem) The zero state of system (6.94) is asymptotically stable if there exists a locally positive definite function $V(X)$ such that $-\mathrm{d}V(X)/\mathrm{d}t$ is locally positive definite.

Theorem 6.6.3 (Lyapunov's Global Asymptotic Stability Theorem) The zero state of system (6.94) is globally asymptotically stable if there exists a positive definite function $V(X)$ such that $-\mathrm{d}V(X)/\mathrm{d}t$ is positive definite.

Let us illustrate the two theorems above by means of simple second-order system examples:

Example 6.6.4 (Illustrating Lyapunov's Asymptotic Stability Theorem)
Let:

$$\begin{aligned}\frac{dx_1}{dt} &= x_1\left(x_1^2 + x_2^2 - 1\right) - x_2 \\ \frac{dx_2}{dt} &= x_1 + x_2\left(x_1^2 + x_2^2 - 1\right)\end{aligned} \tag{6.96}$$

Let $V(x_1, x_2) = x_1^2 + x_2^2$. We obtain

$$\frac{dV(X)}{dt} = 2\left(x_1^2 + x_2^2\right)\left(x_1^2 + x_2^2 - 1\right) \tag{6.97}$$

We see that $-dV(X)/dt$ is l.p.d.f for the neighborhood in the plane (x_1, x_2) defined by the unity circle. Therefore, we have shown using the above Lyapunov function that (0,0) is only locally asymptotically stable. We still do not know for sure if (0,0) may not be globally asymptotically stable, possibly by choosing another Lyapunov function. Simulation studies may show that indeed the zero state is only locally asymptotically stable.

Looking back at Example 6.6.1, we see that it fits the case of global asymptotic stability.

Let us conclude this introduction with mentioning several topics that way beyond the scope of this book. Finding a Lyapunov Function for a given system is like establishing a sufficient condition for stability. It is not a necessary and sufficient condition. For instance, an important result due to Lasalle establishes that in many cases of asymptotic stability, it is often too harsh to require that $-dV(X)/dt$ be pdf. In many cases, a psdf negative derivative will do.

An important application that requires extensively the use of Lyapunov Theory is that of the stability of interconnected power stations. A stable operation of such a grid involves that the stations maintain synchronous operation—equal frequency and small enough phase differences. Sudden load changes on some of the stations may cause loss of synchronism of some of the stations, which are considered to be an instability. Lyapunov Functions establish safe regions of attraction. Typically, each region of attraction is a very conservative estimate of the true region of attraction because any Lyapunov function is only a sufficient condition for stability. Engineers constantly try to find better and better Lyapunov Functions.

In most fields of engineering, "system design" (choosing the system parameters to meet certain design specifications) is harder to do than "system analysis" (understanding of the operation of a given system, possibly by means of simulation). Much analysis must be done prior to the design phase. In this sense, Lyapunov Theory is an anomaly—design is easier than analysis. We mentioned earlier that in general, it is hard to find a Lyapunov Function for a given system model. Many systematic construction techniques have been offered by researchers, and each is helpful for a limited number of cases.

On the other hand, in a control design based on Lyapunov Theory, we may be looking for a control feedback law in order to force certain $V(X)$ functions to become Lyapunov Functions. Often we do it by brute force, and it is straightforward. This is the topic of the next section. More often than not we strive to establish global asymptotic stability.

6.6.2 *Control Design Based on Synthesis of Lyapunov Functions*

Before we demonstrate how a Lyapunov Design is pursued, there is an analysis topic that is very relevant. Say that a Lyapunov function $V(X)$ is found for a given system such that the zero state is globally asymptotically stable. That is, $\mathrm{d}V(X)/\mathrm{d}t < 0$. It means that along any solution trajectory, the function $V(X; t)$ decreases as function of t, but at what rate? Does $V(X)$ carry some information that tells the control engineer if the convergence to zero is fast or slow?

A simple result from linear differential equation theory sets the analysis direction.

Result If $\frac{\mathrm{d}W(t)}{\mathrm{d}t} + \alpha W(t) \leq 0$ for all $t \geq 0$, then $W(t) \leq W(0)\mathrm{e}^{-\alpha t}$ for all $t \geq 0$. In other words, the solution of the linear first-order differential equation is bounded from above by an exponential function.

Let us consider the ratio between $\mathrm{d}V(X)/\mathrm{d}t$ and $V(X)$:

$$\begin{aligned} \frac{\mathrm{d}V(X;t)}{\mathrm{d}t} &= \frac{\frac{\mathrm{d}V(X;t)}{\mathrm{d}t}}{V(X;t)} V(X;t) \Rightarrow \frac{\mathrm{d}V(X;t)}{\mathrm{d}t} \\ &\leq \left\{ \max \left[\frac{\frac{\mathrm{d}V(X;t)}{\mathrm{d}t}}{V(X;t)} \right] \right\} V(X;t) \leq \rho V(X;t) \end{aligned} \tag{6.98}$$

That is, if we maximize $\{\mathrm{d}V(X)/\mathrm{d}t/V(X)\}$ over all X vectors in some neighborhood of the zero state, calling it ρ, then it tells us what is the exponentially decreasing function that bounds the convergence of $V(X; t)$ to zero:

$$\frac{\mathrm{d}V(X;t)}{\mathrm{d}t} \leq -\rho V(X;t) \Rightarrow V(X;t) \leq V(X;t_0)\mathrm{e}^{-\rho(t-t_0)} \tag{6.99}$$

Let us revisit Example 6.6.4 to demonstrate how it is done.

Example 6.6.4 Revisited

We recall that for $V(x_1, x_2) = x_1^2 + x_2^2$ we obtained

$$\frac{dV(X)}{dt} = 2\left(x_1^2 + x_2^2\right)\left(x_1^2 + x_2^2 - 1\right) \tag{6.97}$$

We observe in (6.97) that $dV(X)/dt = 2V(X)(V(X) - 1)$. That is

$$\begin{aligned} &\frac{dV(X)/dt}{V(X)} = 2(V(X) - 1) \\ &\left\{X : x_1^2 + x_2^2 \le 1 \Rightarrow \frac{dV(X)/dt}{V(X)} \le -2\right\} \end{aligned} \tag{6.100}$$

Therefore near the zero state, $V(X; t)$ convergence to zero is bounded by a time constant of $1/\rho = 0.5$.

The implication for Lyapunov Design is that if we try to maximize $-dV(X)/dt$ then the time response of the system is the fastest that can be obtained.

Example 6.6.5 (Demonstration of Lyapunov Design for Stabilization of a Tumbling Satellite)
Let X, Y and Z be the principal body axes of a satellite, and let ω_x, ω_y, and ω_z be the angular rates about each of these axes. We denote by I_x, I_y, and I_z the corresponding moments of inertia. The satellite is equipped with three thrusters, each can create a torque about one of the motion axes. In a stable satellite, the three angular rates are zero. However as the satellite is hit by random debris and by meteorites, it may begin to tumble. The purpose of the thrusters is to bring the angular rates back to zero whenever a tumbling motion starts. The motion equations of a satellite can be shown to be

$$\begin{aligned} I_x \frac{d\omega_x}{dt} - \left(I_y - I_z\right)\omega_y\omega_z &= u_1 \\ I_y \frac{d\omega_y}{dt} - (I_z - I_x)\omega_z\omega_x &= u_2 \\ I_z \frac{d\omega_z}{dt} - \left(I_x - I_y\right)\omega_x\omega_y &= u_3 \end{aligned} \tag{6.101}$$

We define the state vector X to be

$$X = \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} \omega_x \\ \omega_y \\ \omega_z \end{pmatrix} \tag{6.102}$$

The control objective is to develop feedback laws for the three control inputs u_1, u_2, and u_3 to assure that the zero state is globally asymptotically stable.

If we choose $V(X) = (I_x x_1)^2 + (I_y x_2)^2 + (I_z x_3)^2$, then

$$\frac{dV(X)}{dt} = 2\left(I_x x_1 u_1 + I_y x_2 u_2 + I_z x_3 u_3\right) \tag{6.103}$$

Now we need to choose each thruster control law to make $-\mathrm{d}V(X)/\mathrm{d}t$ positive definite. There are many feedback solutions $u_i = -f_i(x_i)$ for $i = 1, 2$, and 3 that can do the job. Each nonlinear function f_i should be a first and third quadrants function, as follows:

$$\begin{aligned} &x_i f_i(x_i) > 0 \quad \text{for} \quad x_i \neq 0 \\ &\lim_{x_i \to \infty} \int_0^{x_i} f_i(\sigma)\mathrm{d}\sigma = \infty \end{aligned} \tag{6.104}$$

We see that even a simple linear control law (a proportional control for each thruster) is a member of the radially unbounded family of nonlinear control laws shown in (6.104). Recalling (6.99), we may let each proportional control gain be as large as possible. Since each thruster has some maximum torque $|u_i| \leq U_i$, $i = 1, 2, 3$ that the thruster is capable of delivering, we choose

$$u_i = -U_i \mathrm{sign}(x_i) \tag{6.105}$$

This done in order to maximize $-\mathrm{d}V(X)/\mathrm{d}t$, resulting in a Bang–Bang control.

A Simulink diagram of the tumbling satellite in open loop is shown in Fig. 6.117, and the oscillatory angular rates are shown in Fig. 6.118. A Simulink diagram of the tumbling satellite in decentralized linear proportional control is shown in Fig. 6.119, along with Fig. 6.120 that shows how each angular velocity converges to zero.

The final two Figs. 6.121 and 6.122 demonstrate the bang–bang control of the tumbling satellite.

As shown in Margaliot and Langholz [12], in a Lyapunov Design, we very often pick some arbitrary but reasonable $V(X)$ function (such as the sum of squares of the state variables) and then in choosing a feedback control law, we try by brute force to make $-\mathrm{d}V(X)/\mathrm{d}t$ a positive definite function, very often in the presence of significant modeling uncertainties. This often leads to some kind of a "Rule-Based Control", such as fuzzy logic control.

Example 6.6.6 (Margaliot–Langholz Lyapunov Design of a Water Tank Level Control)

In a cylindrical water tank with base radius of r, let x be the amount of water. Let $q(t)$ be the inflow of water to the tank, assuming that $q(t)$ may vary around some nominal fixed flow q_s. At the bottom of the tank, there is an outflow drain. The amount of water outflowing from the tank is $p(x)u$, where $p(x) = a\sqrt{2g(x/\pi r^2)}$. The constant a is positive and g is the gravity acceleration. The control variable u is the cross-section area of the drain outlet.

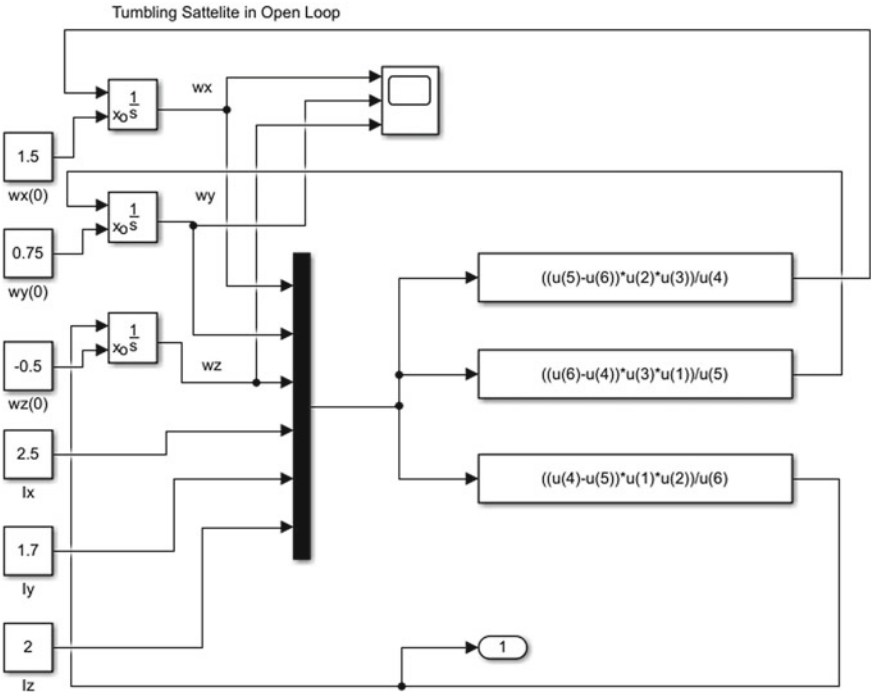


Fig. 6.117 Simulink of a tumbling satellite in open loop

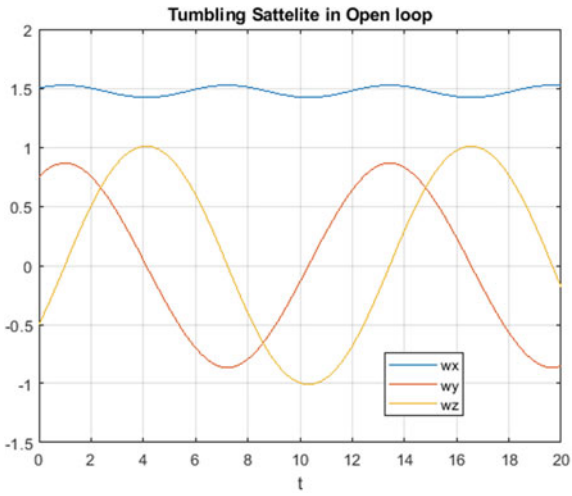


Fig. 6.118 Oscillatory angular rates in an open-loop tumbling satellite

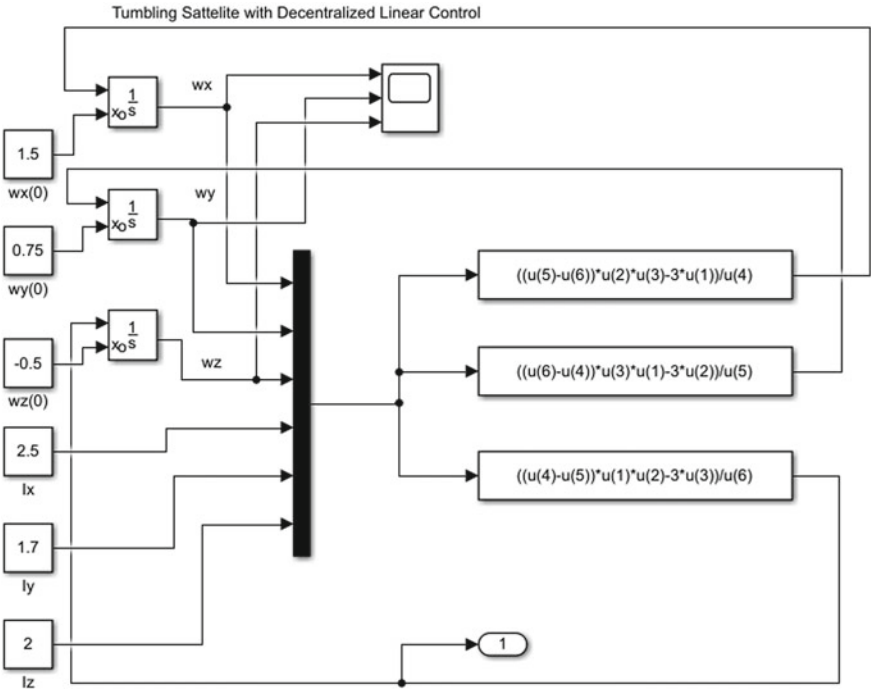


Fig. 6.119 Simulink diagram of a tumbling satellite in a decentralized proportional control

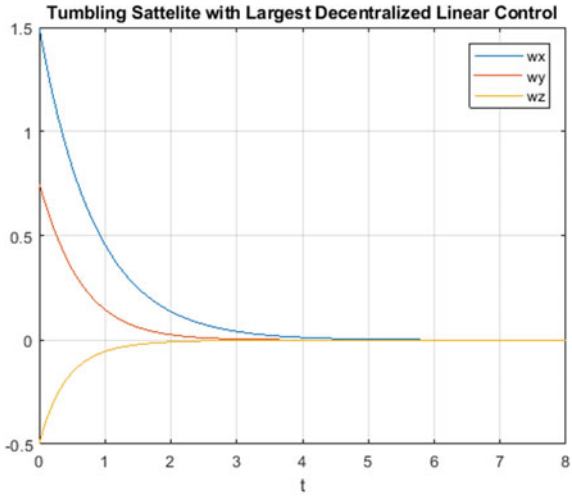


Fig. 6.120 Asymptotic stability of tumbling satellite in linear control

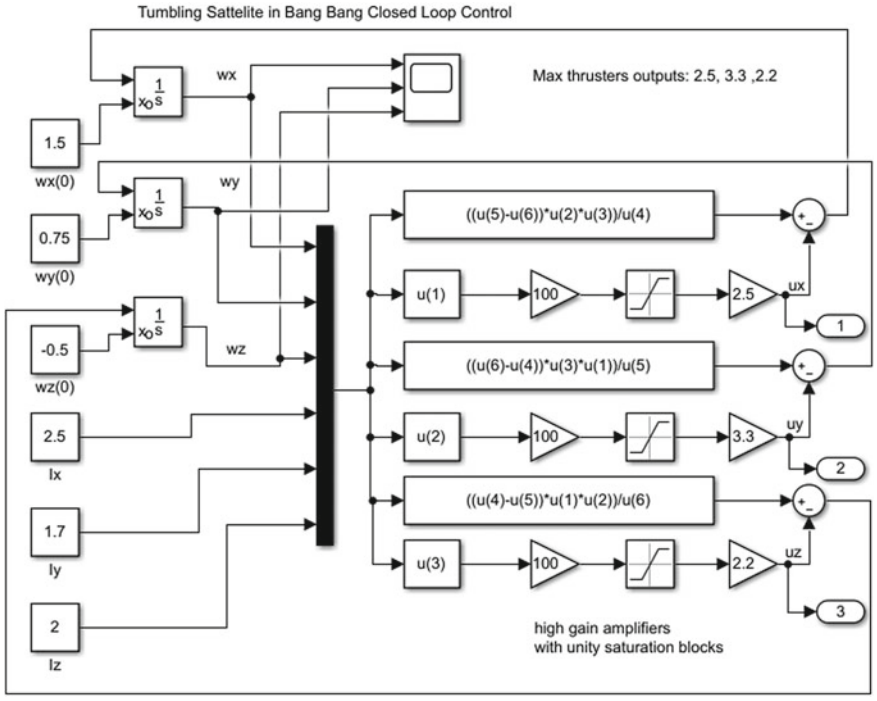


Fig. 6.121 Simulink diagram of a tumbling satellite in bang–bang control

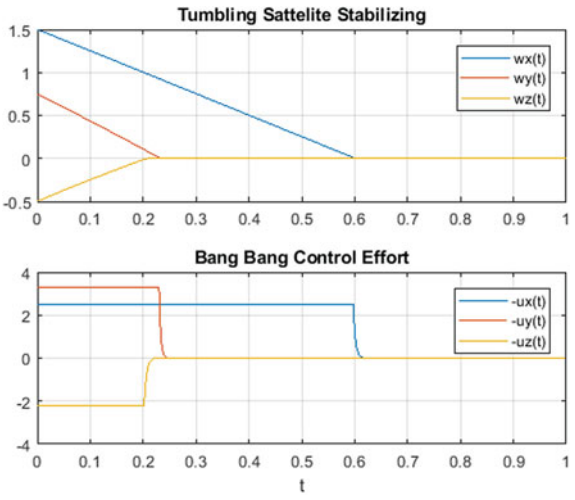


Fig. 6.122 Minimum time asymptotic stability convergence due to bang–bang control

The tank dynamic model is therefore

$$\frac{\mathrm{d}x}{\mathrm{d}t} = q(t) - p(x)u \tag{6.106}$$

The control objective is to design u to regulate $x(t)$ to a desired nominal amount x_s. It is assumed that $x(t)$ is measurable.

There is some modeling uncertainty: We do not have an explicit knowledge of $p(x)$. All we know is that $p(x)$ is positive but that the value $p_s = p(x_s)$ is known.

We let $V(x) = \frac{1}{2}(x - x_s)^2$ and its differentiation yields

$$\frac{\mathrm{d}V(x)}{\mathrm{d}t} = (x - x_s)\frac{\mathrm{d}x}{\mathrm{d}t} = (x - x_s)(q(t) - p(x)u) \tag{6.107}$$

Lyapunov Design means the choice of u in order to force $-\mathrm{d}V(x)/\mathrm{d}t$ to be positive for any nonzero value of x.

If x is low, we have that $x - x_s$ is negative, and because $q(t)$ is positive, we simply shut off the valve, taking $u = 0$.

If x is high, $x - x_s$ is positive, and therefore we must open the valve to a maximum: $u = u_{\max}$ to make $\mathrm{d}V(x)/\mathrm{d}t$ negative.

In [Margaliot–Langholz], a linear control law was designed for medium levels of x, choosing the gains to assure that $\mathrm{d}V(X)/\mathrm{d}t$ remains negative. Another option is to view the relay action, of levels 0 and $u_{\max}$ relating to the low x and high x cases, and then replace it with a high-gain saturating amplifier. The designer may experiment with several values of the linear region width in tuning the control signal u.

6.7 Summary

Very often, a control designer, when encountering nonlinearities, opts to linearize the model in order to be able to apply the vast arsenal of design methods available for linear models. A controller that is designed for a linearized model has limited validity. It is valid as long as the linear approximation is valid.

This chapter, on practical nonlinear control design, includes a variety of open-loop and closed-loop control design techniques that not only do not avoid the nonlinearities of the model but sometimes utilize the nonlinearities as part of a successful control strategy.

The chapter includes many MATLAB Simulink examples. One of the chapter purposes is to serve as a MATLAB Simulink tutorial.

Nonlinearities in the process model are often the "heart and soul" of the process model. We tried to demonstrate it with many real-life examples—aircraft motion models, missile guidance, physiological models, and more. In all these examples, the nonlinearities play a central role.

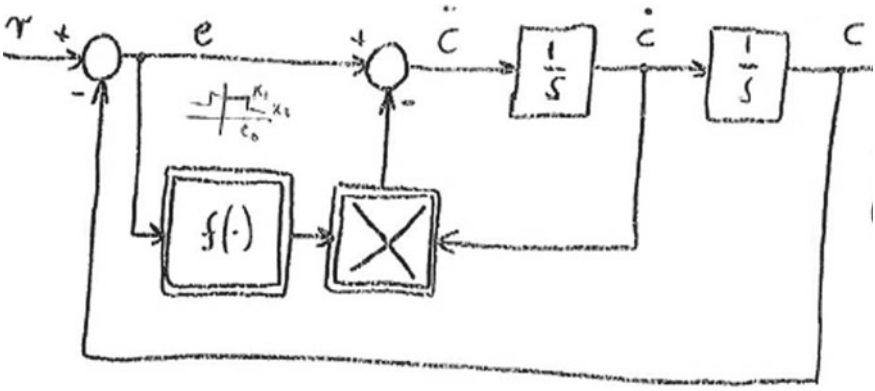
Dynamic simulations, if done carefully and systematically, can compensate for lack of full control design experience and training. A simulator in the loop that

performs multiple runs of a slow enough process can provide meaningful control insight that can be put into action in real time.

Sliding mode control and Lyapunov Design are powerful control design techniques in their own right. These control design methods particularly shine in the presence of modeling uncertainties and poorly modeled disturbance signals. One may even apply these methods to linear process in order to obtain superior performance at certain circumstances.

Due to time and space constraints, several subjects regrettably ended up being omitted from this chapter (awaiting possibly future editions of this book). These subjects include Model Linearization by means of MATLAB and Simulink [1] and PID Control Auto-Tuning. Earlier chapters of this book demonstrate the design of PID controllers when the process model is fully known (and linear). In industrial applications, the tuning of PID controllers is done in the presence of significant modeling uncertainties, and significant nonlinearities (such as state-dependent gains, time constants and time delay effects). The design of the PID parameters based on finite number of tests done to a "black box process" is a task that may often belong to the class of practical nonlinear techniques. Several notable PID tuning methods involve relays in the loop [13, 14].

HomeWork 6.1: The control system (below) is called Lewis' servomechanism. It consists of an analog multiplier that multiplies the signal $\dot{c}(t)$ by one of two possible gains: K_1 if $|e(t)|$ is smaller than a threshold e_0, or by K_2 (which is smaller than K_1) if the error is larger than the threshold e_0. The purpose of such nonlinear arrangement is to improve the system's response to step inputs $r(t) = r_0 = \text{constant}$ (for $t \geq 0$). The idea is that for large values of the error signal $e(t)$, the system's damping coefficient becomes small, yielding a fast initial response, but as the error becomes small enough, the damping coefficient becomes large, helping $e(t)$ to converge to zero with no overshoot and/or decaying oscillations. In this problem take: $K_1 = 3$, $K_2 = 0$ and $e_0 = 0.5$.



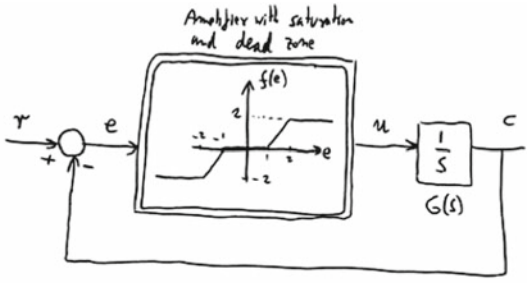
6.1a: Write the system's two sets of second-order differential equations, one for each region (small error and large error regions) in terms of e, $\dot{e}(t)$ and $\ddot{e}(t)$. Then write each equation in terms of the state variables: $x_1 = e$ and $x_2 = \dot{e}(t)$.

6.1b: Find the equilibrium points of each model (the one for small error and the one for large error) and indicate the stability and type of each equilibrium, as well as if the point is true or virtual equilibrium.

6.1c: Assume zero initial conditions for $c(t)$ and its derivative. Sketch one phase plane trajectories for $u_0 = 5$. Also, sketch the output signal time response $c(t)$.

HomeWork 6.2: The second-order process is driven by an amplifier that has both —saturation and dead zone regions. The input signal $r(t)$ is a step function. The initial conditions of $c(t)$ and its derivative are zero. For your convenience, the amplifier function $f(e)$ is

$$f(e)=\begin{cases} 2 & if \quad e \geq 2 \\ 2(e-1) & if \quad 1 \leq e \leq 2 \\ 0 & if \quad |e| \leq 1 \\ 2(e+1) & if \quad -2 \leq e \leq -1 \\ -2 & if \quad e \leq -2 \end{cases}$$



6.2a: Write the phase-variable model of the system by choosing the state to be $x = e$. That is, find $\dot{x}$ as function of x. Do it just for the first three regions (upper saturation, linear gain, and dead zone regions). [The other two provide symmetric information—not needed here].

6.2b: Analyze and sketch sample trajectories for the upper saturation region. Are there equilibrium points and if so what can you say about it?

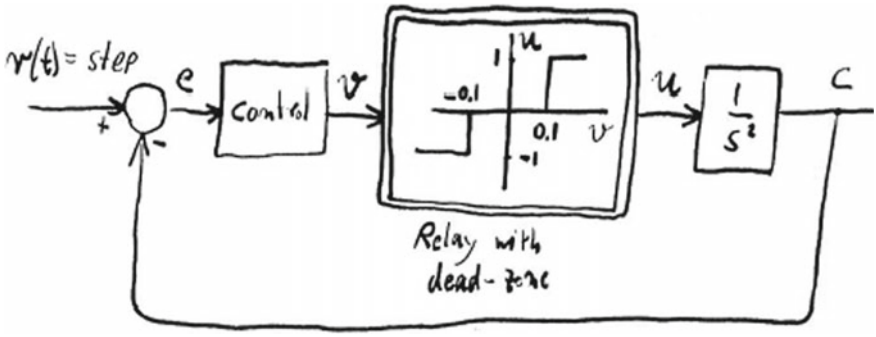
6.2c: Analyze and sketch sample trajectories for the upper linear gain region. Are there equilibrium points and if so what can you say about it?

6.2d: Analyze and sketch sample trajectories for the dead zone region. Are there equilibrium points and if so what can you say about it?

6.2e: Let the step input have a size of 5. Show the full phase plane trajectory for this input. Explain the features of this step response (response underdamped or overdamped? Steady-state error?)

HomeWork 6.3: Consider a relay with dead zone in a servo system. In this problem, the input $r(t)$ is a step of size 2. Assume zero initial conditions for $c(0)$ and $\dot{c}(0)$.

6.3a: With no control action (that is, v = e), explain and sketch the phase plane trajectories of the feedback system. Is it stable? *Hint*: Consider first the phase plane trajectories of the double integrator whenever its input is constant, and whenever it is zero.



6.3b: Rewrite the phase-variable model whenever the control is $v = e + \frac{de}{dt}$.

6.3c: What will be the phase plane trajectories of the system? Sketch approximately and explain.

6.3d: How many sliding regimes are there? Find each one.

6.3e: Does the system have steady-state error to this particular step input? If so, what is the error? Explain.

6.3f: Design the switching curve for a minimum-time relay control (known as "bang–bang control"). What is the optimal switching curve? Discuss it based on the known trajectories for each relay position.

6.3g: Does the system have steady-state error for the particular step input taken in this problem? Explain your answer.

HomeWork 6.4: Consider the following first-order time-varying nonlinear process: $\dot{x} = ax^3 + u$. That is, the system starts to run at t = 0, where $x(0)$ = 0. The coefficient a of x^3 is known to be between 2 and 4 (but the exact value is unknown). It is required that $x(t)$ performs perfect tracking of the input signal $x_d(t) = 3\sin(5t)$.

6.4a: Write an expression of the sliding surface. Also calculate the numerical value of $|\dot{x}_d|_{\text{MAX}}$.

6.4b: Perform a two-relay sliding mode control design (one relay to handle the uncertain model coefficient and one relay to maintain sliding conditions). Show mathematically what the control signal $u(t)$ should be.

6.4c: Draw a block diagram implementation for your design in (6.4b).

6.4d: Design a single-relay sliding mode controller, showing what the control signal u needs to be.

6.4e: Draw an implementation block diagram of your controller of (6.4d).

References

1. Dabney, J. B., & Harman, T. L. (2004). *Mastering SIMULINK*. Prentice-Hall.
2. Phillips, C. L., & Harbor, R. D. (2000). *Feedback control systems* (4th edn.). Prentice-Hall.
3. Schoen, E. *Selected Control Systems 1 Lecture Notes, 1973-1978, Technion, Israel Institute of Technology* (in Hebrew).
4. Murray, J. D. (2002). Mathematical biology: I. an introduction (3rd edn.). Springer.
5. Britton, N. F. (2003). *Essential mathematical biology*. London: Springer.
6. htpps://www.cdc.gov/vhf/ebola/outbreaks/2014-west-africa/index.html.
7. Khoo, M. C. K. (1999). *Physiological control systems: Analysis, simulation and estimation*. IEEE Press Series on Biomedical Engineering.
8. Truxal, J. G. (1955). *Automatic feedback control system synthesis.* McGraw-Hill.
9. Slotine, J. J., & Shankar Sastry, S. (1983). Tracking control of non-linear systems using sliding surfaces, with application to robot manipulators. *International Journal of Control, 38* (2), 465–492.
10. Slotine, J.-J. E., & Li, W. (1991). *Applied nonlinear control*. Prentice-Hall.
11. Khalil, H. K. (2015). *Nonlinear control*. Pearson Education.
12. Margaliot, M., & Langholz, G. (1999). Fuzzy Lyapunov-based approach to the design of fuzzy controllers. *Fuzzy Sets and Systems, 106,* 45–59.
13. Astrom, K. J., & Hagglund, T. (1984). Automatic tuning of simple regulator with specification on phase and amplitude margin. *Automatica, 20*(5), 645–651.
14. Voda, A., & Landau, I. D. (1995). A method for the auto-calibration of PID controllers. *Automatica, 31*(2).

Chapter 7
Fuzzy Logic Control Systems



In Chap. 5, we provided detailed discussions about the classical linear control systems or PID control systems with various design methods. One of the most important and critical key requirements to design and implement a PID control system is that the dynamic model of the process or plant is the prerequisite condition. However, in some actual applications, sometimes it is very difficult or impossible to obtain those models. Therefore, it looks like that it is impossible to design any appropriate controller for those control systems.

In order to solve this problem, the fuzzy logic control system is appeared, and it is more suitable to replace the PID controller to provide appropriate control role for those non-model systems. Another advantage of using Fuzzy Logic Controller (FLC) is that an FLC can perform control functions for both linear and nonlinear systems. Therefore, it is a robust controller.

To successfully design and apply a fuzzy logic controller to a control system whose model is not available, we should first answer some questions about this kind of controllers.

What is fuzzy logic? What is the meaning of the terminology fuzzy?

Generally speaking, fuzzy logic idea is similar to the human being's feeling and inference process. For example, in one winter day's morning after you get up, your mom told you: Dress more today and it is very cold! How cold is it? Your mom did not tell you the exact temperature such as 40 °F or 32 °F at all, but you have a 100% feeling and an idea about the meaning of "very cold" your mom meant. Yes, this is a fuzzy idea and it is the meaning of the terminology fuzzy.

Fuzzy idea and fuzzy logic are so often utilized in our routine life that nobody even pays attention to it. For instance, to answer some questions in certain surveys, most time one could answer with "Not Very Satisfied" or "Somehow Satisfied", which are also fuzzy or ambiguous answers. Exactly in which degree is one satisfied or dissatisfied with some service or product for those surveys? These vague answers can only be created and implemented by human beings, but not any machines. Is it possible for a computer to directly answer those survey questions as

© Springer Nature Switzerland AG 2019

Y. Bai and Z. S. Roth, *Classical and Modern Controls with Microcontrollers*,
Advances in Industrial Control, https://doi.org/10.1007/978-3-030-01382-0_7

human beings did? Absolutely it is impossible. Computers can only understand either "0" or "1", and "HIGH" or "LOW". Those data are called crisp or classic data and can be processed by all machines.

Is it possible to allow computers to handle those ambiguous data with the help of human being? If it is, how can computers and machines handle those vague data? The answer to the first question is yes. But to answer the second question, we need some fuzzy logic technique and knowledge of fuzzy inference system.

7.1 Introduction to Fuzzy Logic Control Technology

The idea of fuzzy logic was invented by Prof. L. A. Zadeh of the University of California at Berkeley in 1965. This invention was not well recognized until Dr. E. H. Mamdani, who is a professor at London University, applied the fuzzy logic in a practical application to control an automatic steam engine in 1974, which is almost 10 years after the fuzzy theory was invented. Then in 1976, Blue Circle Cement and SIRA in Denmark developed an industrial application to control cement kiln. That system began to operate in 1982. More and more fuzzy implementations have been reported since 1980s, including those applications in industrial manufacturing, automatic controls, automobiles productions, banks, hospitals, libraries, and academic educations. Fuzzy logic techniques have been widely applied in all aspects of our society today.

To implement fuzzy logic technique to a real application, one needs the following three operations or processing steps:

(1) Fuzzifications—Convert classical data or crisp data into the fuzzy data or Membership Functions (MFs),
(2) Fuzzy Inference Process—Combine membership functions with the control rules together to derive the fuzzy output, and
(3) Defuzzifications—Use different methods to calculate each associated output and put them into a table: Lookup Table. Pick up the output from the lookup table based on the current input during an application.

As mentioned before, all machines can process crisp or classical data such as either "0" or "1". In order to allow machines to handle the vague language input such as "Somehow Satisfied", the crisp input and output must be converted to linguistic variables with fuzzy components. For instance, to control an air conditioner system, the input temperature and the output control variables must be converted to the associated linguistic variables such as "HIGH", "MEDIUM", "LOW" and "FAST", "MEDIUM", or "SLOW". The former is corresponding to the input temperature and the latter is associated with the rotation speed of the operating motor. Besides those conversions, both the input and the output must also be converted from the crisp data to the fuzzy data. All of these jobs are performed by the first step—fuzzifications.

In the second step, to begin the fuzzy inference process, one needs to combine the membership functions with the control rules together to derive the control output and arrange those outputs into a table called lookup table. The control rule is the core of the fuzzy inference process, and those rules are directly related to the human beings' intuition and feeling. For example, still in the air conditioner control system, if the temperature is too high, the heater should be turn off, or the heat driving motor should be slow down, which is the human beings' intuition or common sense. Different methods such as Center-of-Gravity (COG) or Mean of Maximum (MOM) are utilized to calculate the associated control output, and each control output should be arranged into a table called lookup table.

During a real application, a control output should be selected from the lookup table developed from the last step based on the current input. Furthermore, that control output should be converted from the linguistic variable back to the crisp variable and output to the control operator. This process is called defuzzifications or step 3.

In most cases, the input variables are more than one dimension for real applications, one needs to perform fuzzifications to develop membership function for each dimensional variable separately. Perform the same operation if the system has multiple output variables.

Summarily, a fuzzy process is a process of crisp-fuzzy-crisp for a real system. The original input and the terminal output must be crisp variables, but the intermediate process is a fuzzy inference process. The reason why one needs to change the crisp to a fuzzy variable is: from the point of view of fuzzy control or human beings' intuition, no absolutely crisp variable is existed in our real world. Any physical variable may contain some other components. For instance, if someone says: the temperature here is high. This high temperature contains some middle and even low-temperature components. From this point of view, fuzzy control uses universal or global components, not just a limited range of components as the classical variables did.

As the rapid development of the fuzzy technologies, different fuzzy control strategies have been developed based on different classical control methods, such as the PID fuzzy control, sliding-mode fuzzy control, neural-fuzzy control, adaptor fuzzy control, and phase-plan mapping fuzzy control. More and more new fuzzy control strategies or combined crisp and fuzzy control techniques are being developed and will be applied to all aspects of our society in the future.

This chapter starts from an introduction to the fuzzy logic control and the fuzzy terminology. Following sections provide a detailed description of the fuzzy inference process and the architecture of the fuzzy logic control. Section 7.2 discusses the fuzzy sets and crisp sets. The fuzzifications and membership function are represented in Sect. 7.3. Fuzzy control rule and defuzzifications are discussed in Sects. 7.4 and 7.5, respectively. Some related combined fuzzy control methods are provided in the rest sections of this chapter. Section 7.6 provides a powerful tool, MATLAB Fuzzy Logic Toolbox, to help users to design and build professional FLC for most popular control systems. Another useful tool, MATLAB Simulink, is discussed in Sect. 7.7 to provide another easy way to assist users to build and tune the designed FLC to obtain the optimal control performances for given control systems.

7.2 Fuzzy Sets

The concept of the fuzzy set is only an extension of the concept of a classical or crisp set. Fuzzy set is actually a fundamentally broader set compared with the classical or crisp set. The classical set only considers a limited number of degrees of membership such as "0" or "1", or a range of data with limited degrees of membership. For instance, if a temperature is defined as a crisp high, its range must be between 80 °F and higher and it has nothing to do with 70 °F or even 60 °F. But fuzzy set will take care of a much broader range for this high temperature. In other words, fuzzy set will consider a much larger temperature range such as from 0 °F to higher degrees as a high temperature. The exact degree to which the 0 °F can contribute to that high temperature depends on the membership function. This means that the fuzzy set uses a universe of discourse as its base and it considers an infinite number of degrees of membership in a set. In this way, the classical or crisp set can be considered as a subset of the fuzzy set.

7.2.1 Classical Sets and Operations

A classical set is a collection of objects in a given range with a sharp boundary. An object can either belong to the set or not belong to the set. For example, we assume to create a faculty set or a faculty collection F with ten faculty members x_1, x_2 … x_{10}, in a college.

$$F = \{x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8, x_9, x_{10}\} \tag{7.1}$$

In general, the entire object of discussion F is called a universe of discourse, and each member x_i is called an element. Assuming that elements $x_1 \sim x_4$ belong to the department of computer science, which can be considered as another set A. The elements $x_1 \sim x_3$ are under age 40, which can be considered as a set B. Therefore, the following relations are existed:

$$X \in F, \quad A = \{x_1, x_2, x_3, x_4\} \in F, \quad B = \{x_1, x_2, x_3\} \subset A \tag{7.2}$$

It can be found that all elements in set B belong to set A, or the set A contains set B. In this case, set B can be considered as a subset of set A and can be expressed as $B \subset A$.

The relationship among different sets we discussed above can be described in Fig. 7.1. The basic classical set operations include complement, intersection, and union, which are represented as

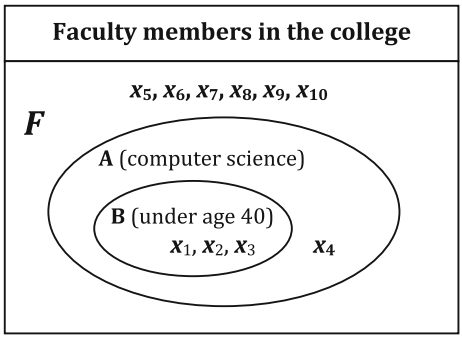


Fig. 7.1 The classical set of faculty members in a college

Complement of A (A^C)

$$A^C(x) = 1 - A \tag{7.3}$$

Intersection of A and B ($A \cap B$)

$$A \cap B = A(x) \cap B(x) \tag{7.4}$$

Union of A and B ($A \cup B$)

$$A \cup B = A(x) \cup B(x) \tag{7.5}$$

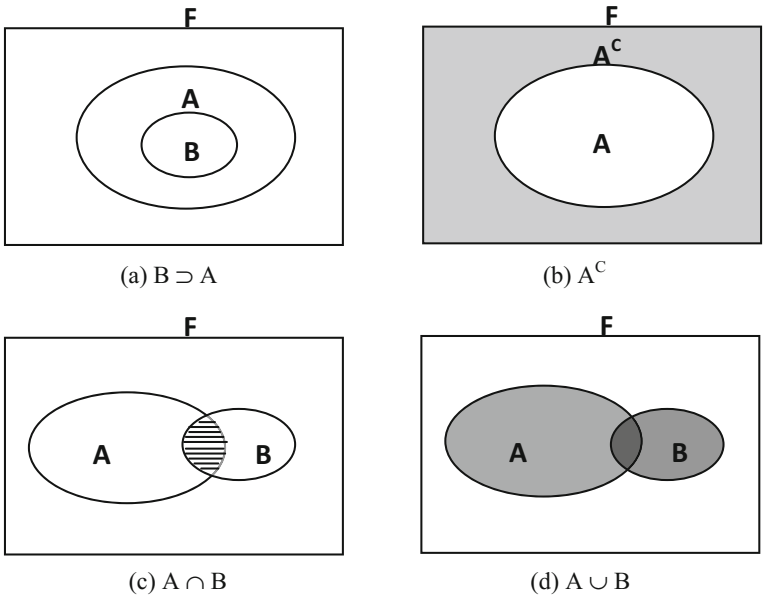
The representations of those classical set operations are shown in Fig. 7.2.

It is clear that either an element belongs to a set or does not belong to that set in the classical set and its operation. There is a sharp boundary between different elements for different sets and cannot be mixed for each other. But for the fuzzy set, it has different laws.

7.2.2 Mapping of Classical Sets to Functions

The classical set we discussed in the last section can be represented or mapped into some functions. This means that we can relate set-theoretic forms to function-theoretic representations, and map elements on one universe of discourse to elements or sets in another universe. For the classical or crisp sets, this mapping is very easy and straightforward. Assume that X and Y are two different universes of discourse. If an element x belongs to X and it corresponds to an element y belonging to Y, the mapping between them can be expressed as

$$\mu_A(x) = \begin{cases} 1, & x \in A \\ 0, & x \notin A \end{cases} \tag{7.6}$$



(a) B ⊃ A (b) A^C

(c) A ∩ B (d) A ∪ B

Fig. 7.2 Classical set and its operations

where μ_A represents "membership" in a set A for the element x in the universe. This membership idea is exactly a mapping from the element x in the universe X to one of two elements in universe Y, or to elements 0 or 1.

Still by using the example in the last section, a set of faculty members in a college contains two subsets A and B. We want to define a new set $P(x)$ as

$$P(x) = \{A, B\} \tag{7.7}$$

and mapping these two sets from universe of discourse F to another universe Y with two elements, 0 or 1.

$$M\{P(x)\} = \{(1,1,1,1,0,0,0,0,0,0), (1,1,1,0,0,0,0,0,0,0)\} \tag{7.8}$$

Now we can define operations of two classical sets, says A and B, on the universe F as follows:

$$\begin{array}{ll} \text{Union:} & A \cup B = \mu_A(x) \cup \mu_B(x) = \max(\mu_A(x), \mu_B(x)) \\ \text{Intersection:} & A \cap B = \mu_A(x) \cap \mu_B(x) = \min(\mu_A(x), \mu_B(x)) \\ \text{Complement:} & A^C = F \backslash A \end{array} \tag{7.9}$$

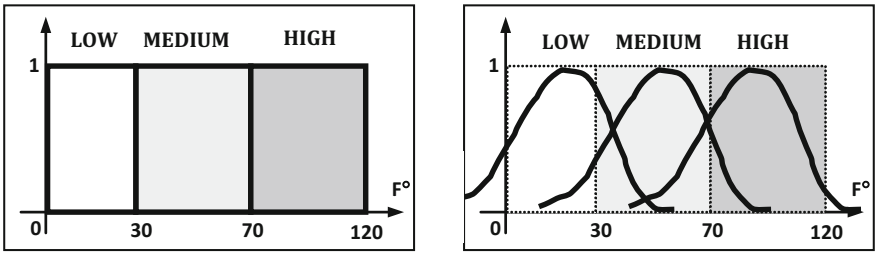
7.2.3 *Fuzzy Sets and Operations*

As we discussed in the previous section, the classical set has a sharp boundary, which means that a member either belongs to that set or not. Also, this classical set can be mapped to a function with two elements, 0 or 1. For example, in the previous section, we defined the faculty member in the department of computer science as set A. A faculty either fully belongs to this set ($\mu_A(x) = 1$) if one is a faculty of the computer science department or has nothing to do with set A ($\mu_A(x) = 0$) if he is not a faculty in that department. This mapping is straightforward with sharp boundary without any ambiguity. In other words, this fully belonging to can be mapped as a member of set A with degree of 1, and not belonging to can be mapped as a member of set A with degree of 0. This mapping is similar to a black-and-white binary categorization.

Compared with classical set, fuzzy set allows members to have a smooth boundary. In other words, fuzzy set allows a member to belong to a set with some partial degree. For instance, still we use the temperature as an example. The temperature can be divided into three categories: LOW (0 ~ 30 °F), MEDIUM (30 ~ 70 °F) and HIGH (70 ~ 120 °F) from the point of view of the classical set, which is shown in Fig. 7.3a.

In the classical set, any temperature can only be categorized into one subset, LOW, MEDIUM or HIGH, and the boundary is crystal clear. But in the fuzzy set such as shown in Fig. 7.3b, these boundaries become vague or smooth. One temperature can be categorized into two or maybe even three subsets simultaneously. For example, temperature 40 °F can be considered to belong to LOW in certain degree, say 0.5°, but at the same time, it can belong to MEDIUM with about 0.7°. Another interesting thing is the temperature 50 °F, which can be considered to belong to LOW and HIGH at around 0.2° and belong to MEDIUM at almost 1°. The dashed line in Fig. 7.3b represents the classical set boundary.

It is clear that a fuzzy set contains elements which have varying degrees of membership in the set, and this is contrasted with the classical or crisp sets because members of a classical set cannot be members unless their membership is full or complete in that set. A fuzzy set allows a member to have partial degree of



(a) Classical set (b) Fuzzy set

Fig. 7.3 The representations of the classical and fuzzy sets

membership and this partial degree membership can be mapped into a function or a universe of membership values. Assume that we have a fuzzy set A, and if an element x is a member of this fuzzy set A, this mapping can be denoted as

$$\mu_A(x) \in [0,1] \quad (A = (x, \mu_A(x)|x \in X) \tag{7.10}$$

A fuzzy subset A with an element x has a membership function of $\mu_A(x)$. When the universe of discourse X is discrete and finite, this mapping can be expressed as

$$A = \frac{\mu_A(x_1)}{x_1} + \frac{\mu_A(x_2)}{x_2} + \cdots = \sum_i \frac{\mu_A(x_i)}{x_i} \tag{7.11}$$

When the universe X is continuous and infinite, the fuzzy set A can be represented as

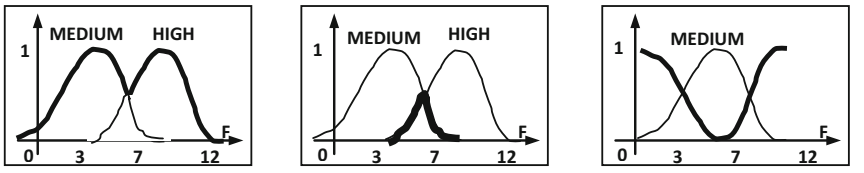
$$A = \int \frac{\mu_A(x)}{x} \tag{7.12}$$

As we discussed in the previous section for the classical set operations, the basic fuzzy set operations also include intersection, union, and complement, and those operations are defined as

$$\begin{array}{ll} \text{Union:} & A \cup B = \mu_A(x) \cup \mu_B(x) = \max(\mu_A(x), \mu_B(x)) \\ \text{Intersection:} & A \cap B = \mu_A(x) \cap \mu_B(x) = \min(\mu_A(x), \mu_B(x)) \\ \text{Complement:} & A^C = F \backslash A \end{array} \tag{7.13}$$

where A and B are two fuzzy sets and x is an element in the universe of discourse, X. The graphical illustrations of those operations are shown in Fig. 7.4.

According to the definition of the fuzzy set operations, a fuzzy set union operation is exactly equivalent to select the maximum member from those members in the sets, and the intersection operation is to select the minimum member from the sets.



(a) Fuzzy set union (b) Fuzzy set intersection (c) Fuzzy set complement

Fig. 7.4 Fuzzy set operations

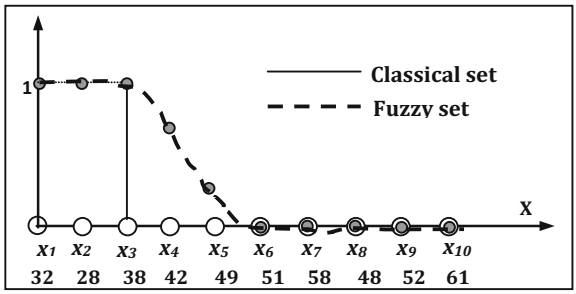


Fig. 7.5 The comparison between the classical and fuzzy sets

7.2.4 The Comparison Between the Classical and Fuzzy Sets

Recall that in Sect. 7.2.1, we defined a college faculty set ***F***, a faculty set for the computer science department, set ***A***, and a set of faculty members who are under age 40, or set ***B***. Now we can make a clear comparison for the classical and fuzzy sets using this example.

According to the classical or crisp set theory, set ***B*** includes only three members who are under age 40 (x_1, x_2, x_3), so the boundary between the faculty who are under age 40 and the faculty who are above age 40 is crystal clear (see solid line in Fig. 7.5). But for the fuzzy set theory, set ***B*** contains not only those three members, x_1, x_2, x_3, but also contains some other members in some varying degree. The actual age distribution of all 10 faculty members is displayed in the x-axis in Fig. 7.5.

It can be found from Fig. 7.5 that members x_4 and x_5 are not belong to the set ***B*** in the view of crisp set, but they can be considered as a partial member in a partial membership of 0.7° and 0.3°, respectively. To map this to a membership function, it can be expressed as

$$\mu_B(x) = \frac{1}{x_1} + \frac{1}{x_2} + \frac{1}{x_3} + \frac{0.7}{x_4} + \frac{0.3}{x_5}$$

where the divider operator "/" is called a separator, and "+" is the *or* operator. All other members whose membership is 0 are omitted from this function.

7.3 Fuzzifications and Membership Functions

Fuzzy set is a powerful tool and allows us to represent objects or members in a vague or ambiguous way. Fuzzy set also provides a way that is similar to the human beings' concepts and think process. But only fuzzy set itself cannot lead to any useful and practical products until the fuzzy inference process is applied. To implement fuzzy inference to a real product or to solve an actual problem, as we discussed before, three consecutive steps are needed, which are: Fuzzification, fuzzy inference, and defuzzification.

Fuzzification is the first step to apply a fuzzy inference system. Most variables existed in the real world are crisp or classical variables. One needs to convert those crisp variables (both input and output) to the fuzzy variables, and then apply fuzzy inference to process those data to obtain the desired output. Finally, in most cases, those fuzzy outputs need to be converted back to crisp variables to complete the desired control objectives.

Generally, fuzzification contains two processes: derive the membership functions for input and output variables and represent them with linguistic variables. This process is equivalent to convert or map classical set to the fuzzy set with varying degrees.

In practical, membership functions can have multiple different types, such as the triangular waveform, trapezoidal waveform, Gaussian waveform, bell-shaped waveform, sigmoidal waveform, and *S*-curve waveform. The exact type depends on the actual applications. For those systems that need significant dynamic variation in a short period of time, a triangular or a trapezoidal waveform should be utilized. For those systems that need very high control accuracy, a Gaussian or *S*-curve waveform should be selected.

To illustrate the process of the fuzzifications, we still use the air conditioner example we developed in the previous section. Assume that we have an air conditioner control system that is under the control of only a heater. If the temperature is high, the heater control motor should be turn-off, and if the temperature is low, that heater motor should be turn-on, which is the common sense.

Regularly, the normal temperature range is from 20 ~ 90 °F. This range can be further divided into three subranges or subsets, which are:

Low temperature: 20 ~ 40 °F, 30 °F is center
Medium temperature: 30 ~ 80 °F, 55 °F is center
High temperature: 60 ~ 90 °F, 75 °F is center

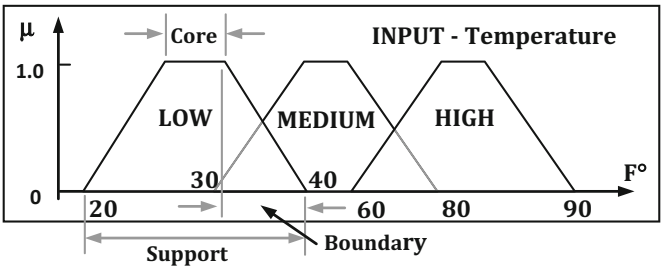
Next, those three ranges need to be converted to linguistic variables: LOW, MEDIUM, and HIGH, which are corresponding to three temperature ranges listed above.

The membership function of these temperatures is shown in Fig. 7.6. To make thing simple, a trapezoidal waveform is utilized for the type of the membership function. A crisp low temperature can be considered as a medium temperature with some degree in this fuzzy membership function representation. For instance, a 35 °F will belong to a LOW and MEDIUM in 0.5°. Some terminologies used for the membership function are also shown in Fig. 7.6.

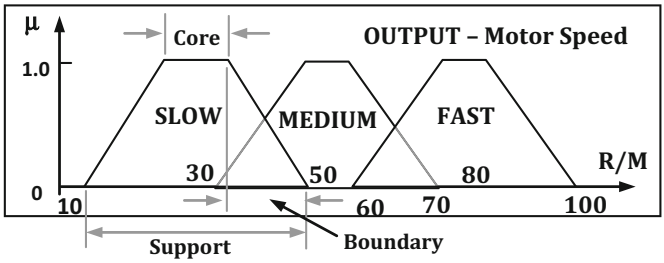
The support of a fuzzy set, says LOW, is the set of elements whose degree of membership in LOW is greater than 0. This support can be expressed in a functional form as

$$\text{Support (LOW)} = \{x \in T | \mu_{\text{LOW}}(x) > 0\} \tag{7.14}$$

It can be found that the support of a fuzzy set is a classical set.



(a) Membership function of the input.



(b) Membership function of the output.

Fig. 7.6 Membership functions of input and output

The core of a fuzzy set is the set of elements whose degree of membership in that set is equal to 1, which is equivalent to a crisp set. The boundary of a fuzzy set indicates the range in which all elements whose degree of membership in that set is between 0 and 1 (0 and 1 are excluded).

After the membership functions are defined for both input and output, the next step is to define the fuzzy control rule.

7.4 Fuzzy Control Rules

Fuzzy control rule can be considered as the knowledge of an expert in any related field of application. The fuzzy rule is represented by a sequence of the form IF-THEN, leading to algorithms describing what action or output should be taken in terms of the currently observed information, which includes both input and feedback if a closed-loop control system is applied. The law to design or to build a set of fuzzy rules is based on the human beings' knowledge or experience, which is dependent on each different actual application.

A fuzzy IF-THEN rule associates a condition described using linguistic variables and fuzzy sets to an output or a conclusion. The IF part is mainly used to capture knowledge by using the elastic conditions, but the THEN part can be utilized to

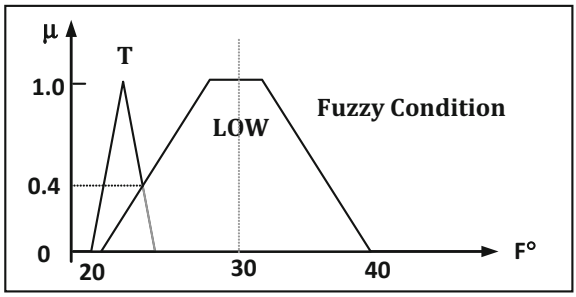


Fig. 7.7 Match a fuzzy input with a fuzzy condition

give the conclusion or output in the linguistic variable form. This IF-THEN rule is widely used by the fuzzy inference system to compute the degree the input data matches the condition of a rule. Figure 7.7 illustrates a way to calculate the degree between a fuzzy input T (temperature) and a fuzzy condition LOW. Here we still use the air conditioner system as an example.

This calculation can also be represented by the function

$$M(T, \text{LOW}) = \text{Support } \min(\mu_T(x), \mu_{\text{LOW}}(x)) \tag{7.15}$$

Two types of fuzzy control rules are widely utilized for most real applications. One is fuzzy mapping rules and the other is called fuzzy implication rules.

7.4.1 Fuzzy Mapping Rules

Fuzzy mapping rule provides a functional mapping between the input and the output using linguistic variables. The foundation of a fuzzy mapping rule is a fuzzy graph, which describes the relationship between the fuzzy input and the fuzzy output. Sometimes in real applications, it is very hard to derive a certain relationship between the input and the output, or the relationship between those inputs and outputs are very complicated even that relationship is developed. Fuzzy mapping rule is a good solution for those situations.

Fuzzy mapping rule works a similar way as human intuitions or insights, and each fuzzy mapping rule only approximates a limited number of elements of the function, so the entire function should be approximated by a set of fuzzy mapping rules. Still use our air conditioner system as an example. A fuzzy mapping rule can be derived as

IF the temperature is **LOW**, THEN the heater motor should be rotated **FAST**.

For other input temperatures, different rules should be developed.

For most actual applications, the input variables are commonly more than one dimension. For example, in our air conditioner system, the inputs include both current temperature and the change rate of the temperature. The fuzzy control rules

Table 7.1 An example of fuzzy rules

$\dot{T}$	T		
	LOW	MEDIUM	HIGH
LOW	FAST	MEDIUM	MEDIUM
MEDIUM	FAST	SLOW	SLOW
HIGH	MEDIUM	SLOW	SLOW

should also be extended to allow multiple inputs to be considered to derive the output. Table 7.1 is an example of fuzzy control rules applied in our air conditioner system.

The row and column represent two inputs, the temperature input and the change rate of the temperature input, and those inputs are related to IF parts in IF-THEN rules. The conclusion or control output can be considered as a third-dimensional variable that is located at the cross point of each row (temperature) and each column (change rate of the temperature), and that conclusion is associated with the THEN part in IF-THEN rules. For example, when the current temperature is LOW, and the current change rate of the temperature is also LOW, the heater motor's speed should be FAST to increase the temperature as soon as possible. This can be represented by the IF-THEN rule as

IF the temperature is LOW, and the change rate of the temperature is LOW,
THEN the conclusion or output (heater motor speed) should be FAST

All other rules follow the similar strategy, which is very similar to human beings' intuition.

In this air conditioner example, totally nine rules are developed. For those applications that need high control accuracy, the input and output should be divided into more small segments, and more fuzzy rules should be applied.

7.4.2 Fuzzy Implication Rules

Fuzzy implication rule describes a generalized logic implication relationship between inputs and outputs. The foundation of a fuzzy implication rule is the narrow sense of fuzzy logic. The fuzzy implication rules are related to classical two-valued logic and multiple-valued logic.

Still using the air conditioner system as an example, the implication is:

IF the temperature is LOW, THEN the heater motor should be FAST.

Based on this implication and a fact, as the temperature is HIGH, the result that the heater motor should be slow down or SLOW can be inferred.

7.5 Defuzzifications and Lookup Table

The conclusion or control output derived from the combination of input, output membership functions, and fuzzy rules is still a vague or fuzzy element and this process is called fuzzy inference. To make that conclusion or fuzzy output available to the real applications, a defuzzification process is needed. The defuzzification process is exactly to convert the fuzzy output back to the crisp or classical output for the control objective. Remember, the fuzzy conclusion or output is still a linguistic variable, and this linguistic variable needs to be converted to the crisp variable via the defuzzification process. Three defuzzification techniques are commonly used: mean of maximum method, center-of-gravity method, and the height method.

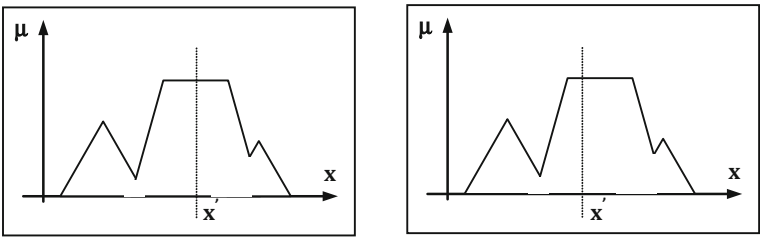
7.5.1 Mean of Maximum (MOM) Method

The Mean of Maximum (MOM) defuzzification method computes the average of those fuzzy conclusions or outputs that have the highest degrees. For example, the fuzzy conclusion is: the heater motor x is rotated FAST. By using the MOM method, this defuzzification can be expressed as

$$\text{MOM(FAST)} = \frac{\sum_{x' \in T} x'}{T} \quad T = \{x' | \mu_{\text{FAST}}(x') = \text{Support}\, \mu_{\text{FAST}}(x)\} \tag{7.16}$$

where T is the set of output x who has the highest degrees in the set FAST. A graphic representation of the MOM method is shown in Fig. 7.8a.

A shortcoming of the MOM method is that it does not consider the entire shape of the output membership function, and it only takes care of the points that have the highest degrees in that function. For those membership functions that have different shapes but same highest degrees, this method will get the same result.



(a) MOM method example (b) COG method example

Fig. 7.8 Graphic representations of defuzzification techniques

7.5.2 Center-of-Gravity (COG) Method

The Center-of-Gravity method (COG) is the most popular defuzzification technique and widely utilized in actual applications. This method is similar to the formula for calculating the center of gravity in physics. The weighted average of the membership function or the center of the gravity of the area bounded by the membership function curve is computed to be the most-crisp value of the fuzzy quantity. For example, for the conclusion: the heater motor x is rotated FAST. The COG output can be represented as

$$\mathrm{COG}(\mathrm{FAST}) = \frac{\sum_x \mu_{\mathrm{FAST}}(x) \times x}{\sum_x \mu_{\mathrm{FAST}}(x)} \tag{7.17}$$

If x is continuous variable, this defuzzification result is

$$\mathrm{COG}(\mathrm{FAST}) = \frac{\int \mu_{\mathrm{FAST}}(x) x \mathrm{d}x}{\int \mu_{\mathrm{FAST}}(x) \mathrm{d}x} \tag{7.18}$$

A graphic representation of COG method is shown in Fig. 7.8b.

7.5.3 The Height Method (HM)

This defuzzification method is valid only for the case where the output membership function is an aggregated union result of symmetrical functions. This method can be divided into two steps. First, the consequent membership function F_i can be converted into crisp consequent $x = f_i$ where f_i is the center of gravity of F_i. Then the COG method is applied to the rules with crisp consequents, which can be expressed as

$$x = \frac{\sum_{i=1}^{M} w_i f_i}{\sum_{i=1}^{M} w_i} \tag{7.19}$$

where w_i is the degree to which the ith rule matches the input data. The advantage of this method is its simplicity. Therefore, many neural-fuzzy models use this defuzzification method to reduce the complexity of calculations.

7.5.4 The Lookup Table

The terminal product of the defuzzification is the lookup table. The defuzzification needs to be performed for each subset of membership function, both inputs and

outputs. For instance, in the air conditioner system, one needs to perform the defuzzification for each subset of temperature input such as LOW, MEDIUM, and HIGH based on the associated fuzzy rules. The defuzzification result for each subset needs to be stored in the associated location in the lookup table according to the current temperature and temperature change rate. Following we use the air conditioner system as an example to illustrate the defuzzification process and the creation of the lookup table.

To make this illustration simple, we make two assumptions: (1) Assume that the membership function of the change rate of the temperature can be described as one shown in Fig. 7.9. (2) Only four rules are applied to this air conditioner system, which are:

(1) IF the temperature is LOW, and the change rate of the temperature is LOW, THEN the heater motor speed should be FAST.
(2) IF the temperature is MEDIUM, and the change rate of the temperature is MEDIUM, THEN the heater motor speed should be SLOW.
(3) IF the temperature is LOW, and the change rate of the temperature is MEDIUM, THEN the heater motor speed should be FAST.
(4) IF the temperature is MEDIUM, and the change rate of the temperature is LOW, THEN the heater motor speed should be MEDIUM.

Based on the assumption made for the membership function and fuzzy rules, we can illustrate this defuzzification process using a graph. Four fuzzy rules can be interpreted as functional diagrams, as shown in Fig. 7.10.

As an example, consider the current input temperature is 35 °F and the change rate of the temperature is 1 °F per hour.

From Fig. 7.10, it can be found that the points of intersection between the temperature values of 35 °F and the graph in the first column (temperature input T) have the membership functions of 0.6, 0.8, 0.5 and 0.8. Likewise, the second column (temperature change rate ΔT) shows that a temperature change rate of 1 °F per hour has the membership functions of 1.0, 0.4, 0.4, and 1.0. The fuzzy output for the four rules is the intersection of the paired values obtained from the graph, or the AND result between the temperature input and the temperature change rate input. According to Eq. (7.13), this operational result should be: min (0.6, 1.0), min (0.8, 0.4), min (0.5, 0.4), and min (0.8, 1.0), which produces to 0.6, 0.4, 0.4, and 0.8, respectively.

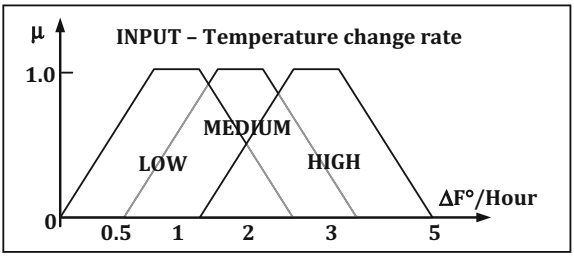


Fig. 7.9 The membership function of the change rate of the temperature

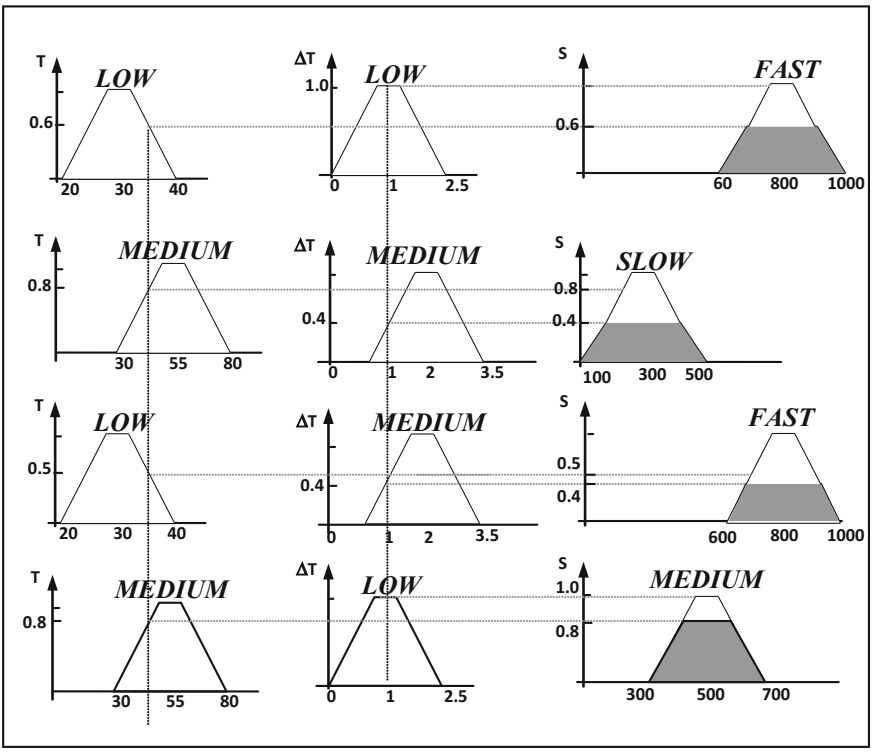


Fig. 7.10 An illustration of the fuzzy output calculation

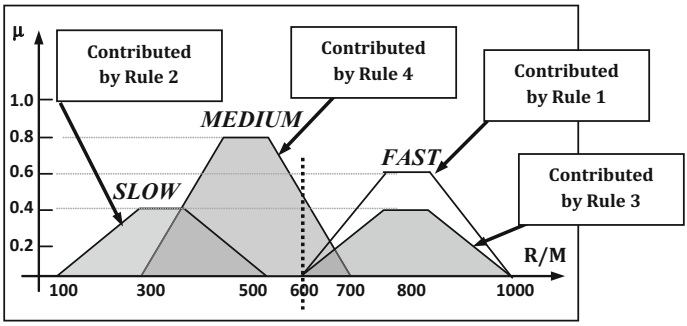


Fig. 7.11 Determination of the fuzzy output by the center-of-gravity method

The membership functions representing the control adjustment are weighted according to the input change and the different control contributions as shown in Fig. 7.11.

Now, for a pair of temperature and temperature change rate, four sets of fuzzy outputs exist. To determine the crisp value of action to be taken from these contributions, one can either choose the maximum value using the MOM method or use the Center-of-Gravity method (COG). In this example, the COG method is used and the action is given by the center of the summed area, which is contributed by the different fuzzy outputs. Furthermore, the COG method gives a more reliable lookup table compared with the MOM operation.

Thus, for a temperature of 35 °F and a change rate of the temperature is 1 °F per hour, the fuzzy output element y for this input pair is

$$y = \frac{0.6 \times 800 + 0.4 \times 300 + 0.4 \times 800 + 0.8 \times 500}{0.6 + 0.4 + 0.4 + 0.8} = 600 \text{ R/M} \tag{7.20}$$

This defuzzified fuzzy output is a crisp or classical value and should be entered into a certain location in a table called lookup table. Since this fuzzy output element is associated with a temperature input pair with a temperature of 35 °F (belong to LOW in the temperature membership function) and a change rate of the temperature of 1 °F/H (also belong to LOW in the membership function of change rate of the temperature), this output value should be located in the cross point between the LOW temperature T (row) and the LOW change rate $\dot{T}$ (column) as shown in Table 7.2.

Table 7.2 shows an example of a lookup table. To fill this table, one needs to use the defuzzification technique to calculate all other fuzzy output values and locate them to the associated positions in the lookup table as we did in this example. Generally, the dimensions of the fuzzy rules should be identical with the dimensions of the lookup table as shown in this example. To obtain more accurate control accuracy or fine fuzzy output element values, one can divide the inputs, say the temperature input and change rate of the temperature, into multiple smaller subsets to get finer membership functions. For instance, in this example, one can define a VERY LOW subset that covers 20 ~ 30 °F for the temperature input, a LOW subset that contains a range of 30 ~ 40 °F, and so on. Perform the same process to the change rate of the temperature and the heater motor output, one can get a much finer lookup table.

For higher control accuracy applications, an interpolation process can be added after the lookup table to obtain finer output.

When implementing a fuzzy logic technique to a real system, the lookup table can be stored in a computer's memory space and the fuzzy output can be obtained

Table 7.2 An example of a lookup table

$\dot{T}$	T		
	LOW	MEDIUM	HIGH
LOW	600	?	?
MEDIUM	?	?	?
HIGH	?	?	?

based on the current inputs. In practice, there are two ways to calculate the lookup table using fuzzy inference process in the real control applications: offline and online methods.

7.5.5 *The Offline and Online Defuzzifications*

As we mentioned in the previous sections, the defuzzification process is to derive the desired crisp output value using a defuzzification technique that combines the membership functions with the fuzzy rules together. The defuzzification process can be further divided into two categories: offline defuzzification and online defuzzification.

The so-called offline defuzzification means that all input and output membership functions, fuzzy rules, and lookup table should be developed based on the estimations of the real application prior to the implementation of the fuzzy logic technique to the actual application. This means that all inputs and outputs membership functions are developed based on the actual experience or the input and output parameter ranges of a specified application, and the lookup table is calculated in terms of those definitions of input and output membership functions. The advantage of this method is that most fuzzy inference-related calculations are performed prior to the real implementation, and therefore, the fuzzy process is less time-consuming. The disadvantage of this technique is that the fuzzy output is only based on the estimation of inputs and outputs parameters, so the control accuracy is not as high as that of online method.

The online method has the real-time controllability. Both inputs and outputs membership functions are developed during the real-time processing of an actual application. Also, the lookup table elements are calculated in real time based on the current actual inputs and outputs. In this method, only fuzzy rules are developed prior to the real applications. The advantage of this method is that higher control accuracy can be obtained for a process and the fuzzy output can be computed in real time. The disadvantage of this method is that a longer processing time is needed and it is a little time-consuming process. But as the development of new computer technologies, today much faster CPUs are available, and the processing time is no longer a big deal for this method.

7.6 Architectures of Fuzzy Logic Control Systems

Combined the discussion we made in the previous sections, a structure or architecture of fuzzy logic control system is given in this part.

As shown in Fig. 7.12, it is a typical fuzzy closed-loop control system. The inputs are error and error rate, which are combined together by block *M* to input to the fuzzy inference system. The lookup table is derived based on the membership

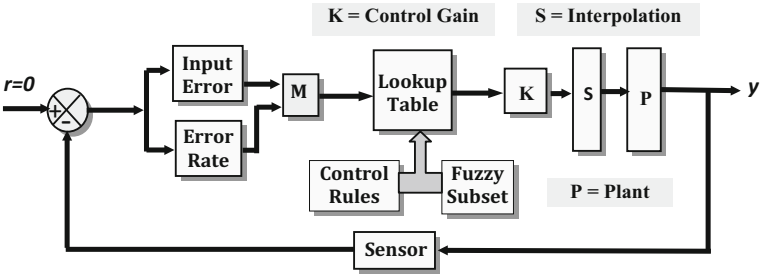


Fig. 7.12 The block diagram of a fuzzy control system

function of inputs, the output, and the fuzzy control rules. A control gain factor G is used to tune the output of the lookup table to obtain different output values. The interpolation block S is used to smooth the output element of the lookup table. A feedback signal is obtained from the output of the system.

For a system that needs higher control accuracy, a multiple lookup tables fuzzy control system is needed, which is shown in Fig. 7.13.

Two lookup tables can be developed for a control system, a coarse and a fine table. During the application, the switch between the coarse and the fine table is under the control of the input error limit. This limit value can be defined by the user based on the real application. Two-set membership functions and control rules are utilized in this system to satisfy the requirement of higher control accuracy. When the system needs quick responses or quick actions, the coarse table is used. While the system needs high control accuracy or small control errors, the fine lookup table is selected. The sacrifice for this method is that more memory spaces are needed to store both coarse and fine tables, and a little longer time to make decision in selecting table in terms of the input error limit value.

A review of fundamental of fuzzy set, fuzzy rules, and fuzzy inference system is provided in this part. Beginning with the crisp or classical sets and its operations, we derived the fuzzy sets and its operations. The classical set membership functions

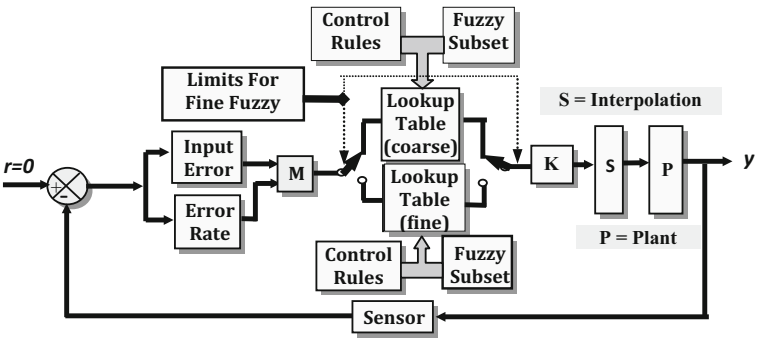


Fig. 7.13 The block diagram of an accurate fuzzy control system

and the fuzzy membership functions are discussed in detail following the sets theory. Fuzzy rules are described using an air conditioner control example. The different defuzzification techniques and their processes are discussed with the same example step-by-step. Finally, some other fuzzy techniques are discussed such as the offline and online fuzzy control systems as well as a fuzzy closed-loop control system including multiple lookup tables.

7.6.1 Fuzzy Proportional–Derivative (PD) Controller

A conventional PD control algorithm or system is defined as

$$u(t) = K_{\mathrm{P}} e(t) + K_{\mathrm{D}} \frac{\mathrm{d}e(t)}{\mathrm{d}t} \tag{7.21}$$

The digital implementation for this PD control algorithm can be represented as

$$\Delta u_k = K_{\mathrm{P}} e_k + K_{\mathrm{D}} \Delta e = K_{\mathrm{P}} e_k + K_{\mathrm{D}} (e_k - e_{k-1}) \tag{7.22}$$

$$u_{k+1} = u_k + \Delta u_k \tag{7.23}$$

Equations (7.22) and (7.23) would become a fuzzy-PD controller if e_k and Δe_k are fuzzy variables. Therefore, a practical fuzzy-PD control algorithm can be obtained and can be presented as

$$\begin{aligned} u_{k+1} &= u_k + \Delta u_k = u_k + K \Delta U_k \\ \Delta U_k &= F\{E_k, \Delta E_k\} = F\{K_{\mathrm{P}} e_k, K_{\mathrm{D}} \Delta e_k\} \end{aligned} \tag{7.24}$$

where F signifies the fuzzy function that acts on the rules given in the form of the lookup table. A functional block diagram of a fuzzy-PD control system is shown in Fig. 7.14.

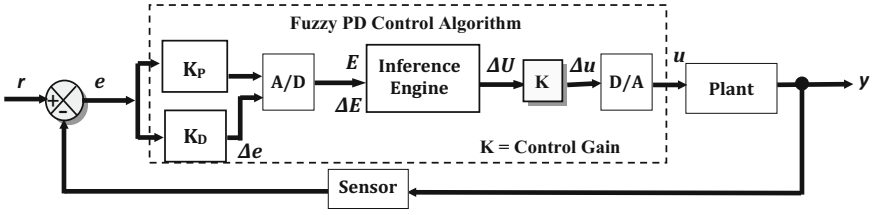


Fig. 7.14 Fuzzy-PD control system block diagram

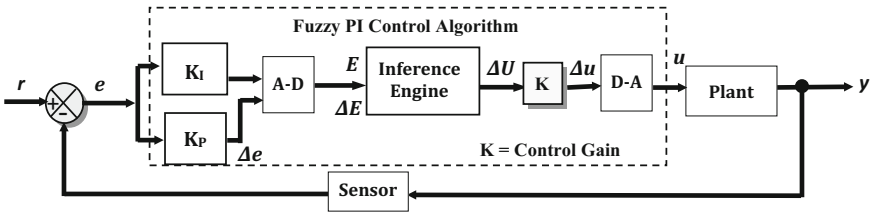


Fig. 7.15 Fuzzy-PI control system block diagram

7.6.2 *Fuzzy Proportional Integrated (PI) Controller*

A conventional PI control algorithm or system is defined as

$$u(t) = K_{\mathrm{P}} e(t) + K_{\mathrm{I}} \int e \mathrm{d}t \tag{7.25}$$

The digital implementation for this PI control algorithm can be represented as

$$\Delta u_k = K_{\mathrm{P}} \Delta e_k + K_{\mathrm{I}} e_k = K_{\mathrm{P}}(e_k - e_{k-1}) + K_{\mathrm{I}} e_k \tag{7.26}$$

$$u_{k+1} = u_k + \Delta u_k \tag{7.27}$$

Equations (7.26) and (7.27) would become a fuzzy-PI controller if e_k and Δe_k are fuzzy variables. Therefore, a practical fuzzy-PI control algorithm can be obtained and can be presented as

$$\begin{aligned} u_{k+1} &= u_k + \Delta u_k = u_k + K \Delta U_k \\ \Delta U_k &= F\{E_k, \Delta E_k\} = F\{K_{\mathrm{I}} e_k, K_{\mathrm{P}} \Delta e_k\} \end{aligned} \tag{7.28}$$

where F signifies the fuzzy function that acts on the rules given in the form of the lookup table. A functional block diagram of a fuzzy-PI control system is shown in Fig. 7.15.

Next, let's introduce a powerful tool, MATLAB Fuzzy Logic Toolbox, to assist users to design, build, and simulate a fuzzy closed-loop control system easily, efficiently, and conveniently.

7.7 Introduction to MATLAB Fuzzy Logic Toolbox

According to the type definitions, the fuzzy logic inference systems can be divided into two categories, Type 1 and Type 2, fuzzy system. In this chapter, we mainly concentrate on Type 1 fuzzy logic system and will discuss Type 2 fuzzy system in Chap. 9.

MATLAB provides a very powerful tool, Fuzzy Logic Toolbox, to assist and enable users to design, build, and implement fuzzy logic controllers in an easy and efficient way. Basically, this Toolbox is composed of a set of functions that provide a wide spectrum of operations to enable users to design and build their fuzzy control system in a convenient way. In addition to these functions, MATALB also provides a Fuzzy Logic Toolbox Graphical User Interface Tool (FLTGUIT) for these functions to enable users to access them via a visual way to make their design and building of fuzzy logic controller much easier and interesting. First, let's concentrate on these fuzzy functions, and then we will provide a detailed introduction to that FLTGUIT.

Most functions can be categorized into the following groups based on their functions:

(1) Create or generate a new fuzzy inference system.
(2) Add new variables into the fuzzy inference system.
(3) Add new membership functions into the fuzzy inference system.
(4) Add, edit, and display control rules in the fuzzy inference system.
(5) Perform fuzzy inference calculations and evaluate membership functions.
(6) Plot and show the fuzzy inference system.
(7) Save and load the fuzzy inference system.

Now let's have a detailed discussion about each function on these groups.

7.7.1 Fuzzy Inference Functions

The following functions are included in group 1 category:

- `newfis()`—Create a new fuzzy inference system.
  The syntax is: `fis = newfis('name')`. This function returns a default Mamdani fuzzy inference system with the specified `name`.
- `genfis()`—Generate a new fuzzy inference system structure from data.
  The syntax is: `fis = genfis(inputData,outputData)`. This function returns a single-output Sugeno fuzzy inference system (FIS) using a grid partition of the given input and output data. The inputData is a *N*-Column array, where *N* is the number of FIS inputs. Similarly, the `outputData` is a *M*-Column array, and *M* is the number of FIS outputs.

The following functions are included in group 2 category:

- `addvar()`—Add new variables into the fuzzy inference system.
  The syntax is: `fis = addvar(fis,varType,varName,varBounds)`. This function contains four input arguments:
  - `fis`—Fuzzy inference system in the Workspace, specified as an FIS structure.
  - `varType`—Type of variable to add, specified as "input" or "output".

  - `varName`—Name of the variable to add, specified as a character vector or string.
  - `varBounds`—Variable range, specified as a two-element vector, where the first element is the minimum value and the second element is the maximum value for the variable.

- `rmvar()`—Remove variable from the current fuzzy inference system.
  The syntax is: `fis = rmvar(fis,varType,varIndex)`. This function removes the variable `varType`, of index `varIndex`, from the fuzzy inference system associated with the workspace FIS structure fis:

  - `SpecifyvarType` as either "input" or "output".
  - `varIndex` is an integer for the index of the variable. This index represents the order in which the variables are listed.

The following three functions belong to the group of adding new membership functions into the fuzzy inference system:

- `addmf()`—Add new membership function (MF) into the fuzzy inference system.
  The syntax is: `fis = addmf(fis,varType,varIndex,mfName, mfType,mfParams)`. A point to be noted is that a membership function cannot be added into any fuzzy inference system (FIS) directly; instead, it must be added only to a variable in an existing FIS. Indices are assigned to membership functions in the order in which they are added. The function requires six input arguments and they are:

  - `fis`—Fuzzy inference system in the workspace, specified as an FIS structure.
  - `varType`—Type of variable to add membership function to, can be either as "input" or "output".
  - `varIndex`—Index of variable to add membership function to, specified as a positive integer less than or equal to the number of variables of the type specified by `varType`.
  - `mfName`—The name of the new membership function, can be a character vector or a string.
  - `mfType`—The type of the new membership function.

- `rmmf()`—Remove a membership function from the fuzzy inference system.
  The syntax is: `fis = rmmf(fis,varType,varIndex,'mf',mfIndex)`.
  An example of using these functions to create a new FIS named `tipper`, add new variables and membership functions into this FIS is shown below:

```
a = newfis('tipper');                              % create a new FIS tipper
a = addvar(a,'input','service',[0 10]);            % add a new input variable
a = addmf(a,'input',1,'poor','gaussmf',[1.5 0]);   % add a new MF named poor
a = addmf(a,'input',1,'good','gaussmf',[1.5 5]);   % add a new MF named good
a = addmf(a,'input',1,'excellent','gaussmf',[1.5 10]);
```

- `mfedit()`—Open the membership editor in the FLTGUIT.
  The syntaxes are: `mfedit(fis)` and `mfedit(fileName)`. The first function `mfedit(fis)` operates on a MATLAB® workspace variable, for an FIS structure, `fis`. The function `mfedit(fileName)` generates a membership function editor that allows you to inspect and modify all the membership functions for the FIS stored in the file, `fileName`.

The following five functions belong to the group of adding, editing, and displaying control rules in the fuzzy inference system:

- `addrule()`—Add a new rule to the fuzzy inference system.
  The syntax is: `fis = addrule(fis,ruleList)`. The second argument `ruleList` is a matrix of one or more rows, each of them represents a given rule. The format of the rule list matrix is very specific. If there are m inputs and n outputs, there must be exactly $m + n + 2$ columns to the rule list.
- `showrule()`—Display the rule list for the given FIS.
  The syntax is: `showrule(fis)`.
- `parsrule()`—Parse fuzzy rules.
  The syntax is: `newFIS = parsrule(oldFIS,ruleList)`. This function returns a fuzzy inference system, `newFIS`, which is equivalent to the input fuzzy system, `oldFIS`, but with fuzzy rules replaced by the rules specified in `ruleList`.
- `ruleedit()`—Open the rule editor.
  The syntaxes are: `ruleedit(fileName)` and `ruleedit(fis)`. The first function `ruleedit(fileName)` opens the Rule Editor for the FIS stored in the file, `fileName`. The second function `ruleedit(fis)` operates on a MATLAB® workspace variable, for an FIS structure, `fis`.
- `ruleview()`—Open the rule viewer in the FLTGUIT. The syntaxes are: `ruleview(fileName)` and `ruleview(fis)`. The function `ruleview (fileName)` depicts the fuzzy inference diagram for the fuzzy inference system stored in file `fileName`. The function `ruleview(fis)` opens the Rule Viewer for the fuzzy inference system, `fis`. It is used to view the entire implication process from beginning to end. You can move around the line indices that correspond to the inputs and then watch the system readjust and compute the new output.

To understand the format of the rule list matrix, let's see an example.

For instance, if we have a fuzzy inference system called `tipper`, which is used to calculate the amount of the tip to be paid for the entire service in a restaurant. Three variables, two input variables, `service`, `food`, and one output variable `tip`, are added into this inference system. Also, each variable has three related membership functions shown below:

(1) For `service`, three levels of MF are: **poor**, **good**, and **excellent**.
(2) For `food`, three levels of MF are: **rancid**, **acceptable**, and **delicious**.
(3) For `tip`, three levels of MF are: **cheap**, **average**, and **generous**.

Based on these variables and membership functions, a simple rule list can be written as

1. `If(service is poor)and(food is rancid)then(tip is cheap).`
2. `If(service is good)and(food is rancid)then(tip is average).`
3. `If(service is excellent)and(food is delicious)then(tip is generous).`

In this rule list, each MF level is indexed with a rising order.

1. For service, `poor` = 1, `good` = 2, and `excellent` = 3.
2. For food, `rancid` = 1, `acceptable` = 2, and `delicious` = 3.
3. For tip, `cheap` = 1, `average` = 2, and `generous` = 3.

Based on these MF levels definitions, we can rewrite the above three rules as a rule list:

```
1       1       1
2       1       2
3       3       3
```

The following two functions belong to the group of performing fuzzy inference calculations and evaluating membership functions:

- `evalfis()`—Perform fuzzy inference calculations. The syntaxes are: `output = evalfis(input,fismat)` and `output = evalfis(input, fismat,numPts)`.
  This function has the following input arguments:
  - `input`—Number or a matrix specifying input values. If input is an **M**-by-**N** matrix, where **N** is number of input variables, then the function takes each row of input as an input vector and returns the **M**-by-**L** matrix to the variable, `output`, where each row is an output vector and **L** is the number of output variables.
  - `fismat`—FIS structure to be evaluated.
  - `numPts`—Number of sample points on which to evaluate the membership functions over the output variable range, specified as an integer greater than 1. If you do not specify `numPts`, a default value of 101 will be used.

  An example of using this function is

```
fismat = readfis('tipper');
output = evalfis([2 1;4 9],fismat)
```

- `evalmf()`—Calculate a membership function for the given inputs.
  The syntax is: `y = evalmf(x,mfParams,mfType)`. This function evaluates any membership function, where `x` is the variable range for the membership

function evaluation, `mfType` is a membership function from the toolbox, and `mfParams` are appropriate parameters for that function.
An example of using the `evalmf()` function is

```
x = 0:0.1:10;
mfparams = [2 4 6];
mftype = 'gbellmf';
y = evalmf(x,mfparams,mftype);
```

The following five functions belong to the group of plotting and showing the fuzzy inference system:

- `plotfis()`—Plot the fuzzy inference system. The syntax is: `plotfis(fismat)`. This function displays a high-level diagram of an FIS, `fismat`.
- `showfis()`—Display the contents of the selected fuzzy inference system. The syntax is: `showfis(fismat)`.
- `surfview()`—Open the surface viewer to display the surface or envelope of the fuzzy inference system in 3D view. The syntax is: `surfview(fis)`.
- `plotmf()`—Plot the membership functions for the given variable.
  The syntax is: `plotmf(fis,varType,varIndex)`. This function plots all membership functions in the FIS named `fis` associated with a given variable whose type and index are, respectively, given by `varType` (either "input" or "output"), and `varIndex`. This function can also be used with the `subplot()` function.
- `gensurf()`—Generate a fuzzy inference system output surface. This function has the similar function as that of `surfview()`, but the only difference is that this function does not open the surface viewer. The syntax is: `gensurf(fis)`.

The following functions belong to the group of saving and loading fuzzy inference systems:

- `writefis()`—Save fuzzy inference system into a file.
  The syntaxes are: `writefis(fis)` and `writefis(fis, fileName)`. The function `writefis(fis)` saves a MATLAB® workspace FIS structure, `fis`, as a `.fis` file. The function `writefis(fis,fileName)` writes a `.fis` file corresponding to the FIS structure, `fis`, to a file named `filename.fis`.
- `readfis()`—Load a fuzzy inference system from a file.
  The syntax is: `fis = readfis(fileName)`.

Two examples of using `writefis()` and `readfis()` are:
`writefis(a,'tipper'); and a = readfis('tipper');`
Next let's take care of the fuzzy logic graphical user interface tool (FLGUIT) since it provides a powerful and convenient way to help users to design, build, and implement fuzzy inference system to perform fuzzy controls in real world.

7.7.2 *Fuzzy Logic Toolbox Graphical User Interface Tool (FLTGUIT)*

The main purpose of this tool is to set up a bridge via this graphical user interface to enable users to easily and quickly design and build a fuzzy inference system by accessing those fuzzy functions we discussed in the Fuzzy Logic Toolbox in the above section. The advantage of using this FLTGUIT is that users do not need to know very details about those fuzzy functions, and only by performing some dragging, moving, and clicking related trajectory elements, they can easily design and build a professional fuzzy inference system to get their design objectives. Because most or all of details are hidden by the graphical user interface, users do not need to take care of those functions. However, the shortcoming is that it would be difficult to build more professional fuzzy inference systems in real time since most details about those functions are not clear to users and those functions could be black boxes for the designers.

Therefore, in order to design and build more professional fuzzy inference system and apply them in the real time, a detailed study and understanding about those fuzzy functions are prerequisite requirements.

The architecture or a relationship between the fuzzy logic toolbox graphical user interface tool (FLTGUIT) and all fuzzy functions located in the Fuzzy Logic Toolbox is shown in Fig. 7.16.

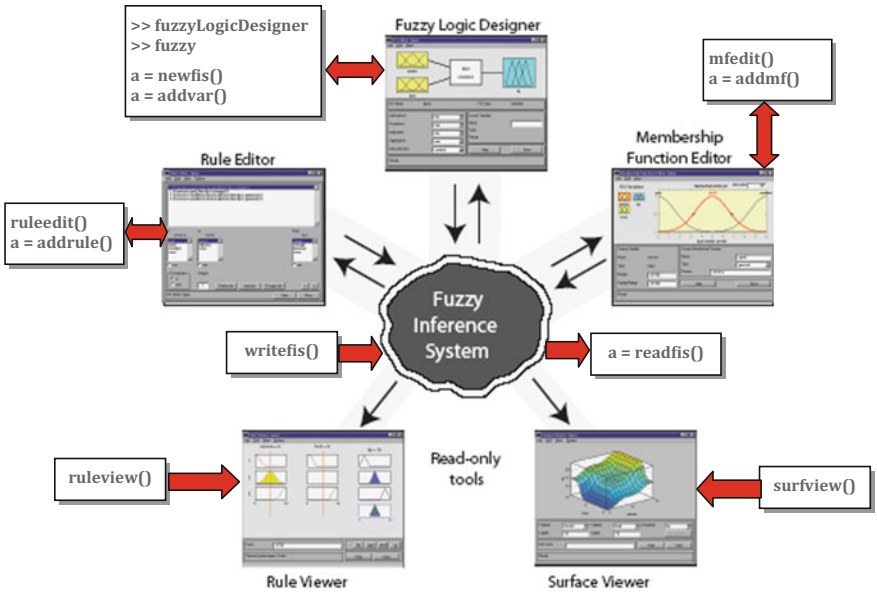


Fig. 7.16 Relationships between each UI and related functions. Reprinted with the permission of MathWorks, Inc.

The entire FLTGUIT is composed of five User Interfaces (UTs) with the Fuzzy Inference System as a core:

(1) Fuzzy Logic Designer,
(2) Membership Function Editor,
(3) Rule Editor,
(4) Rule Viewer, and
(5) Surface Viewer.

Next, let's discuss each of them one by one.

7.7.2.1 Fuzzy Logic Designer

This UI provides an entry point for this tool to enable users to open the FLTGUIT to create a new FIS, import an existing FIS or export a completed FIS to the workspace. One can open this UI by typing command **fuzzy** or **fuzzyLogicDesigner** in the MATLAB Command window and press the **Enter** key on the keyboard. Three menu items are available to assist users to design and build an FIS in an easy and interesting way. These menu items are (refer to Fig. 7.17):

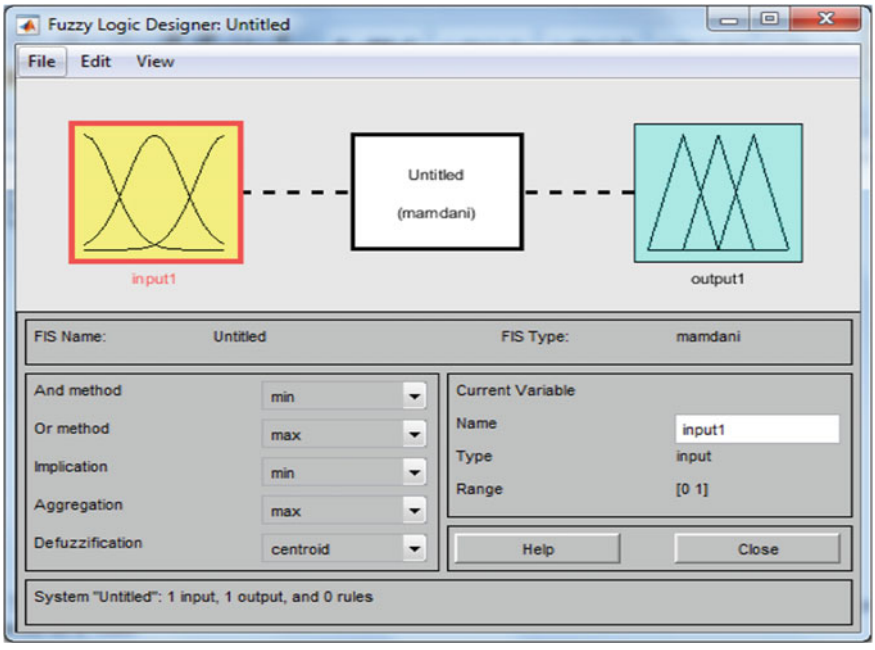


Fig. 7.17 The opened Fuzzy Logic Designer. Reprinted with the permission of MathWorks, Inc.

- **File→New FIS**, **File→Import**, **File→Export**. For a new FIS, two types of FIS can be built: Mamdani and Sugeno. For both Import and Export, one can either import from a file or from the workspace. Same selections are available to the Export operation.
- The **Edit** menu item provides more powerful functions with four items:
  - **Edit→Add Variable→Input** or **Output**. This item enables users to add fuzzy variables, either input or output, to the FIS. In fact, this is equivalent to calling the fuzzy function `addvar()` we discussed above. The default variable is **Input 1** for the Input, and **Output 1** for the Output variable. The default name for the new added variable is **Input 2** or **Output 2** for the Input or the Output variable. One can change or modify the default name (**Input 1**) for the added variable by typing a new name in the **Name** box, as shown in Fig. 7.17.
  - **Edit→Remove Selected Variable** item enables users to remove the added and selected variable. This item is equivalent to the fuzzy function `rmvar()`.
  - **Edit→Membership Functions** item allows users to configure and modify the membership function for the selected variable. Figure 7.18 shows an example of the opened Edit Membership Function UI for the first variable—Input 1. With the help of this UI, users can

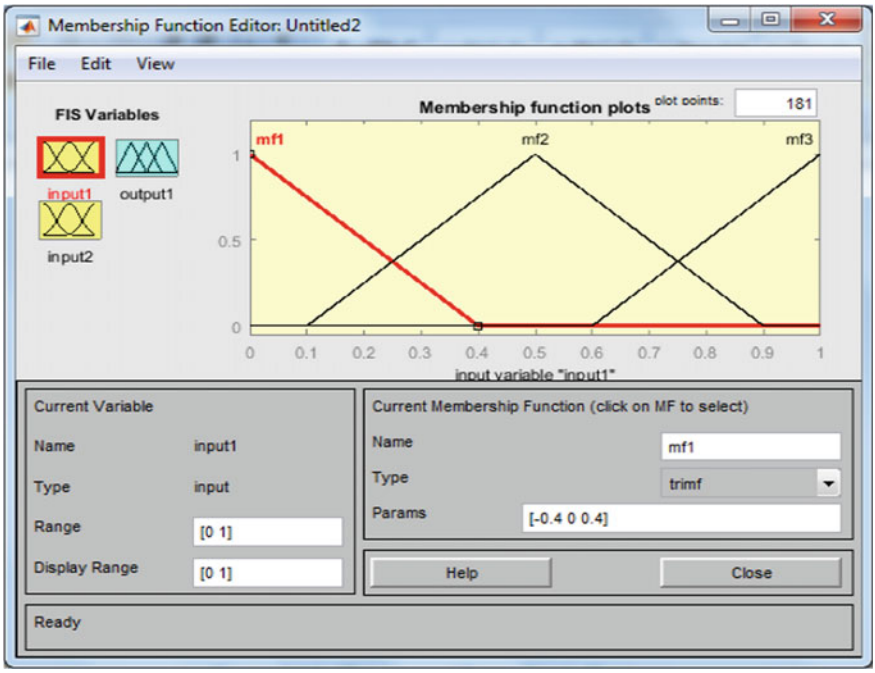


Fig. 7.18 The opened Edit Membership Functions UI. Reprinted with the permission of MathWorks, Inc.

(1) Define and change the name for the selected member in this membership function by typing a new name in the **Name** box. In Fig. 7.18, the selected member is **mf1** for the variable **Input 1**.
(2) Define and select different or desired type for the membership function by going to the **Type** combo box and selecting the desired membership function type from the drop-down list. The default type is the triangle **trimf**, which is shown in Fig. 7.18.
(3) Define and change the range for the entire membership function or universe by going to the **Range** box and typing the desired range into this box. The default Range is [0 1], as shown in Fig. 7.18.
(4) Adjust or move the selected member by dragging that member trajectory either to left or right. This is equivalent to adjusting the parameters for the selected member and the values in the **Params** box will also be modified as one did this moving. One can do the similar adjusting by clicking and selecting some other members one by one, such as **mf2** and **mf3**, respectively.
(5) Close this membership function editing by clicking on the **Close** button when the editing job is done. All adjusting and modifications will be saved automatically.

- **Edit→Rules** item enables users to open the **Rule Editor** to create and modify control rules for the selected FIS. An example for the opened Rule Editor is shown in Fig. 7.19. In fact, this editor allows users to build their IF-THEN rules based on their specified applications. The default setups for this editor include:

(1) The connection operation is **AND**, but one can select the **OR** operation by checking the **OR** radio button in the Connection group box, as shown in Fig. 7.19.
(2) The default **Weight** for each rule is 1, which is indicated in the **Weight** box. One can change this weight for the selected rule by typing the desired number into this Weight box.
(3) The default condition for each variable is definite or yes. One can change this condition to not by checking the **not** checkbox under the rule list.
(4) To begin to add a new rule, one can select the first condition or MF from the **Input 1** variable list by clicking on it, and then select the second condition or MF from the second input, **Input 2** variable list and the output from the **output 1** list.
By clicking on the **Add rule** button, the selected rule will be added into the rule editor.
(5) By clicking on the **Close** button, the Rule Editor will be closed and all new added rules can be saved automatically. We will discuss this editor in more details in the later Sect. 7.7.2.3.

One can also change or delete the selected rule by clicking on the **Change rule** or the **Delete rule** buttons, respectively. Figure 7.19 shows an example of customer selected rules. Here three rules are selected based on three members,

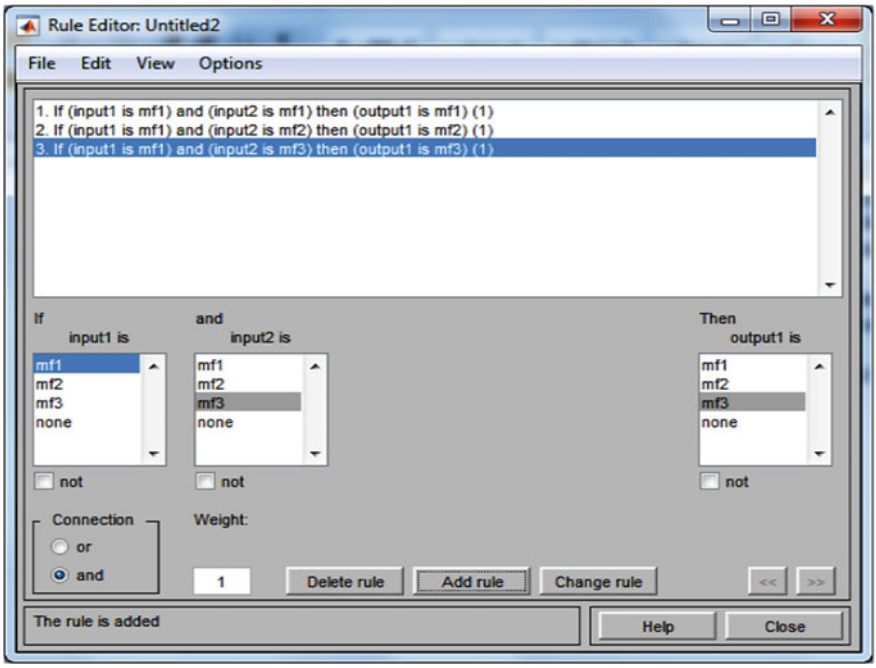


Fig. 7.19 The opened Rule Editor UI. Reprinted with the permission of MathWorks, Inc.

mf1, **mf2**, and **mf3**, for three variables, **Input 1** and **Input 2** as well as the output variable **output 1**. The number (1) attached at the end of each rule is used to indicate that this rule is a single rule without any duplication or connection with any other rules.

- The **View** menu item provides two functions with two items:
  - **View→Rules** item enables users to open the **Rule Viewer** to have a dynamic or real-time view for the FIS operations, as shown in Fig. 7.20. Still we use a rule example with three rules built in Fig. 7.19. As shown in Fig. 7.20, the so-called dynamic or real-time view for this FIS is that when you moving or shifting any vertical bar crossing all membership functions for each variable to left or right, the desired output will be obtained and displayed in the output variable immediately in real time. This can be seen clearly from the example shown in Fig. 7.20. When moving or shifting the vertical bar crossing all MFs for each variable, including both inputs and the output, the instant values of inputs and the output are displayed on the top of each variable. For this example, the inputs are: **Input 1** = 0.234, **Input 2** = 0.723, and the output **output 1** = 0.567. This is a very convenient and easy way to derive the input–output relationship in the real time.
    One can also get more flexible views by clicking on the **left**, **right**, **down**, or **up** button located at the lower bottom corner of this Viewer to move the

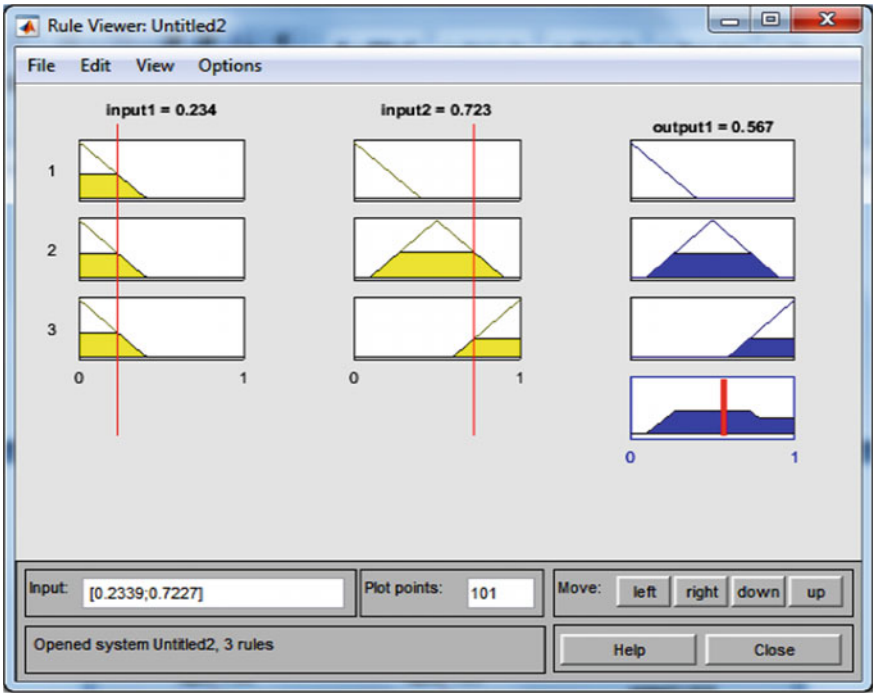


Fig. 7.20 The opened rule viewer UI. Reprinted with the permission of MathWorks, Inc.

entire view to the different directions. By clicking on the **Close** button, the Viewer can be closed and all current settings can be saved automatically.

– **View→Surface** item enables users to open the **Surface Viewer** to display a 3D envelope of surface of the desired FIS. As shown in Fig. 7.21, two inputs, **Input 1** and **Input 2**, work as the *x*-*y* plane variables, and the output is displayed as the third dimension variable. One can change the points of grid for both *X* and *Y* direction by modifying the number in the **X grids** or **Y grids** box. One can also modify the variable location by selecting the different or desired variable from the **X(Input)**, **Y(Input)**, and **Z(output)** combo boxes, respectively. This Viewer can be closed by clicking on the **Close** button and all current settings will be saved automatically.

Now let's have a closer look at the Membership Function Editor.

7.7.2.2 Membership Function Editor

When clicking on the **Edit→Membership Functions** item from the Fuzzy Logic Designer, the Membership Function Editor is opened, as shown in Fig. 7.18. In this part, we will provide more detailed discussions about this editor.

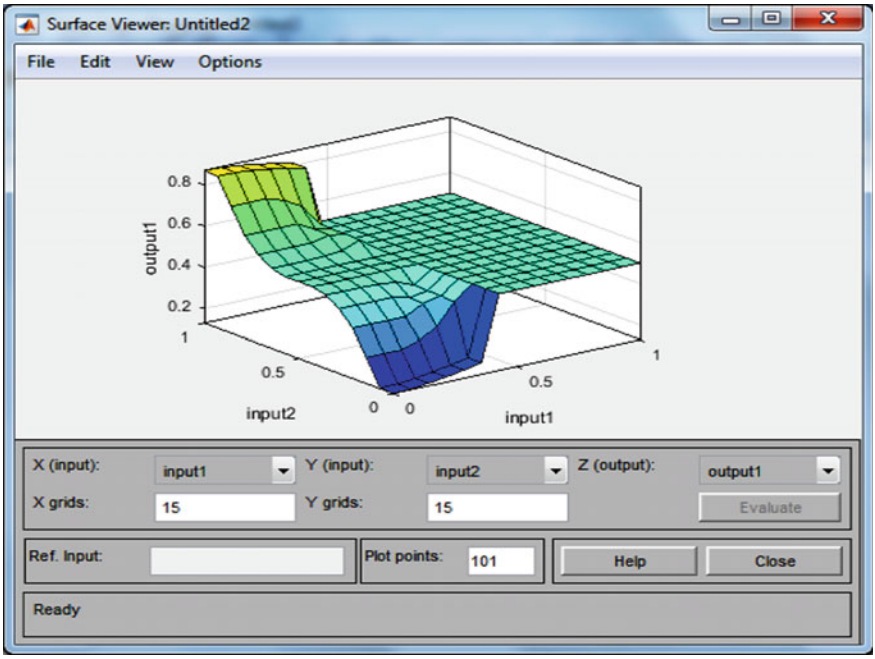


Fig. 7.21 The opened Surface Viewer UI. Reprinted with the permission of MathWorks, Inc.

Three menu items are available in this UI: **File**, **Edit**, and **View**. Both items, File and View, are identical with those used in the Fuzzy Logic Designer. The only different item is the Edit, which provides more powerful functions to enable users to manage their selected or desired membership functions. Totally seven (7) submenus are available under this Edit item, they are:

- **Edit→Add MFs** item enables users to add new Membership Functions (MFs) based on the existing or default MFs. A popup dialog box will be displayed when this item is selected to ask you to enter the type and the number of new members for new MF to be added.
- **Edit→Add Customer MF** item enables users to add the customer-made MF. The so-called customer-made MF is that you need to select all parameters related to the desired customer MF, such as the type, name, range, and parameter list, yourself without any default values being available.
- **Edit→Remove Selected MF** item allows users to delete the selected or added MF. Before activating this item, you need to select at least one MF from the UI, which is to be removed.
- **Edit→Remove All MFs** item enables users to remove all existing MFs from the UI. Of course, you can recover all of those removed MFs by going to the **Edit→Undo** menu item.

- **Edit→FIS Properties** item returns to the Fuzzy Logic Designer UI.
- **Edit→Rules** item opens the Rule Editor.

You can close this UI by clicking on the **Close** button. All current settings will be saved automatically when this UI is closed.

7.7.2.3 Rule Editor

The Rule Editor has been discussed in Sect. 7.7.2.1 and shown up in Fig. 7.19. Four (4) menu items are available in the UI: **File**, **Edit**, **View**, and **Options**. The functions of two of them, File and View, are identical with those menus in the Fuzzy Logic Designer. Three submenu items are under the Edit menu, and they are:

- **Edit→FIS Properties** item returns to the Fuzzy Logic Designer.
- **Edit→Membership Functions** item opens the Membership Function Editor UI.
- **Edit→Undo** item recovers or undoes the previous operation.

Two submenu items are under the Options item and they are:

- **Options→Language** item allows users to select different language for the rule definitions.
- **Options→Format** enables users to select different format to represent the rules. The default format for the rule representations is the **Verbose**. A useful format is the **Indexed**. Once this format is selected, each rule is represented by a sequence of index, just as we did for a rule list example in Sect. 7.7.1. For example, for our created three rules shown in Fig. 7.19, after the Indexed format is selected, those rules become the indexed format:

```
1       1       1 (1)
1       2       2 (1)
1       3       3 (1)
```

Now that we have some knowledge and understanding about the fuzzy logic toolbox and fuzzy control technologies, let's start to implement them in our actual design applications.

7.8 Implementation of Fuzzy Logic Controls with Fuzzy Logic Toolbox

As we mentioned, there are two ways to implement fuzzy logic toolbox in real applications. The first way is to access fuzzy inference functions via Fuzzy Logic Toolbox Graphical User Interface Tool (FLTGUIT), and the second way is to directly access and use those fuzzy functions to build our fuzzy control systems. The advantage of using the first way is that the designers do not need to know details about those fuzzy functions, and they can easily design and build a fuzzy logic control system or fuzzy inference system via various UIs provided by the FLTGUIT. However, the shortcoming is that the applications built in this way have some implementation limitations. For example, they can only be used in the FLTGUIT environment, but cannot be implemented in a real-time application. The advantage of using the second way is that any application developed in that way can be used in all different environments in real time.

First, let's use the first way to design and build a fuzzy inference system to control a DC motor system.

7.8.1 Applications of FLTGUIT for a DC Motor System

In this section, we will use an example DC Motor system we used for Lab Project 5.1 in Chap. 5. The given transfer function or dynamic model is

$$G(s) = \frac{6520}{s(s+430.6)} e^{-0.005\,s} \tag{7.29}$$

The propagation time delay for this model is 0.005 s or 5 ms.

In fact, we do not need any dynamic model when applying a fuzzy logic controller to a control system since it is a modeless controller. Here just for the reference purpose.

The functional block diagram of this DC motor fuzzy logic closed-loop control system is shown in Fig. 7.22.

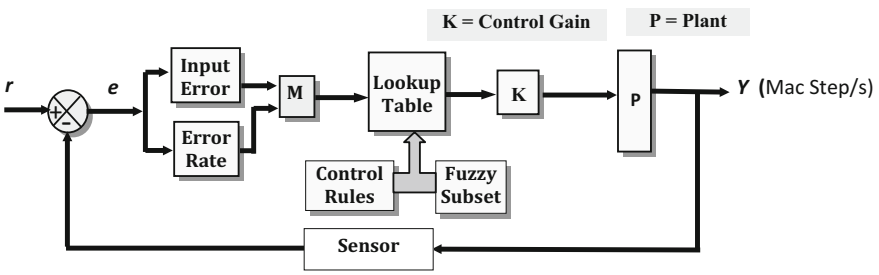


Fig. 7.22 The DC motor fuzzy logic closed-loop control system

Table 7.3 Ranges of the input–output variables

Variables	Variables ranges	Units
Input error	−0.1 to 0.1	V
Input error rate	−0.06 to 0.06	V
Motor output speed	−600 to 600	Machine steps/s

The error input voltage is coming from a tachometer checking the actual motor velocity and the setup velocity of the motor's rotation. The sensor is used to transfer the motor rotation velocity (*machine steps/second*) to the related error voltages.

Two inputs are applied to this control system, the error and the error rate. The control gain G is used to adjust and tune the fuzzy logic control output via a lookup table.

The required parameters we need to design a fuzzy logic controller (FLC) are the input and output variable ranges for this DC motor system. Table 7.3 shows the input and the output ranges for this motor control system.

7.8.1.1 Design of the Membership Functions for the Fuzzy Logic Control System

To represent the error input and error rate input using fuzzy sets, a set of linguistic variables is chosen to represent a 7-*degree* of error, 7-*degree* of error rate and 7-*degree* of control output.

Membership functions are constructed to represent the input and output in which degree belongs to a different membership set or linguistic variable set.

The membership functions of the input error, the error rate, and the controller output are shown in Fig. 7.23. The units for both the input error and error rate are *Voltage*, and the unit for output u is *Machine steps/second*.

The linguistic variables are defined as follows:

- LN—Large Negative,
- MN—Medium Negative,
- SN—Small Negative,
- ZE—Zero,
- SP—Small Positive,
- MP—Medium Positive, and
- LP—Large Positive.

Open the Fuzzy Logic Toolbox by typing the command **fuzzyLogicDesigner** or **fuzzy** in the MATLAB Command window and press the **Enter** key. Create three membership functions for the input error **Error**, input error rate **ErrorRate**, and the output **U** with the following operational steps:

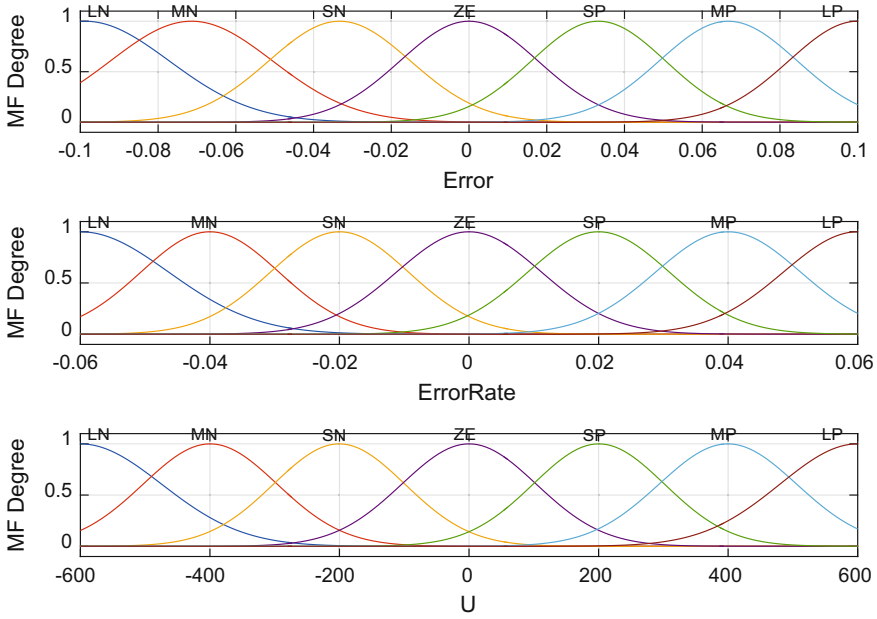


Fig. 7.23 Membership functions for inputs and output

(1) Go to **Edit→Add Variable→Input** to add the second input variable. Change the name for this variable to **ErrorRate** by typing this into the **Name** box. Click on the first input variable **Input 1** and perform the same operation to change its name to **Error**.
(2) Now double click on the **Error** variable icon to open the Membership Function Editor. Go to **Edit→Remove All MFs** to remove all default MFs.
(3) Then go to **Edit→Add MFs** to open the Membership Function dialog box. Enter **7** into the Number of **MFs** box and select gaussmf item from the **MF type** combo box, and click on the **OK** button to add these MFs into the FIS.
(4) Change the name for each member function by clicking on it to select each one, and type the related name, such as LN, MN, SN, ZE, SP, MP, and LP, to each **Name** box. You can also change the range for each function by dragging each trajectory either to left or right to get them a wider range to cover.
(5) Modify the range for the entire universe of discourse by typing the desired upper and lower bounds into the **Range** box. For the **Error** input, they are −0.1 and 0.1. For the **ErrorRate**, they are −0.06 to 0.06. For the output **U**, they are −600 to 600.
(6) Perform the similar operations to all membership functions of three variables. An example MF of the input **Error** is shown in Fig. 7.24.

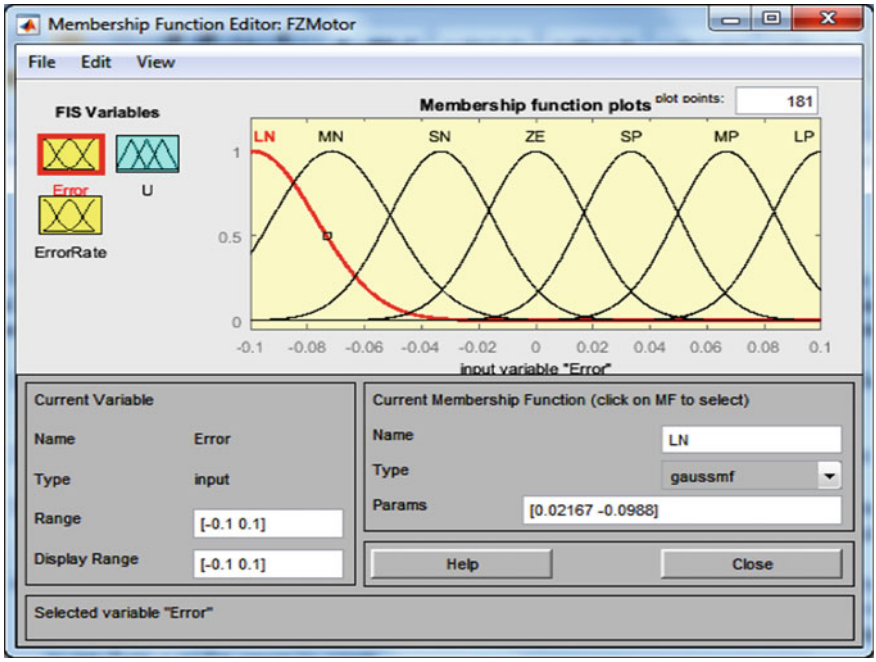


Fig. 7.24 An example MF for the input error—Error

The reason we select the type of all the membership functions as Gaussian waveform is that we want to obtain more smooth control output to the DC motor to get better responses.

The sensor used in this closed-loop system and shown in Fig. 7.22 is a tachometer that works as a coefficient used to transfer the motor rotation speed to the related voltage. The value of this coefficient is calculated by dividing the upbound of the input voltage and the motor speed (0.1/600 = 0.00017 V/Machine Steps).

At any time, you can save this FIS either to a file or to the workspace with the desired name by going to the menu **File→Export→To File** or **File→Export→To Workspace**. In this application, we save this FIS as a file named **FZMotor.fis** and **FZMotor** as the name to the workspace.

7.8.1.2 Design of the Control Rules for the Fuzzy Logic Control System

The control rules are really the control laws that should be determined by the mathematical descriptions or some physical laws. Also, these control rules should be combined with the membership functions to form the lookup table to obtain the output of the FLC. The determination of the control rules depends highly on the application. For example, in a simple temperature control system, the air condition should be turned off when the room becomes cool enough and should be turned on

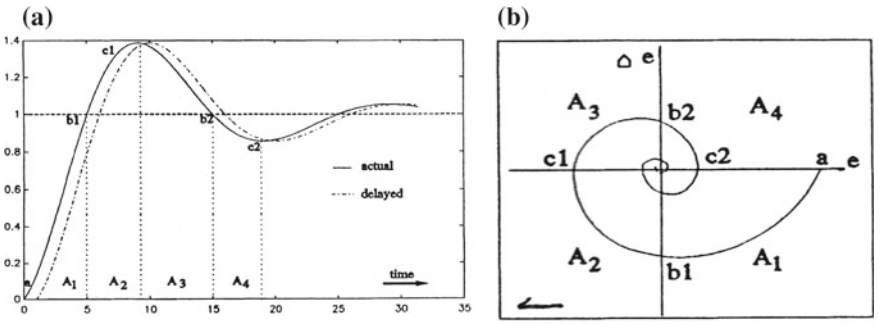


Fig. 7.25 Mapping of step response between the time domain and the phase plane

when the room becomes too warm. This is a control rule if "cool enough" and "too warm" are quantified and this is a crisp control rule. The effect of this control is limited and relatively poorer than that of the fuzzy logic control. The rules in the FLC depend on the human judgment to a great extent.

The control rules can be designed based on the common senses of human beings or based on some control theory tools, such as the step response of the system in the time domain. The step response in the time domain can be mapped into the phase plane. The relationship between the time domain and phase plane for the step response is used to derive the relevant control rules. This method has some advantages over some other methods. One important feature of the method is that the phase advance information can be introduced into the phase plane, and in turn, be integrated into the control rules to improve the effect of the time delay existed in the system.According to H. X. Li and H. B. Gatland's study[1], a step response in the time domain can be mapped to a related response in the phase plane, as shown in Fig. 7.25.

As shown in Fig. 7.25a, a step response in the time domain can be divided into four areas, $\mathbf{A_1}$, $\mathbf{A_2}$, $\mathbf{A_3}$, and $\mathbf{A_4}$ with five separating points, $\boldsymbol{a}$, $\boldsymbol{b_1}$, $\boldsymbol{b_2}$, $\boldsymbol{c_1}$, and $\boldsymbol{c_2}$, respectively. These areas and points can be equivalently mapped to a phase plane as shown in Fig. 7.25b. The horizontal axis is defined as the error $\boldsymbol{e}$ and the vertical axis is defined as the error rate $\boldsymbol{\Delta e}$. The definitions for these two variables are:

$$e = 1 - y \quad \text{and} \quad \Delta e = e_k - e_{k-1}$$

Based on these definitions, four areas of a step response in the time domain can be defined and mapped to the phase plane as (refer to Fig. 7.25a, b):

- $\mathbf{A_1}$: $e > 0$ and $\Delta e < 0$ ⟺ $\mathbf{A_1}$ in the phase plane
- $\mathbf{A_2}$: $e < 0$ and $\Delta e < 0$ ⟺ $\mathbf{A_2}$ in the phase plane
- $\mathbf{A_3}$: $e < 0$ and $\Delta e > 0$ ⟺ $\mathbf{A_3}$ in the phase plane
- $\mathbf{A_4}$: $e > 0$ and $\Delta e > 0$ ⟺ $\mathbf{A_4}$ in the phase plane

[1]H. X. Li and H. B. Gatland, "A New Methodology for Designing a Fuzzy Logic Controller", *IEEE Trans. On Systems, Man, and Cybernetics*, Vol. 25, No. 3, March 1995, pp. 505–512.

The step response with the solid line in Fig. 7.25a is an ideal response, but the step response with the dashed line is an actual response with a propagation time delay.

In order to compensate this time delay in the time domain, one can just shift the ideal response to the right with a time interval that is equal to the time delay. This shifting is equivalent to shifting four areas $\mathbf{A_1} \sim \mathbf{A_4}$ with five points, $\boldsymbol{a}$, $\boldsymbol{b_1}$, $\boldsymbol{b_2}$, $\boldsymbol{c_1}$, and $\boldsymbol{c_2}$ together to right. However, if this shifting is mapped to the phase plane, it is equivalent to shifting trajectory at the bottom half in the phase plane ($\Delta e < 0$) to the right, and the trajectory at the top half plane ($\Delta e > 0$) to the left. Also, this shift can be mapped to the rule base that is shown in Table 7.4b. On comparing the control rules shown in Table 7.4a and rule base shown in Table 7.4b, one can find a similar mapping relationship between the real control rules and the rule base.

In this application, we decide to build 49 rules based on two 7-degree MFs for error and error rate, which is shown in Table 7.4a. When performing the shifting in the phase plane as we did above, this shifting can be equivalent to shifting the bottom rules to the right and the top rules to the left to compensate or reduce the effect of the propagation time delay on a control system.

Using the definitions and analysis above, we can derive all elements of the control rules for our FLC. A total of 49 control rules (7-*degree* error input and 7-*degree* error rate input) are shown in Table 7.4a. This is a regular control rule without time delay ($T_d = 0$).

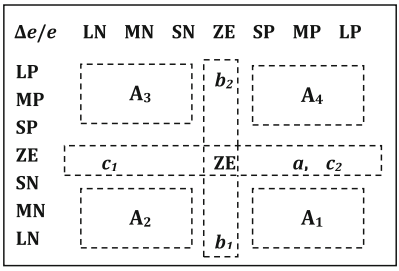
One of the advantages of using this kind of control rule is that, as mentioned before, it can effectively reduce the effect of the time delay existed in the system. The reason for this is that the time delay can be mapped into the fuzzy logic control system by shifting the control rules in the phase plane.

For most of real systems, a propagation time delay always exists and causes some undesired characteristics to the control system. In the worst situation, the output of the system will be unstable or even oscillate. In order to overcome the time delay, the phase lag technique was commonly adopted in traditional digital control systems.

Table 7.4 (a) Fuzzy control rules

Fuzzy control rules							
$\dot{e}$	e						
	LN	MN	SN	ZB	SP	MP	LP
LP	ZE	SP	SP	MP	MP	LP	LP
MP	SN	ZE	SP	SP	MP	LP	LP
SP	MN	SN	ZE	SP	SP	LP	LP
ZE	LN	MN	SN	ZE	SP	MP	LP
SN	LN	LN	SN	SN	ZE	SP	MP
MN	LN	LN	MN	SN	SN	ZE	SP
LN	LN	LN	MN	MN	SN	SN	ZE

Table 7.4 (b) Mapping of the rule base



The main idea in using FLC to overcome time delay is to introduce some delay information into the original control system, or equivalently, to introduce the delay information into the control rules in the phase plane. The equivalent action in the phase plane is to shift the control rules and the resultant control rules are shown in Table 7.5.

Now let's build our control rules by using the MATLAB Fuzzy Logic Toolbox Graphical User Interface Tool. On the opened **FZMotor** fuzzy inference system, go to **Edit→Rules** to open the **Rule Editor**. Build our control rules based on Table 7.4 using the IF-THEN conditions. In Table 7.4, each column represents the **Error** value and each row represents the **ErrorRate** value. The intersection cell for each column and each row is the related output value.

For example, in the first column and the first row as well as the related intersection cell in Table 7.4a, the control rule is:

IF the **Error** is LN and the **ErrorRate** is LP, THEN the output **U** is ZE.

This is making sense since the motor rotating speed should be zero if the input error is LN and the error rate is LP.

Build this control rule table in the Rule Editor by selecting the related MFs for input error, error rate, and the output, and then click the **Add rule** button to add

Table 7.5 Shifted fuzzy control rules

Shift ←	$\dot{e}$	Fuzzy control rules after shifting							
		e							
		LN	MN	SN	ZE	SP	MP	LP	
	LP	SP	SP	MP	MP	LP	LP	LP	
	MP	ZE	SP	SP	MP	LP	LP	LP	
	SP	SN	ZE	SP	SP	LP	LP	LP	
	ZE	LN	MN	SN	ZE	SP	MP	LP	
	SN	LN	LN	LN	SN	SN	ZE	SP	
	MN	LN	LN	LN	MN	SN	SN	ZE	
	LN	LN	LN	LN	MN	MN	SN	SN	Shift →

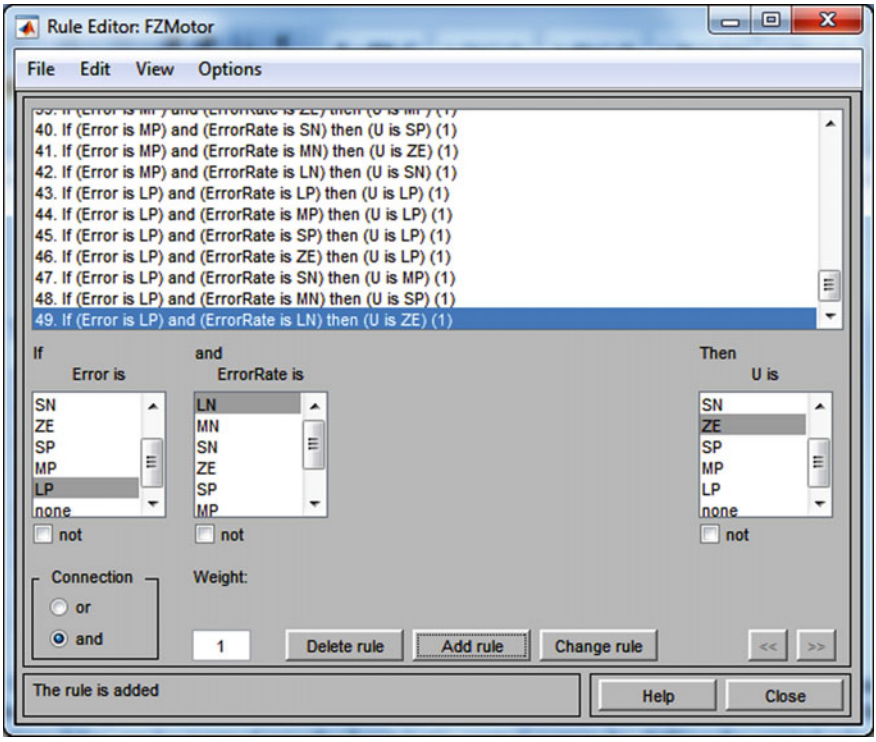


Fig. 7.26 The finished control rule table

each of rules into the editor. An example of the finished rule table is shown in Fig. 7.26. Click on the **Close** button to save and close this editor.

By going to **View→Rules** menu item, the **Rule Viewer** is opened and all of 49 rules can be displayed with the real-time inputs and output values displayed, as shown in Fig. 7.27. When moving the vertical bar for both inputs, the output can be obtained immediately in real time.

The envelope or surface view of this fuzzy control system is shown in Fig. 7.28.

7.8.2 Applications of Fuzzy Logic Controls with Fuzzy Functions

In this section, we will discuss how to access fuzzy inference functions to build our fuzzy logic controller with the same DC motor system introduced in the last section.

Here we want to create same membership functions for the input error **Error**, error rate **ErrorRate**, and the output **U** by using the fuzzy logic toolbox functions.

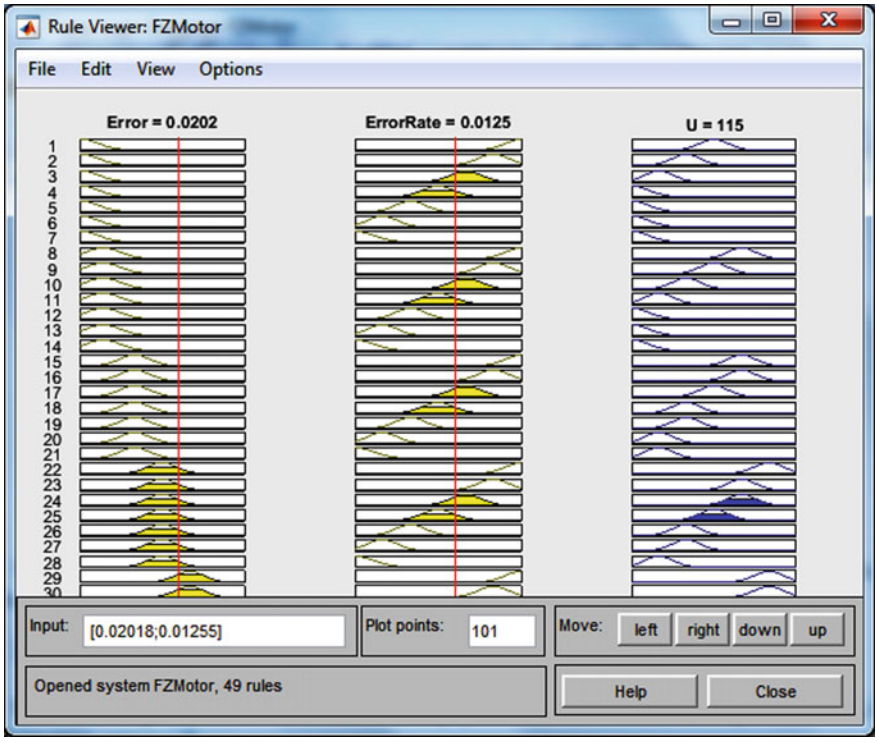


Fig. 7.27 The opened Rule Viewer

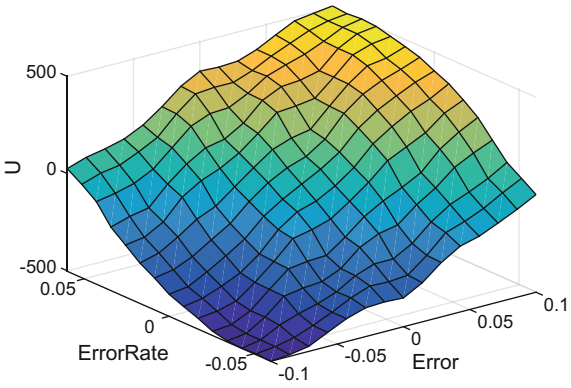


Fig. 7.28 The envelope view or the surface view of the FZMotor FIS

As we discussed in Sect. 7.7.1, the operational sequence of creating a new fuzzy inference system and adding membership functions and control rules into that FIS should include the following steps:

(1) Using **newfis()** or **genfis()** function to create a new FIS.
(2) Using **addvar()** to add variables into this newly created FIS.
(3) Using **addmf()** function to add new membership functions to the inputs and the output variables. One can refer to the parameters and ranges used for the same membership functions created by using the FLTGUIT in the last section.
(4) Calling **addrule()** function to add new control rules into this FIS. One can refer to the rules created in the last section to perform this operation. An easy way to do this is to convert the control rules from the verbal to the indexed format to complete this rule addition.
(5) Using **writefis()** function to save the new created FIS into a file and to the workspace.
(6) Calling **plotmf()** and **gensurf()** functions to display the membership functions and the envelope or the surface of the created FIS.
(7) Calling **evalfis()** function to get the real-time output of the FIS based on the current input values.

Now let's start this process by following these steps to create a new FIS, add new variables and add membership functions into this FIS with fuzzy logic toolbox functions.

7.8.2.1 Create a New FIS and Add Membership Functions

Create a new MATLAB Script file by going to **HOME→New** Script menu item from the MATLAB main window and save this file as **FZMotorFunc.m** to your desired folder. Since we need to copy all parameters for all MFs and rules from the fuzzy inference system **FZMotor.fis** created in the last section, and paste them into the related fuzzy functions in this section, thus type the following commands on the MATLAB Command window to get all parameters:

```
>> a = readfis('FZMotor');
>> showfis(a)
```

All parameters related to the FIS **FZMotor.fis** are displayed in this Command window. Later we can select and use some of them in our following fuzzy functions.

Then enter the following codes into the created script file **FZMotorFunc.m**.

```
a = newfis('FZMotorFunc');                                  % create a new FIS
a = addvar(a,'input','Error',[-0.1 0.1]);                   % add a new input var
a = addmf(a,'input',1,'LN','gaussmf',[0.02167 -0.0988]);% add a new MF LN
a = addmf(a,'input',1,'MN','gaussmf',[0.02087 -0.0714]);% add a new MF MN
a = addmf(a,'input',1,'SN','gaussmf',[0.01823 -0.0333]);% add a new MF SN
a = addmf(a,'input',1,'ZE','gaussmf',[0.01775 0]);          % add a new MF ZE
a = addmf(a,'input',1,'SP','gaussmf',[0.01727 0.0333]); % add a new MF SP
a = addmf(a,'input',1,'MP','gaussmf',[0.01778 0.0667]); % add a new MF MP
a = addmf(a,'input',1,'LP','gaussmf',[0.01775 0.1]);        % add a new MF LP

a = addvar(a,'input','ErrorRate',[-0.06 0.06]);             % add a new input var
a = addmf(a,'input',2,'LN','gaussmf',[0.01362 -0.06]);   % add a new MF LN
a = addmf(a,'input',2,'MN','gaussmf',[0.01065 -0.04]);   % add a new MF MN
a = addmf(a,'input',2,'SN','gaussmf',[0.01065 -0.02]);   % add a new MF SN
a = addmf(a,'input',2,'ZE','gaussmf',[0.01119 0]);          % add a new MF ZE
a = addmf(a,'input',2,'SP','gaussmf',[0.01092 0.02]);    % add a new MF SP
a = addmf(a,'input',2,'MP','gaussmf',[0.01119 0.04]);    % add a new MF MP
a = addmf(a,'input',2,'LP','gaussmf',[0.01146 0.06]);    % add a new MF LP

a = addvar(a,'output','U',[-600 600]);                    % add a new output var
a = addmf(a,'output',1,'LN','gaussmf',[125.4 -600]);        % add a new MF LN
a = addmf(a,'output',1,'MN','gaussmf',[103.5 -399.6]);   % add a new MF MN
a = addmf(a,'output',1,'SN','gaussmf',[101.4 -200.4]);   % add a new MF SN
a = addmf(a,'output',1,'ZE','gaussmf',[103.8 0]);           % add a new MF ZE
a = addmf(a,'output',1,'SP','gaussmf',[101.4 200.4]);    % add a new MF SP
a = addmf(a,'output',1,'MP','gaussmf',[106.2 399.6]);    % add a new MF MP
a = addmf(a,'output',1,'LP','gaussmf',[120 600]);           % add a new MF LP
```

All related parameters that have been added into the above FIS file can be found from the running results of the function `showfis(a)`. Here is a part of running results:

```
1.  Name             FZMotor
2.  Type             mamdani
3.  Inputs/Outputs   [2 1]
4.  NumInputMFs      [7 7]
5.  NumOutputMFs     7
6.  NumRules         49
7.  AndMethod        min
8.  OrMethod         max
9.  ImpMethod        min
10. AggMethod        max
11. DefuzzMethod     centroid
12. InLabels         Error
13.                  ErrorRate
14. OutLabels        U
15. InRange          [-0.1 0.1]
16.                  [-0.06 0.06]
17. OutRange         [-600 600]
```

The names for three new added variables can be found from the **InLabels** and the **OutLabels**, and the ranges for those variables can be gotten from **InRange** and **OutRange**.

The names of all MFs, including both input MFs and the output MFs, can be found from the **InMFLabels** and **OutMFLabels**, respectively (not shown here).

The value ranges for all those `addmf()` functions used in above file can be found from the **InMFParams** and **OutMFParams**, as shown below.

```
60. InMFParams      [0.02167 -0.0988 0 0]   : LN - Error (1)
61.                 [0.02087 -0.0714 0 0]   : MN - Error (2)
62.                 [0.01823 -0.0333 0 0]   : SN - Error (3)
63.                 [0.01775 0 0 0]         : ZE - Error (4)
64.                 [0.01727 0.0333 0 0]    : SP - Error (5)
65.                 [0.01778 0.0667 0 0]    : MP - Error (6)
66.                 [0.01775 0.1 0 0]       : LP - Error (7)
67.                 [0.01362 -0.06 0 0]     : LN - ErrorRate (1)
68.                 [0.01065 -0.04 0 0]     : MN - ErrorRate (2)
69.                 [0.01065 -0.02 0 0]     : SN - ErrorRate (3)
70.                 [0.01119 0 0 0]         : ZE - ErrorRate (4)
71.                 [0.01092 0.02 0 0]      : SP - ErrorRate (5)
72.                 [0.01119 0.04 0 0]      : MP - ErrorRate (6)
73.                 [0.01146 0.06 0 0]      : LP - ErrorRate (7)
74. OutMFParams     [125.4 -600 0 0]        : LN - U (1)
75.                 [103.5 -399.6 0 0]      : MN - U (2)
76.                 [101.4 -200.4 0 0]      : SN - U (3)
77.                 [103.8 0 0 0]           : ZE - U (4)
78.                 [101.4 200.4 0 0]       : SP - U (5)
79.                 [106.2 399.6 0 0]       : MP - U (6)
80.                 [120 600 0 0]           : LP - U (7)
```

These parameters are arranged in the same sequence as those names of all MFs listed in the **InMFLabels** and **OutMFLabels**, starting from LN, MN, SN, ZE, SP, MP, and LP. One can just copy and paste them one by one into the `addmf()` functions in the above FIS file. One point to be noted is that one only needs to copy and paste the first two numbers but ignore the last two numbers (0 0) since they are not important to the usage of Gaussian curve. The first two numbers represent the standard deviation and center of the Gaussian curve, which are enough to define a Gaussian waveform and use it as a type of the MF. Each number attached at the end of each MF parameter and is inside of the parenthesis is the order number of each MF.

7.8.2.2 Create and Add New Rules to the FIS

To make this adding control rules operation easier, first one needs to open the file **FZMotor.fis** with the FLTGUI Tool and convert all rules to the indexed format. Then one can copy, paste, and attach all rules into the script file **FZMotorFunc.m** under the codes we created in the last section.

Perform the following operational steps to create and add 49 control rules into this script:

(1) Type **fuzzy** or **fuzzyLogicDesigner** into the MATLAB Command window and press the **Enter** key from the keyboard to open the FLTGUIT if it is not opened.
(2) Go to **File→Import→From File** menu item to open the **FZMotor.**fis file.
(3) Go to **Edit→Rules** menu item to open the Rule Editor.
(4) Go to **Options→Format→Indexed** menu item to convert all 49 rules to the indexed format. Now all rules are represented in the indexed format, such as 1 7, 4 (1): 1. This format is equivalent to a verbose representation of IF-THEN

clause. The equivalent representation is: **IF Error is LN and ErrorRate is LP, THEN U is ZE**. The Error MF LN is ordered as 1, the ErrorRate MF LP is defined as 7, the U MF ZE is ordered as 4. The number 1 inside the parenthesis is the weight of this rule, and the last number 1 is the connection number to other control rules. Regularly the last two numbers should be equal to 1 for most normal rules. When writing this rule into the **RuleList**, the format should be: 1 7 4 1 1.

(5) Now open the script file **FZMotorFunc.m**, and perform the copy of each rule from the Indexed rules in the **FZMotor.fis**, and then paste each of them to the end of the script file **FZMotorFunc.m** with the Indexed format as shown above.

Your finished **ruleList** should match one that is shown below.

```
ruleList = [1 7 4 1 1
            1 6 3 1 1
            1 5 2 1 1
            1 4 1 1 1
            1 3 1 1 1
            1 2 1 1 1
            1 1 1 1 1
            2 7 5 1 1
            2 6 4 1 1
            2 5 3 1 1
            2 4 2 1 1
            2 3 1 1 1
            2 2 1 1 1
            2 1 1 1 1
            3 7 5 1 1
            3 6 5 1 1
            3 5 4 1 1
            3 4 3 1 1
            3 3 3 1 1
            3 2 2 1 1
            3 1 2 1 1
            4 7 6 1 1
            4 6 5 1 1
            4 5 5 1 1
            4 4 4 1 1
            4 3 3 1 1
            4 3 2 1 1
            4 1 2 1 1
            5 7 6 1 1
            5 6 6 1 1
            5 5 5 1 1
```

```
5 4 5 1 1
5 3 4 1 1
5 2 3 1 1
5 1 3 1 1
6 7 7 1 1
6 6 7 1 1
6 5 7 1 1
6 4 6 1 1
6 3 5 1 1
6 2 4 1 1
6 1 3 1 1
7 7 7 1 1
7 6 7 1 1
7 5 7 1 1
7 4 7 1 1
7 3 6 1 1
7 2 5 1 1
7 1 4 1 1];
```

Finally, use the following two functions to add and save all rules into this FIS:

```
a = addrule(a, ruleList);
writefis(a, 'FZMotorFunc');
```

To test this FIS, one can use the following functions to plot the **ErrorRate** membership function and evaluate a real-time output based on an example input pair:

```
a = readfis('FZMotorFunc');
plotmf(a, 'input', 2);
grid;
evalfis([-0.01 0.02], a)     % calculate an output when Error = -0.01 &
                             % ErrorRate = 0.02
```

The MF of the ErrorRate should be plotted and the evaluation result is **U** = 78.3152.

7.9 Tuning of the Fuzzy Logic Controller

The most difficult part with fuzzy logic controller is the parameter tuning. The objective of the tuning of an FLC is to obtain the optimal or desired design criteria by selecting the proper combination of all control gains or parameters.

As we discussed in Sects. 7.6.1 and 7.6.2, for the most popular fuzzy logic controllers, such as the fuzzy-PD or fuzzy-PI controllers, this tuning strategy or

objective is to select the proper values for three control gains, $\boldsymbol{K_P}$, $\boldsymbol{K_D}$, and $\boldsymbol{K}$ (for the fuzzy-PD control algorithm), or $\boldsymbol{K_P}$, $\boldsymbol{K_I}$, and $\boldsymbol{K}$ (for the fuzzy-PI control algorithm), respectively, to obtain the optimal control objectives or the desired step responses.

During the tuning process of an FLC, the control gains $\boldsymbol{K_P}$ and $\boldsymbol{K_D}$ (or $\boldsymbol{K_P}$ and $\boldsymbol{K_I}$) can be used to adjust or tune the fuzziness of the membership functions for inputs, including both input error and error rate. In this way, we can obtain and test many different Membership Functions (MFs) to try to get the optimal output.

An example input error membership function represented by using the triangle type is shown in Fig. 7.29. A scale factor or control gain, K, is used to multiply the input error to obtain different ranges of the input error. As shown in Fig. 7.29, by changing the scale factor K's value, different MFs can be obtained.

- When $K = 1$, the original membership function is obtained.
- When $K > 1$, a more coarse membership function is obtained.
- When $K < 1$, a finer membership function is obtained.

For the input error rate, one can use the same strategy to obtain different membership functions.

The control gain K can be used to adjust or tune the output of an FLC. During the tuning process of an FLC, many different lookup tables should be tested in order to achieve the desired performance for the control system. It is very tough and time-consuming during the tuning of the FLC if each time a new lookup table needs to be calculated based on the membership functions and control rules. So an easier method is to use a control gain factor, K to handle this tuning process. The output of the lookup table can be conveniently changed and tuned by adjusting the value of the control gain factor. With the help of this control gain, multiple lookup tables can be obtained from one lookup table. The principle of the tuning control gain factor $\boldsymbol{K}$ is:

- When $K < 1$, a finer lookup table is obtained.
- When $K > 1$, a coarser lookup table is obtained.
- When $K = 1$, the original lookup table is obtained.

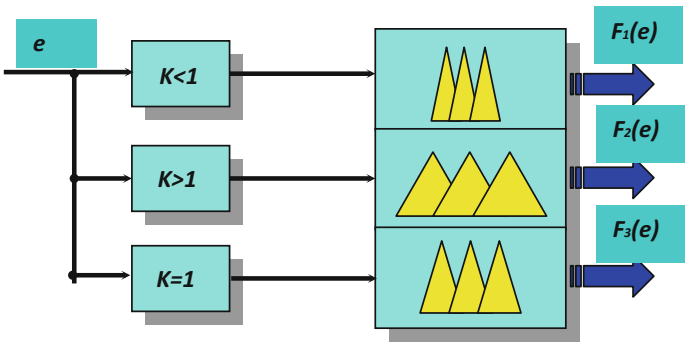


Fig. 7.29 The tuning of membership functions

With the help of this method, a substantiate amount of memory in the computer can be saved. For each fixed gain factor $\boldsymbol{K}$, a related lookup table can be obtained. If totally N different gain factors were used, N different lookup tables will be achieved and available. For example, if an application used a 15 *by* 15 lookup table (the actual amount of elements used in a lookup table depends on the actual system requirement), which means that totally 225 elements will be used for only one lookup table. If eight different control gains are used in that application, as the result, eight different lookup tables were used for each situation. Thus this means that we need to store and search $8 \times 225 = 1800$ elements in our computer memory. In this way, not only a lot of memory would be used but also the processing speed would be greatly affected by searching the control element concerned. So the significance of using the gain factors lies that it not only saves the memory but also speeds up the processing speed and in turn the response speed of the control system.

One issue for the tuning of an FLC is that it is difficult to maintain excellent performance in both transient and steady state for any practical control systems. This is because different control resolutions are required for transient and steady state for a selected control system. For an ideal transient response, large errors need coarse control that requires coarse input and output variables. However, in the steady state, small errors need a fine fuzzy control, which requires fine input and output variables. A more practical way to get both transient and steady-state errors to be handled is to use two lookup tables, as we discussed at the beginning of Sect. 7.6.

7.10 Implementation of Fuzzy Logic Controls with Simulink®

Now that the fuzzy logic control inference system has been built, we need to check and confirm the controllability of the fuzzy logic control system. MATLAB® Simulink® provides a powerful and helpful way to enable us to do this kind of jobs.

On the opened MATLAB® Command window, type the command **Simulink** and press the **Enter** key from the keyboard to open the **Simulink Library Browser**, as shown in Fig. 7.30.

Click on the **New Model** combo box and select the **New Model** item, which is shown in Fig. 7.30, to open a new Simulink model window. Then select and drag all control blocks and add each of them to the new model window. Your finished model should match one that is shown in Fig. 7.31. Save this model as **FZMotor_FLC.slx** by going to **File→Save** menu item.

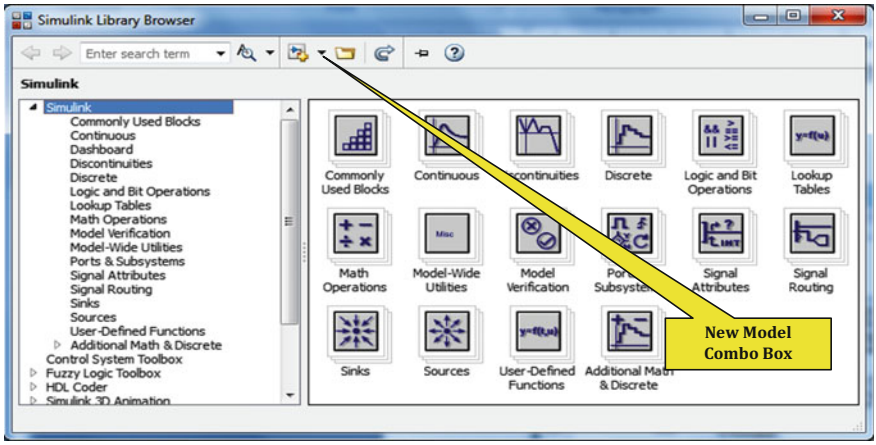


Fig. 7.30 The opened Simulink Library Browser. Reprinted with the permission of MathWorks, Inc.

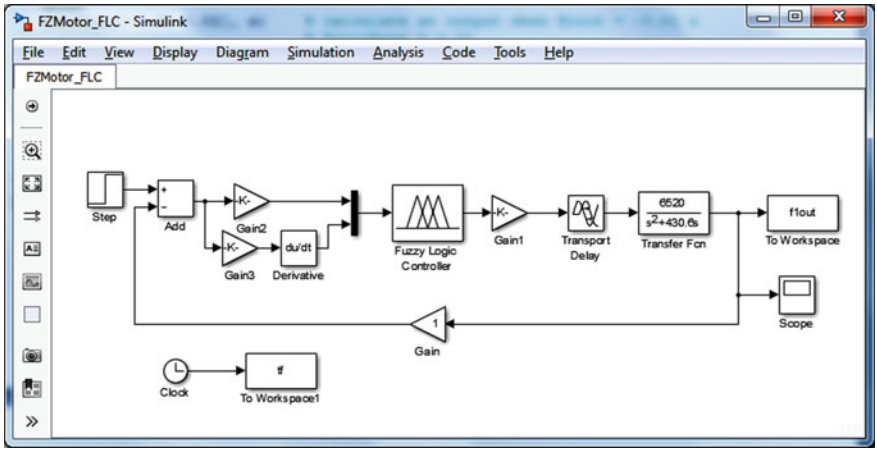


Fig. 7.31 The completed Simulink model FZMotor_FLC

When doing these dragging and placing control blocks, the following points should be paid attention to:

(1) Since Simulink categories and sorts all control blocks in the different related groups, one needs to go to different groups to find these control blocks used in this model.
(2) The **Fuzzy Logic Controller** function block is located in the **Fuzzy Logic Toolbox** group in the Simulink Library Browser.
(3) Since we also need to perform simulation study for the 5 ms time delay existed in this DC motor model, therefore a **Transport Delay** block is used and it is located in the **Continuous** group in the Simulink Library Browser. Double click on this block and set the delay time as 0.005, which is equivalent to 5 ms.

(4) The purpose of using two blocks, output **flout** and Clock **tf**, is to plot the step response of this fuzzy logic control system later, thus we export both of them to the Workspace and will use them later. Three control gains are Gain2 (K_P), Gain3 (K_D), and Gain1 (K).

Before the simulation can be started, an important task one needs to do is to set up the path for the Fuzzy Logic Controller block in this model to connect it to the related FIS developed in the previous section. There are two ways to do this connection, but first one needs to double click on this Fuzzy Logic Controller block to open the **Function Block Parameters** dialog box.

(1) Enter the full name of the FIS you built in the previous section into the **FIS name** box. In our case, it is **FZMotor.fis**. One needs to use a pair of single quotation marks to enclose this name. Otherwise, the Simulink may not be able to recognize it. The complete name one needs to enter into this FIS name box is: **"FZMotor.fis"**. Click on the **OK** button to complete this setup. Sometimes a warning message may be displayed to indicate that the desired FIS is locked, just try to click the **OK** button again to solve this issue.
(2) Enter the name of the FIS you built in the previous section into the **FIS name** box. In our case, it is **FZMotor**. However, if you do this setup in this way, the prerequisite is that you should have already exported the FIS to the Workspace. If you have not done that export operation yet, open the FIS and export it to the Workspace by going to **File→Export→To Workspace** menu item. In this way, no extension and quotation mark are needed for this name.

Now go to **Simulation→Run** menu item to start this simulation process. Open the Scope by double clicking on the **Scope** icon to check the simulation result. When this Scope block opens, you can use **View→Configuration Properties** menu item to set up and adjust the view for the step response. The following parameters or properties are recommended to setup:

- **Main** Menu: Setup **Sample time** as 0.001 s.
- **Time** Menu: Select **Time units** as Seconds.
- **Display** Menu: Setup the **Y-limits (Maximum)** as 1.2.
- Keep all other settings as default with no changes.

Now in order to get the optimal simulation result, you need to change or modify different gains. The tested optimal gains can be obtained by tuning these parameters and run the simulation after each modification. A point to be noted is that each time after any gain is changed, one needs to save any change by going to **File→Save** menu item to do this saving.

The obtained optimal control gains are:

- Gain1: 0.05 (K),
- Gain2: 0.037 (K_P), and
- Gain3: 0.00007 (K_D).

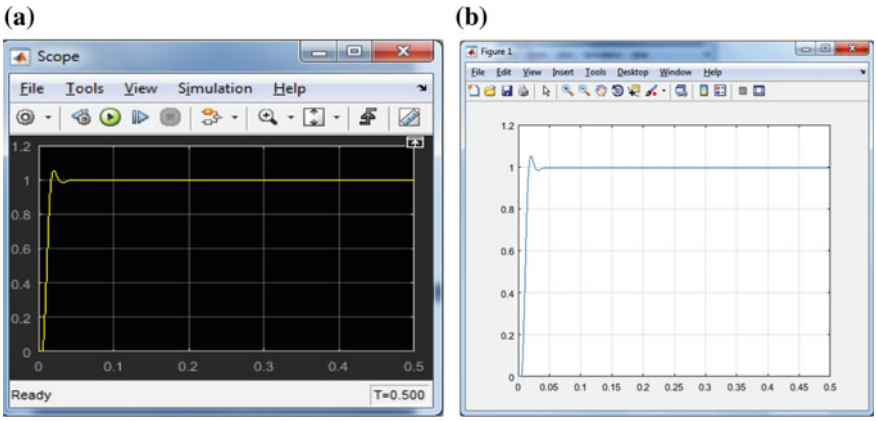


Fig. 7.32 The simulated step response for the DC motor system

The simulated step response is shown in Fig. 7.32a, b, respectively. In order to plot the step response with the data exported to the Workspace, one can use the following MATLAB commands in the Command window directly:

```
>> plot(tf.signals.values, flout.signals.values)
>> grid
```

Since two variables, **flout** and **tf**, are exported to the Workspace in the object format, thus one cannot directly use those objects to do this plotting. Instead, one needs to access and use real values of those objects with the dot operator as shown above.

7.11 Comparison Between the PID Controller and the Fuzzy Logic Controller

We have discussed the PID control systems in detailed in Chap. 5 and fuzzy logic control systems in this Chapter. Comparatively speaking, the design of PID controller depends more on the system model, and on the other hand, the design of the FLC depends more on the experience and measured data about the system.

From the point of view of controllers, the PID controller is very powerful for linear systems, while the FLC is suitable for both linear and nonlinear systems.

As far as the response speed is concerned, the FLC has the advantage of advance in the phase for the control system. So it can overcome the time delay existed in the system and can improve the transient response and stability of the system.

For the point of view of tuning, it is necessary to redesign and retune the controller for the PID control if the system model is changed or the parameters of

the system are changed, which include the system identification and system simulation processes. However, for the FLC, the only things to be tuned are the membership functions and control gain factors when the system model is changed or the systematic parameters are changed greatly, and this tuning is relatively easier and more convenient.

The control parameters to be tuned for a PID controller and an FLC are shown in Table 7.6. The advantages and disadvantages for both controllers are also shown in this Table.

In order to make the fact clear that the FLC has a better performance than that of traditional PD, PI or PID controllers, both simulation and experiment study is supplied in different sections. Both study results are provided to illustrate the advantage of the FLC over the classical PID control strategy.

7.11.1 Simulation Studies

The simulation study is provided in this section and the experimental study will be introduced in next section. The simulation process is divided into four Sections.

(1) A control system containing the time delay is discussed by using both FLC and PID controllers in Sect. 7.11.1.1.
(2) A control system including both noises and time delay is analyzed and compared by using an FLC and PID controller in section, 7.11.1.2.
(3) A comparison study for an uncertain system model applied in the FLC and PID control is presented in Sect. 7.11.1.3.
(4) Finally, a comparison study for FLCs that use different output curves as the output MFs is provided in Sect. 7.11.1.4.

Table 7.6 The comparison between PID control and FLC

Items	PID control	Fuzzy logic control
Model-based	Yes	No
Stability	Good	Better
Nonlinear control	No	Yes
Advance in phase	No	Yes
Response speed	Fast	Faster
Tuning process	Complicated	Simple
Robust control	No	Yes

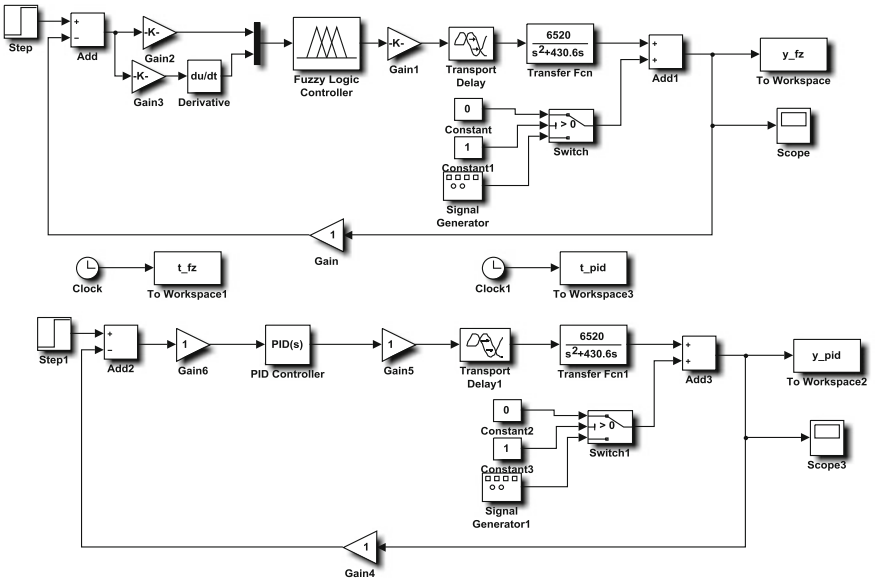


Fig. 7.33 The simulated system for a system with time delay (FLC and PID control)

7.11.1.1 A System with Propagation Time Delay

In this section, both regular and longer time delay are studied to compare the performances of an FLC and a PID control system. The block diagram for this simulation is shown in Fig. 7.33.

In this case, a system time delay (0.005 s = 5 ms) is set up for the transport delay block in the simulation. The comparison between a tuned optimal FLC and a tuned optimal PID controller for the system model with a time delay of 5 ms is simulated and studied.

Create a new MATLAB Simulink model named **PID_FLC_SFT.slx** and add blocks as shown in Fig. 7.33 into this model. The following blocks are important and need to be paid attention to:

- The purpose of two switches, **Switch** and **Switch 1**, are used to select and control the simulated noise and disturbance. The threshold value is set to 0.
- Two signal generator blocks, **Signal Generator** and **Signal Generator 1**, are used to generate different simulated noises.
- The **Transport Delay** blocks represent a pure time delay in the system model (0.005 s).
- Both input signals for the FLC and PID controllers are step input signals.
- Two clocks, **t_fz** and **t_pid**, are used to record the time elapsed and will be sent to Workspace for the graphic plotting and display later.
- Similarly, the output, **yf** and **y_pid** are sent to the Workspace for the graphic display later.

- Two scopes, **Scope** and **Scope3**, are used to monitor and record the output results in real time for the control systems.
- Three PID control parameters or control gains, $\boldsymbol{K_P}$, $\boldsymbol{K_I}$, and $\boldsymbol{K_D}$ can be obtained from a lab project, **Lab Project** 5.2 in Chap. 5. The optimal control gains are: $K_P = 10$, $K_I = 317$, and $K_D = 0.02$. We can enter them into the **PID Controller** block and use these values as the initial control gains to start our simulation process.
- The block **Fuzzy Logic Controller** should be connected to a new fuzzy inference system **FZMotor_SFT.fis** that is a new FIS built by using the shifted control rules shown in Table 7.5. One needs to first build that FIS and export it to the Workspace, and then sets up a connection to this block to perform the simulation.
- A system time delay of 5 ms is set up for the transport delay block in the simulation. During the simulation process, one needs to tune and change three control gains, $\boldsymbol{K_P}$, $\boldsymbol{K_I}$, and $\boldsymbol{K_D}$, to obtain the optimal step response. The optimal control parameters are: $\boldsymbol{K_P} = 5$, $\boldsymbol{K_I} = 1$, and $\boldsymbol{K_D} = 0.01$. Three optimal control gains for the fuzzy logic controller (FLC) are: Gain1 = 0.05 ($\boldsymbol{K}$), Gain2 = 0.037 ($\boldsymbol{K_P}$), Gain3 = 0.00007 ($\boldsymbol{K_D}$). The comparison between the optimal FLC and the optimal PID controllers for a step input is shown in Fig. 7.34.

It can be found from Fig. 7.34 that the step response of the system with FLC is better than that of the PID controller. Especially for the transient response, the settling time for the system with FLC is about 0.04 s. But the settling time for the PID controller is about 0.06 s. Also, the output of the control system with FLC has smaller overshoot than that of PID controller. Figure 7.35 shows the simulation results for both FLC and PID controls with different time delays, from 100 to 800 ms.

From the simulated result shown in Fig. 7.35, it can be seen that the performance of FLC is better than that of PID control in the transient response, and a shorten settling time can be obtained compared with that of the PID controller. In more

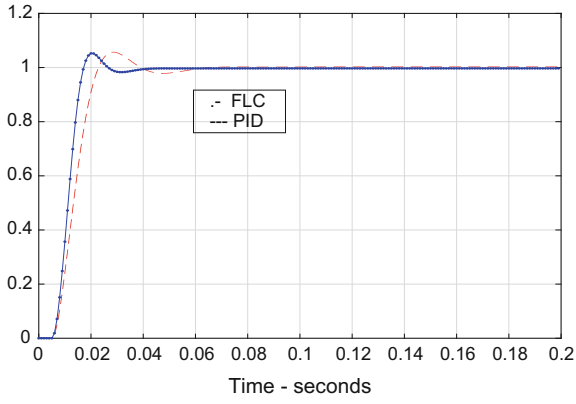
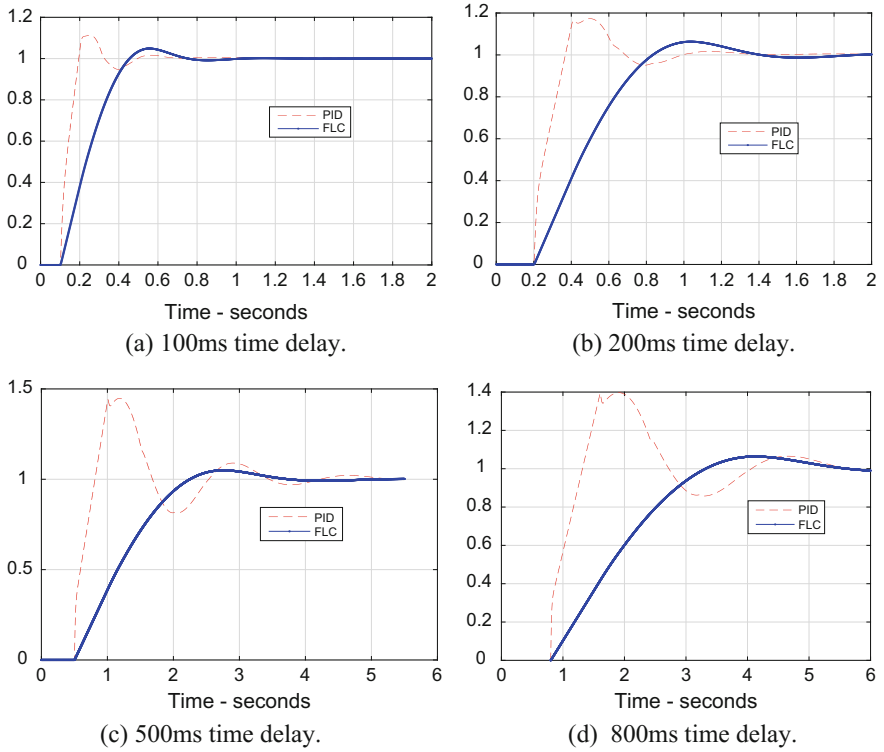


Fig. 7.34 Comparison for FLC and PID controller for 5 ms time delay



(a) 100ms time delay. (b) 200ms time delay.

(c) 500ms time delay. (d) 800ms time delay.

Fig. 7.35 Simulated results for FLC and PID with different time delays

serious situation for longer time delay, the PID control system may become oscillation or even unstable.

The optimal parameters or control gains for FLC and PID controllers under the different time delays are shown in Table 7.7. When tuning the FLC and the PID controller for the different time delay during the simulation process shown in Fig. 7.33, these optimal parameters should be entered into the Fuzzy Logic Controller and the PID Controller blocks in Fig. 7.33 to enable these comparisons to be fair and make sense.

Table 7.7 The optimal control parameters for the FLC and the PID controller

Time delay (ms)	Fuzzy logic controller (FLC)			PID controller		
	Gain1	Gain2	Gain3	*P*	*I*	*D*
100	0.0015	0.04	0.00005	0.5	0.02	0.02
200	0.0008	0.04	0.00005	0.28	0.005	0.02
500	0.0003	0.04	0.00005	0.15	0.002	0.02
800	0.0002	0.04	0.00005	0.09	0.001	0.02

7.11.1.2 A System with Noises and Time Delays

Because of the space limitations, starting from this section, we will use some simulation results directly to illustrate the advantages of using an FLC to effectively reduce some noises and disturbances existed in some control systems. Also, a comparison is made between an FLC and a PID controller to show the better performances of an FLC than that of a PID controller under those noises and time delays environments.

The FLC used for simulations performed in this section contains two lookup tables, a coarse and a fine lookup table, to improve both transient and steady-state performances. As we discussed in Sect. 7.6, the advantage of using two lookup tables is to handle the transient response with a coarse lookup table to get instant response for those big errors and error rates. While the fine lookup table can be used to process small steady-state errors to get fine controls for small errors and error rates.

For the simulation purpose, only random noises and a fixed length time delay (5 ms) are used in this section. Of course, different other noises, such as sinusoidal, triangle, and saw-form waveforms, can also be simulated and the similar results can be obtained.

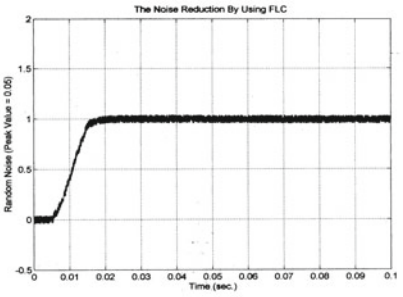
A sequence of random noise is applied for the simulation study for the comparison of the noise reduction between the FLC and PID controller. The simulated results for a step input are shown in Fig. 7.36.

The peak value of the random noise varies from 0.05 to 0.5, and this noise signal works as a noise introduced by the electrical coupling effect on the motor driver. From the output results of the FLC and PID control, it can be found that a 30% noise reduction can be obtained by using the FLC than the PID controller.

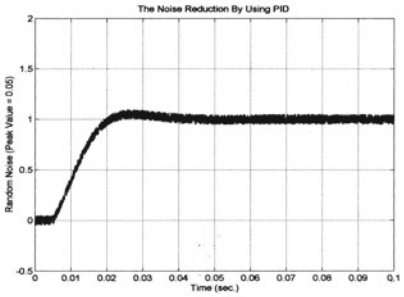
Generally, most FLC has a better performance in the noise reduction compared with the PID controller. This noise reduction depends on the distribution and the shape of the membership functions.

7.11.1.3 A System with Uncertain System Model

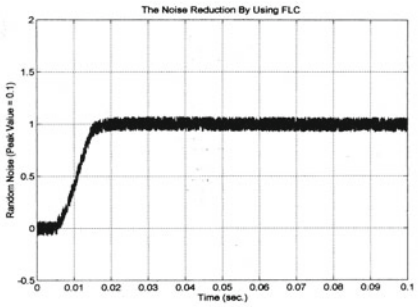
Because some uncertainties existed in a system model can be identified by experimental methods, it is necessary to make analysis based on the uncertain system models developed by using the MATLAB® Identification Toolbox or other methods for some control systems. Following is a comparison between a PID controller and an FLC based on the system model identified using the toolbox. In general, the FLC has a better ability of controlling an uncertain system model because it is a nonlinear controller, and the output of the lookup table is not based on the point-to-point control principle which means that one output point is associated with one input point only. Instead of using this traditional control method, the output of the FLC is based on the membership function and control rules. This means that the output of the FLC is not only determined by the current input point (input error, input error rate, or input error integration) but also determined by the



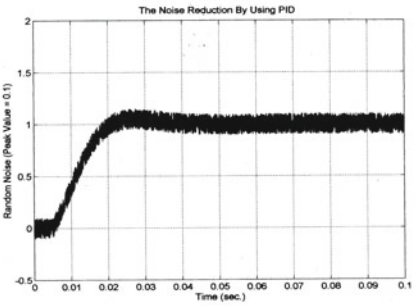
(a) FLC with noise peak = 0.1



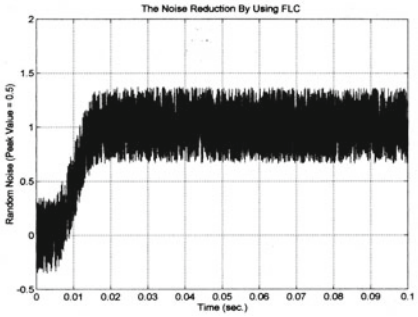
(b) PID Controller with noise peak = 0.1



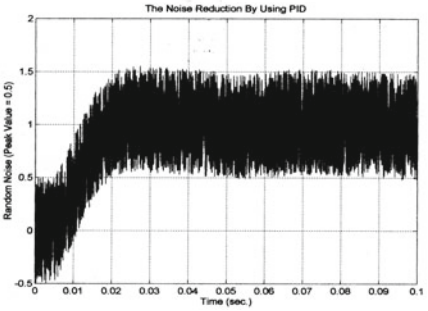
(c) FLC with noise peak = 0.2



(d) PID controller with noise peak = 0.2



(e) FLC with noise peak = 0.5



(f) PID controller with noise peak = 0.5

Fig. 7.36 The simulated results of the noise reduction for FLC and PID controller

neighbor error and error rate (in which degree the neighboring input error and error rate affect the current point). In this way, the output of the FLC depends not only on the current input point (error and error rate) but also on a range of the neighboring input points. From this point of view, an FLC can still realize its control even the system model has some degree of change or uncertainty as long as this change or uncertainty is kept within a limited range which is under the control of the FLC. This control depends on the actual system model. In this section, a comparison

study is performed to compare a PID controller and an FLC with an uncertain system model that is given based on a system model of a DC motor, which is

$$G(s) = 10.68\frac{s+610.5}{s\,(s+430.6)}\mathrm{e}^{-0.005s} \tag{7.30}$$

In order to introduce some uncertainty to the system, first the zero of this transfer function ($z = -610.5$) is modified from 5, 10 to 50%, and the pole is fixed. Then, the pole $s = -430.6$ will be modified from 10, 30 to 50% and the zero is kept unchanged.

A list of modified system models and related change-percentage-rate on zero and pole is given in Table 7.8. The model shown in Eq. (7.30) is the original system models identified by using the MATLAB® System Identification Toolbox for a DC motor. A simplified model is used when designing the conventional PID controllers in Chap. 5.

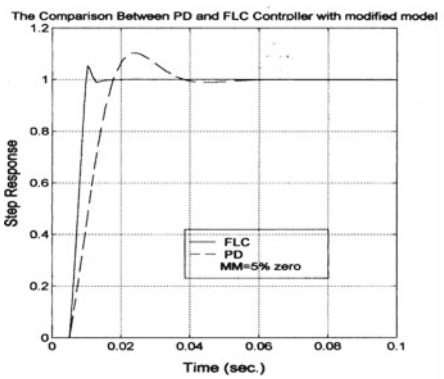
The simulated results are shown in Figs. 7.37 and 7.38 for the modified system models because of some uncertainties existed in the original system model.

It can be found from these comparisons that both the PID controller and the FLC can still work well for the 5 and 10% zero changes. The only difference of the control performance between the original and changed system model is: for the PID controller, a more serious overshoot is occurred (increased about 10% compared with the original model) and the settling time is also longer. For the FLC, the overshoot and settling time keep the same as that of the original model but a steady-state error was increased (about 1%). The FLC is a better controller compared with the PID controller if this small steady-state error is acceptable for the system.

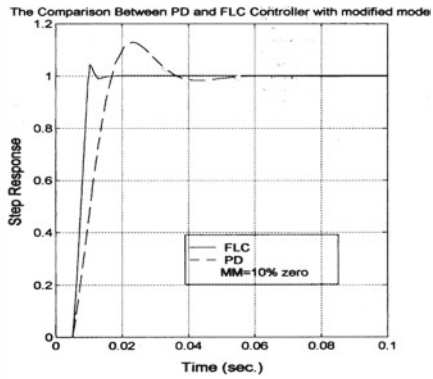
The situation becomes worse when the zero changes to 50% for both the PID controller and the FLC. For the PID controller, a very serious overshoot and a much longer settling time occurs (an increase of about 35% in overshoot and 50% in settling time compared with the original system model). For the FLC, the transient response keeps the same as before but a greater steady-state error is introduced (an increase of about 5% compared with the original situation). So the standard used to evaluate which controller is a better one depends on the actual situation and different control objectives. But comparably speaking, the FLC has a better

Table 7.8 Modified system models and related change rate on zero and pole

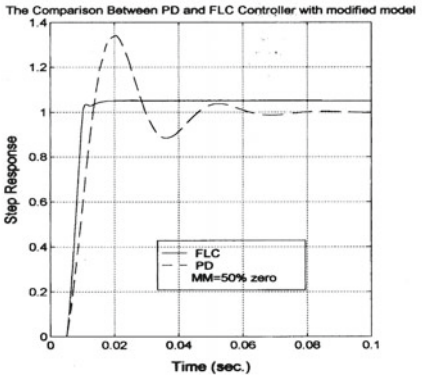
Zero/pole	Change rate (%)	Modified system model
Zero	5	$10.68\frac{s+641.0}{s(s+430.6)}\mathrm{e}^{-0.005\,s}$
Zero	10	$10.68\frac{s+671.5}{s(s+430.6)}\mathrm{e}^{-0.005\,s}$
Zero	50	$10.68\frac{s+915.8}{s(s+430.6)}\mathrm{e}^{-0.005\,s}$
Pole	10	$10.68\frac{s+610.5}{s(s+473.7)}\mathrm{e}^{-0.005\,s}$
Pole	30	$10.68\frac{s+610.5}{s(s+301.4)}\mathrm{e}^{-0.005\,s}$
Pole	50	$10.68\frac{s+610.5}{s(s+215.3)}\mathrm{e}^{-0.005\,s}$



(a) 5% uncertain on zero



(b) 10% uncertain on zero

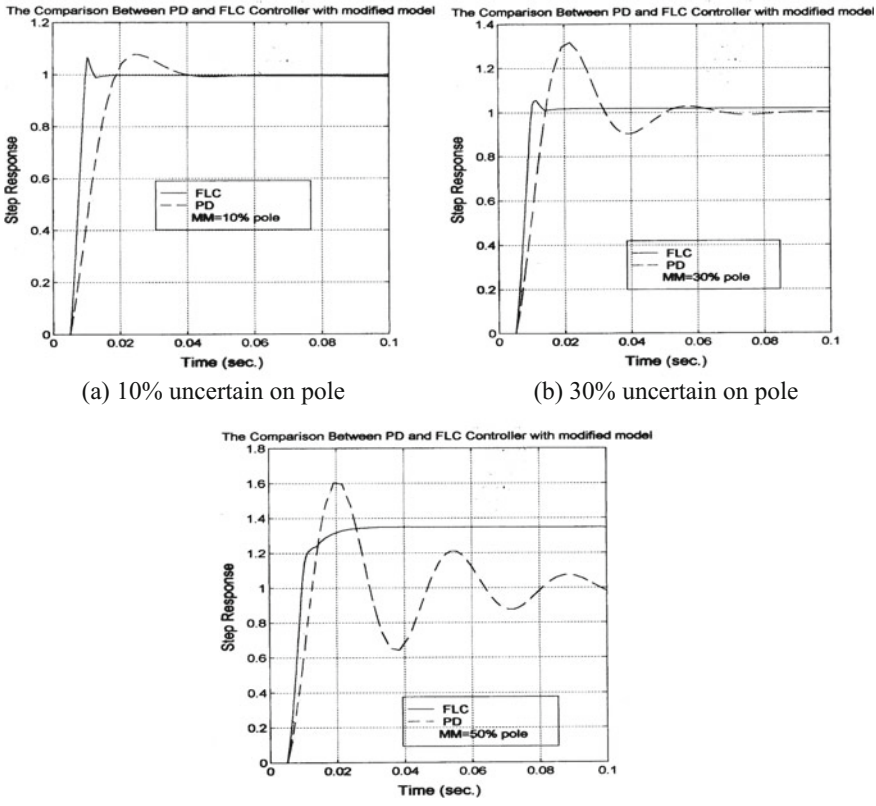


(c) 50% uncertain on zero

Fig. 7.37 Comparison of PID control and FLC with uncertain zero

performance if this steady-state error is allowable for a certain control system. A possible solution for this steady-state error is to use a fine lookup table to limit the steady-state error to a small range and make it closed to zero as a PID controller did. But from the point of view of engineering, it is good enough for the purpose of engineering applications.

For the pole's situation, the PID controller and the FLC can work well under 10% uncertain system model. A small difference is that the PID controller has a relatively larger overshoot and longer settling time, but the FLC has a little greater steady-state error (1%). For the 30% uncertain model, it is only an extension of the situation of 10% uncertainty. That is: the PID controller has a serious overshoot (an increase of 30%) and a longer settling time (an increase of 50%). The FLC has a greater steady-state error (about 3%). In a 50% uncertain system model, the PID controller has a more serious overshoot (an increase of 60%) and a much longer settling time (about 150 ms). The FLC has a larger steady-state error (about 35%).



(a) 10% uncertain on pole (b) 30% uncertain on pole

(c) 50% uncertain on pole

Fig. 7.38 Comparison of PID control and FLC with uncertain pole

Summarily speaking, a fair uncertainty of a system model whose uncertainty is about $20 \sim 30\%$ is acceptable in most general applications, and for this level of uncertainty of the system model, the FLC has a better control ability and performance than that of the PID controller. Note that the conclusion derived above is only based on the analysis for some simulation studies and no general directive purpose. This is because of the specialty of the fuzzy logic controller which is developed by using the human knowledge and experience.

Generally, different FLC has different ability in noise reduction, which depends upon the distribution and the shape of the membership function as well as the control range. In a certain control range, this ability will be better if more membership functions are selected (big density), and in the same situation, Gauss distributed membership function has better performance than that of triangle distributed membership function.

7.11.1.4 The Comparison Between Different Output Curves Used by FLC

A comparison between different output curves used by an FLC is provided in this section to illustrate the effects of different output control curves to the system. Figure 7.39 shows the simulation block diagram for the two kinds of output curves: Gaussian and Triangle.

In Fig. 7.39, four fuzzy logic controller blocks, Fuzzy Logic Controller and Fuzzy Logic Controller 1 are the coarse and fine FLC, in which both output curves are the form of Gaussian. Fuzzy Logic Controller 3 and Fuzzy Logic Controller 4 represent the coarse and fine FLC with the triangle curves. The Signal Generator is used to produce the different noises.

Figure 7.40 shows the simulated results for the FLC with two kinds of curves, Gaussian and triangle, without any noise. Figure 7.41 shows the simulated results for the FLC with two kinds of curves under the random noise (noise power $= 0.1_{pp}$).

It can be found from these results that there is no difference between the performance of the FLC when it used either Gaussian or triangle curve as the output curve. The reason for that is: for our case, the membership function for the input error, error rate, and output are fine enough to ensure that the intervals between the neighboring input error or error rate are so small that it can be considered as almost continuous function. This means that in a small range of input error and error rate, a total of 7 fuzzification levels were applied for both input range (−0.1 to 0.1 V for error and −0.06 to 0.06 V for error rate), and a total 49 control rules were designed. So a $15 \times 15 = 225$ elements in the lookup table can be used for the output

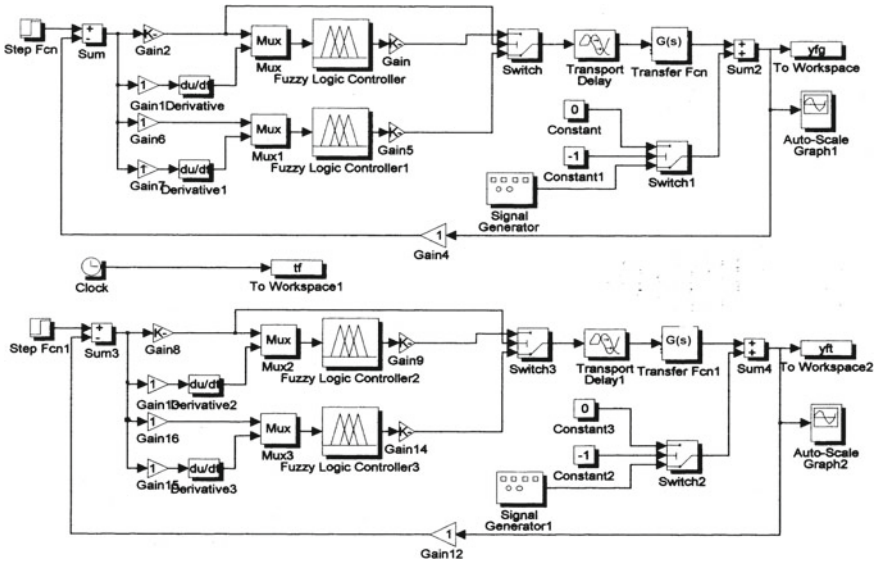
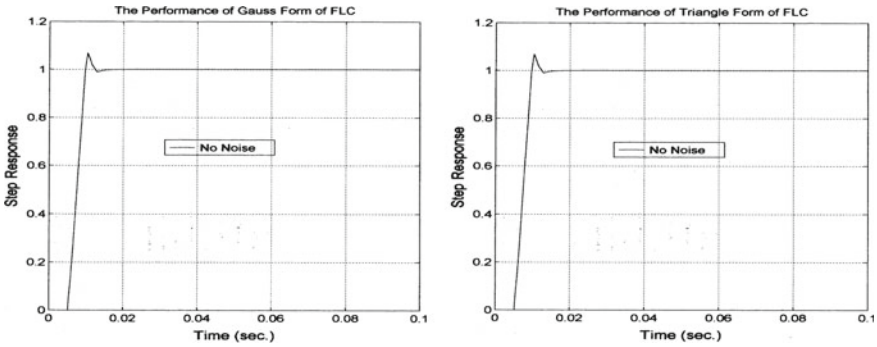
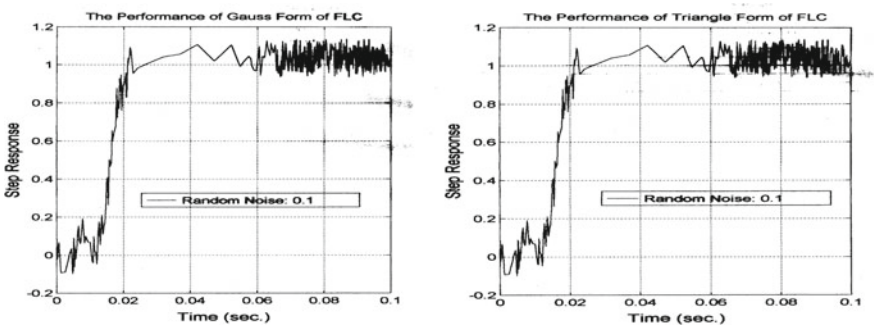


Fig. 7.39 The simulation block diagram for two kinds of output curves used by FLC



(a) Gaussian waveform as curve. (b) Triangle waveform as curve.

Fig. 7.40 The simulated results for two kinds of output curves used by FLC



(a) Gaussian waveform as curve. (b) Triangle waveform as curve.

Fig. 7.41 The simulated result of two output curves of the FLC with random noise

purpose. Just because of this small enough interval between the neighboring errors and error rates, as well as the output of the FLC, it is very difficult to find the difference between the Gaussian and triangle output curves used by the FLC. The situation will be different if few fuzzification levels or control rules are adopted. This difference would be significant if less membership functions were used in a relatively large control range.

7.11.2 Experimental Studies

The experimental study is concentrated on the noise reductions for the fuzzy logic control systems. This section can be divided into two parts: The general noise reduction and specified noise reduction.

7.11.2.1 Comparison of the FLC and PID Controller in General Noise Suppression

The experimental study results of the noise and disturbance reduction for both FLC and PID controller are given in this section. The comparison between the FLC and PID controller in noise suppression is provided in order to illustrate that the FLC has a better ability in overcoming noise and disturbance.

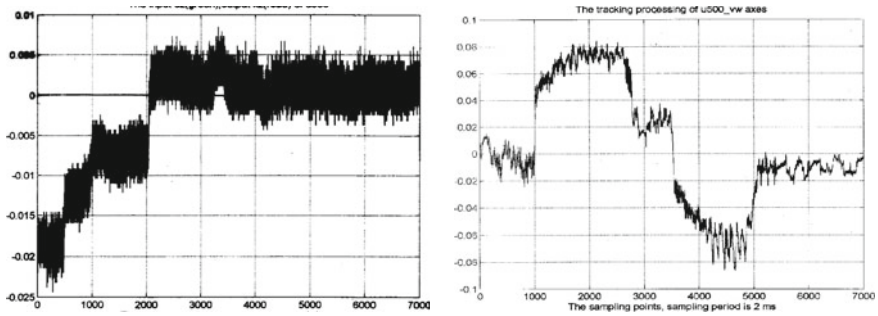
Figure 7.42a, b show the steady-state errors of an FLC and a PID controller for the identified DC motor control system whose transfer function is given in Eq. (7.30). From Fig. 7.42, it can be seen that the average steady-state error of the PID controller (Fig. 7.42b) is about 0.018 V, comparing with the steady-state error of the FLC (Fig. 7.42a), which is about 0.005 V. The former is about threefold greater than that of the latter.

It is obvious that the steady-state error of the system is significantly reduced after using a fine fuzzy logic control strategy. The steady-state error of the PID controller for the same plant is much greater compared with that of an FLC used for the same plant.

Hence, based on the discussion above, the following points can be concluded:

- FLC is a robust controller and can be used to reduce some noises and overcome some disturbances for most general noises and disturbances, such as random noises.
- FLC can realize very accurate control form some specified control plants or processes. The steady-state error of the FLC can be much smaller compared with that obtained by using PID controllers.

Relatively speaking, an FLC can perform better controls for both linear and nonlinear systems in our world, and therefore it is a powerful control strategy.



(a) Steady state error of the FLC. (b) Steady state error of the PID controller

Fig. 7.42 The comparison in noise reduction on an FLC and a PID controller

7.11.2.2 Spectrum Analysis of the Specified Noise Reduction by Using FLC

As we discussed in the above section, the FLC has a better ability to overcome the noise and disturbance for some control systems. In addition to those general noises and disturbances, the FLC has special functionality to reduce some specified noise in a selected frequency range. In order to illustrate this, a spectrum analysis for the noise and disturbance existed in a DC motor control system is provided in this section.

Two kinds of noise sources are existed in this DC motor control system:

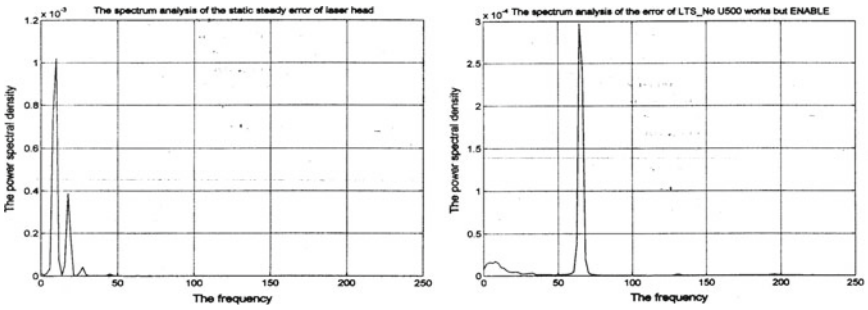
(1) Background noise with a frequency range between 0 and 30 Hz.
(2) Power supply noise around 60 Hz.

The power supply noise is coming from some electrical magnetic equipment or devices, such as transformers or coils, and this kind of noise is distributed via workspace or space and is very difficult to be removed and filtered.

The power spectral or spectrum of two kinds of noise sources are shown in Fig. 7.43a, b, respectively. The background noise has a very low-frequency range, from 0 to 30 Hz, but the power supply noise is around 60 Hz, which is the typical power frequency in US.

Figure 7.44 shows the power spectral density graph of the steady-state error for the PID control system. It can be found from Fig. 7.44 that the noise of the system is a combination of the noises coming from the background and the transformer in the DC motor amplifier (refer to Fig. 7.43). The bandwidth of the system noise is about 75 Hz. It is clear that a pure PID control system cannot overcome this noise or disturbance effectively.

Figure 7.45 shows the steady-state error of the system for a coarse fuzzy logic control strategy. Compared with Fig. 7.44, which is the PID control system, the spectrum of the noise is changed. The second noise peak whose frequency component is about 30 Hz under the PID controller is disappeared in the FLC system. Also, the most obvious point is that the noise component around 60 Hz is



(a) Background noise spectrum (b) Power supply noise spectrum

Fig. 7.43 The power spectral density or spectrum of noise sources

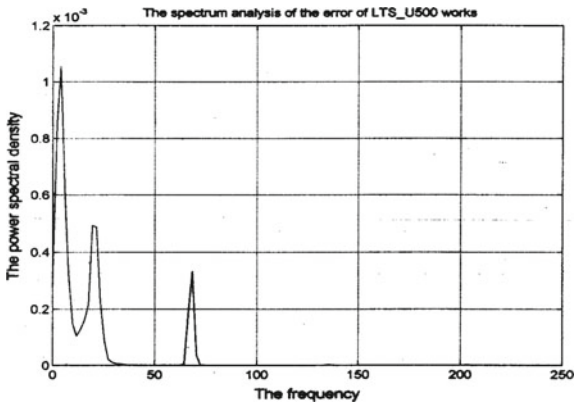


Fig. 7.44 Power spectral density of the steady-state error for the PID controller

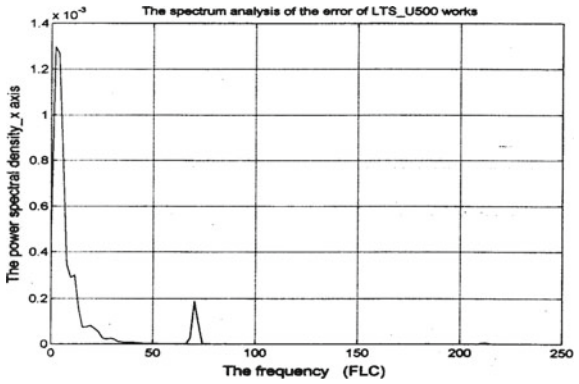


Fig. 7.45 Power spectral density of the noise for a coarse FLC

significantly attenuated, which now is about half of the original power spectral density. So it is evidence that after using the FLC, the system noise was greatly attenuated (about 50%) and the signal-to-noise ratio was significantly improved. This observation is identical with the analysis we made earlier in the time domain (Fig. 7.42).

Figure 7.46 shows a comparison between a coarse fuzzy and a fine fuzzy logic controller in noise reduction, or a noise distribution for the coarse fuzzy and fine fuzzy control strategies.

For the coarse fuzzy control, the power spectral density of the noise is relatively greater and the range of noise distribution is mainly below 50 Hz, which is similar to the graph shown in Fig. 7.45. For the fine fuzzy control, the noise is much smaller (almost 0) than that of coarse FLC did in the range of low frequency (under 50 Hz), and the only noise is distributed around 60 Hz with a relatively lower power spectral density. Therefore from this comparison, it is obvious that the low-frequency noise is significantly attenuated after using the fine fuzzy controller.

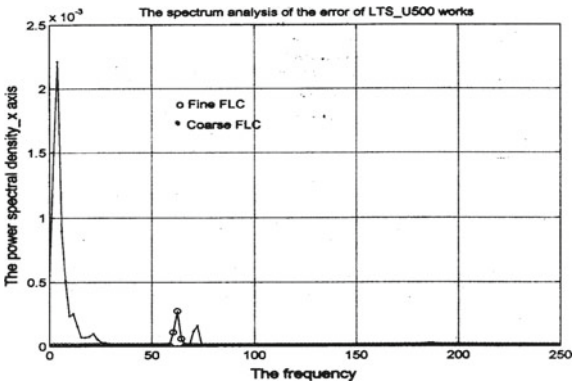


Fig. 7.46 Comparison on a coarse and a fine fuzzy control in noise reduction

Especially, the low-frequency noise (below 50 Hz), which is introduced by the background and environments, is effectively reduced by the fine fuzzy logic controller.

An FLC has a special function of low-pass filter because of its "center-of-gravity" algorithm in calculating the output of the FLC. A simulation for the low-pass function of the PID and the FLC is shown in Fig. 7.47. A coarse and a fine FLC, as well as a PID controller, are presented to be used to compare the low-pass function of the FLC and the PID controller with a sinusoid waveform as the inputs.

A conclusion can be obtained based on the discussion and analysis above, which is: for an ideal system model, the performance of both an FLC and a PID controller is identical. But for the system model including time delays and noises, a better performance can be obtained by using the FLC than that of the PID controllers. Moreover, for a system model which including both time delay and noise

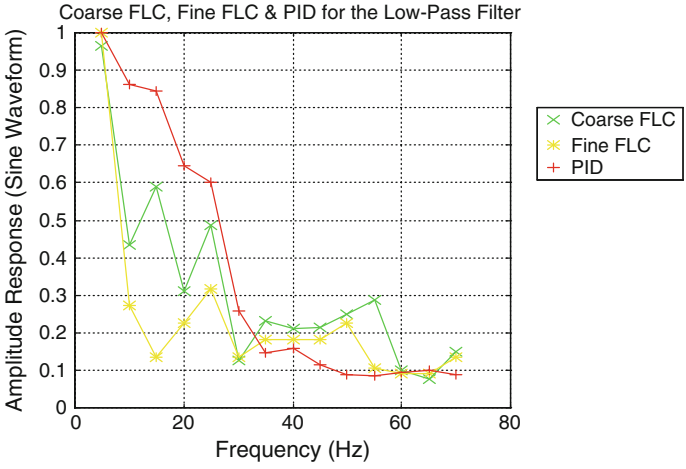


Fig. 7.47 Low-pass property of an FLC and a PID controllers

disturbances, or including some uncertainties, a better control performance can be obtained by using the FLC. It can be concluded that the FLC has a better performance for a control system that includes both time delay and noise disturbance. The latter is more closed to the actual control situation in the real world.

7.12 Summary

The fuzzy logic control strategy, algorithm, and technology are introduced and discussed in this chapter in detail. Starting from Sect. 7.2, all different sets, including the classical sets and fuzzy sets, are discussed in detail with deep and complete illustrations. The idea of fuzzifications and membership functions is introduced and presented in Sect. 7.3. Fuzzy control rules design and implementations are provided in Sect. 7.4. The output of the fuzzy controller, defuzzifications and lookup tables, is presented in Sect. 7.5. All of these discussions and introductions belong to fundamentals and basic knowledge for implementing and applying fuzzy control technology on most control systems, either linear or nonlinear systems.

Section 7.6 provides an introduction to the architecture of a fuzzy logic controller. All fuzzy controllers can be categorized and mapped to either a PD or a PI controller in most real-world applications. A detailed mapping relationship between an FLC and a PD or a PI controller is given by using functional block diagram in this section.

Sections 7.7–7.11 are about the implementations of different fuzzy logic control technologies on different control objectives, mainly on DC motors. A powerful tool, MATLAB® Fuzzy Logic Toolbox and related Fuzzy Logic Toolbox Graphical User Interface Tool (FLTGUIT), is introduced and presented in Sect. 7.8. Two kinds of typical applications; applications developed by using the FLTGUIT, and applications built by using the Fuzzy Logic Toolbox functions, are discussed in detail with real examples.

One of the most important and difficult topics in implementations of fuzzy logic control technology, how to effectively tune a fuzzy logic controller, is presented in Sect. 7.9. Different tuning techniques for FLC are introduced and analyzed to provide a real recipe to help users to build and apply an FLC more efficiently and easily.

Implementations of FLC with MATLAB Simulink, another powerful tool in applying FLC, are discussed in Sect. 7.10. A detailed procedure of creating, designing, tuning, and simulating an FLC in the Simulink environment is provided step-by-step to help and enable users to follow up to design, tune and simulate their real FLC applications quickly and efficiently.

In the last section, a comparison between an FLC and a PID control system is provided to illustrate the advantages of using an FLC over a PID controller for general applications. Both simulation study and experimental study are provided to provide a clear picture that some better control performances can be obtained by

using an FLC than that of a PID controller. These comparisons include the following aspects:

- A system with propagation time delay, short and long time delay.
- A system with different noises and disturbances.
- A system with uncertain dynamic models.
- A spectrum analysis of general and specified noises applied on a DC motor system.

Homework and Labs

7.1 Using MATLAB® FLTGUIT to design an FLC for a heater-fan control system shown in Fig. 7.48. The room temperature is controlled by a heater-fan. If the room temperature is low or too cold, the heater should be turned on to heat the room. Otherwise, if the temperature is high or too hot, the heater-fan should be turned off to cool down the room. A thermometer works as a feedback sensor.
The definition of the membership functions for the input temperature and the output heater-fan rotating speed (Revolution Per Minute—RPM) are given in Fig. 7.49.
Four control rules are applied to this FIS:

(1) IF temp is Cold, THEN the fan_speed is High.
(2) IF temp is Cool, THEN the fan_speed is Med.
(3) IF temp is Warm, THEN the fan_speed is Low.
(4) IF temp is Hot, THEN the fan_speed is Zero.

7.2 Using MATLAB® FLTGUIT to redesign an FLC for a heater-fan control system shown in Fig. 7.50. A thermometer and a humidity detector work as the feedback sensors.
The MFs for both inputs and the output are shown in Fig. 7.51.
The following 12 control rules are implemented in this FIS:

1. IF Temp is *cold* and Humidity is *high*, THEN fan_speed is *high*.
2. IF Temp is *cool* and Humidity is *high*, THEN fan_speed is *med*.
3. IF Temp is *warm* and Humidity is *high*, THEN fan_speed is *low*.
4. IF Temp is *hot* and Humidity is *high*, THEN fan_speed is *zero*.
5. IF Temp is *cold* and Humidity is *med*, THEN fan_speed is *med*.
6. IF Temp is *cool* and Humidity is *med*, THEN fan_speed is *low*.
7. IF Temp is *warm* and Humidity is *med*, THEN fan_speed is *zero*.
8. IF Temp is *hot* and Humidity is *med*, THEN fan_speed is *zero*.
9. IF Temp is *cold* and Humidity is *low*, THEN fan_speed is *med*.

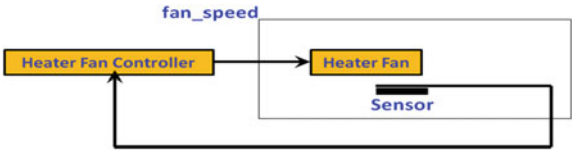


Fig. 7.48 A heater-fan control system

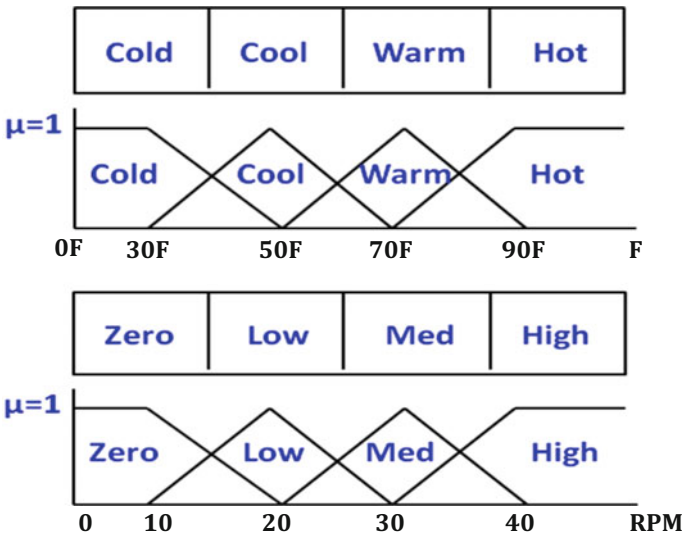


Fig. 7.49 Definition of the MF for the input temperature

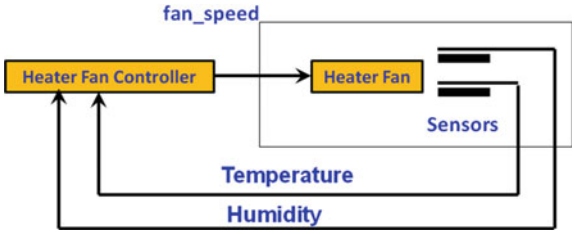


Fig. 7.50 The modified heater-fan control system

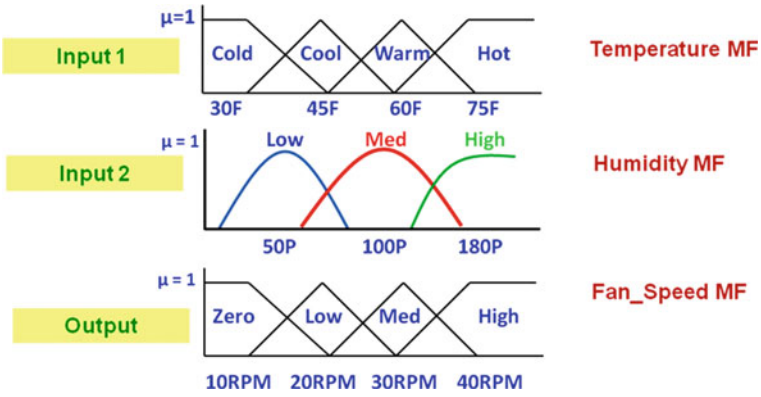


Fig. 7.51 MFs for the inputs and the output

10. IF Temp is *cool* and Humidity is *low*, THEN fan_speed is *low*.
11. IF Temp is *warm* and Humidity is *low*, THEN fan_speed is *zero*.
12. IF Temp is *hot* and Humidity is *low*, THEN fan_speed is *zero*.

7.3 Redo question 7.1 with MATLAB® Fuzzy Logic Toolbox functions to create a new FIS named **HW7_3.fis**.

7.4 Redo question 7.2 with MATLAB® Fuzzy Logic Toolbox functions to create a new FIS named **HW7_4.fis**.

Lab Projects—Fuzzy Logic Control Systems

Lab Project 7.1

Using MATLAB® FLTGUIT to design and build an FLC for a DC motor system that is identified in Sect. 4.7.5 in Chap. 4. The identified dynamic model of that DC motor is:

$$G(s) = \frac{3.776}{(1+0.56s)}e^{-0.09s} \cong \frac{6.74}{(s+1.79)}e^{-0.09s}$$

In Sect. 4.7.5, the input to the DC motor is the PWM values and the output is the motor encoder rotating speed (Pulse Per Revolution—**PPR**). The relationship between the motor encoder speed (PPR) and the control output PWM values is shown in Table 4.12 and repeated in Table 7.9. The ranges of inputs and output relationship are shown in Table 7.10.

Two inputs, the **PWM** value pwm and the pwm rate **pwm_rate**, work as inputs and the output is the motor encoder speed **e_sp**.

Table 7.9 The collected QEI encoder speed values and PWM values

Number	PWM values (u)	Encoder velocity values (es) PPR	
		Hexadecimal	Decimal
1	100	49	73
2	300	F4	244
3	500	15C	348
4	700	19F	415
5	900	1E0	480
6	1100	1EF	495
7	1300	1FA	506
8	1500	201	513
9	1700	206	518
10	1900	20C	524
12	2100	20D	525
13	2300	20C	524
14	2500	20E	526
15	2700	20E	526

The ranges of the input **pwm** should be 500 ~ 1900, and it can be divided into three MFs, small, medium, and high. The range for the **pwm_rate** is about 200 ~ 760, it can also be divided into three MFs, *small*, *medium*, and *large*. The range for the output **e_sp** is from 348 to 524 PPR, and it can be divided into three MFs, *slow*, *med* and *fast*. The waveforms of all MFs can be either triangle or Gaussian type. The reason we only use part of these data that are covered by a rectangular box is that only those data are closed to a linear equation between the PWM values and the motor encoder values, and easy to be controlled by any kind of controllers.

Totally one may need nine (9) control rules for this FIS. Try to build these rules based on the common senses and general control principles.

The feedback path for this closed-loop control system can be a coefficient that is used to convert the motor encoder speed to the PWM value, and this value can be obtained by averaging the ratio between the PWM values over motor encoder speed values, which is 2.45.

The DC motor model is only used later for the comparison between an FLC and a PID controller.

Lab Project 7.2
Redo lab project 7.1 with MATLAB® Fuzzy Logic Toolbox functions.

Lab Project 7.3
Using MATLAB® Simulink to perform a simulation study for the FLC obtained from lab project 7.1 to obtain an optimal step response for a step input.

Lab Project 7.4
For a given DC motor system plant with a transfer function as

$$G(s) = \frac{1}{s(s+1)}$$

Design an FLC for this control system using either FLTGUIT or Fuzzy Logic Toolbox functions method with the following given conditions:

(1) Two input variables, input error **err** and input error rate **err_rate**, work as input voltages to this system.
(2) One output is the motor rotating speed **m_sp** (Revolution Per Minute—RPM).
(3) The ranges for three variables, **err, err_rate** and **m_sp**, are shown in Table 7.10.

Table 7.10 Ranges of the input–output variables

Variables	Variables ranges	Units
Input error (err)	−0.2 to 0.2	V
Input error rate (err_rate)	−0.05 to 0.05	V
Motor output speed (m_sp)	−800 to 800	RPM

(4) For both **err** and **err_rate**, five MFs can be designed as: Small Negative (**SN**), Large Negative (**LN**), Zero (**ZE**) and Small Positive (**SP**), and Large Positive (**LP**). Gaussian curve should be used to get more accurate controls.
(5) For the output **m_sp**, five MFs can be considered as: Large Negative (**LN**), Small Negative (**SN**), Zero (**ZE**), Small Positive (**SP**), and Large Positive (**LP**). Gaussian curve can be used.
(6) Design 25 rules based on the common sense for this FLC system (refer to Table 7.5).

Using MATLAB® Simulink to perform a simulation study to compare the control performances (step response with no time delay) of this FLC and a PID controller designed in Sect. 5.3.6.2 in Chap. 5. The optimal PID control gains are: $P = 2.76$, $I = 1.13$, and $D = 1.69$. Use these gains as the initial PID values for the PID controller in that simulation study.

Lab Project 7.5
Redo lab project 7.4 with different propagation time delays for both controllers: (a) 10 ms, (b) 100 ms, and (c) 500 ms.

Chapter 8
Fuzzy Logic Controller Design in TM4C123G MCU System



In the last chapter, we provided very detailed discussions about the fuzzy logic control systems, including the fundamental theoretical knowledge of the fuzzy technologies and fuzzy inference systems, such as fuzzy sets, membership functions, control rules, lookup tables, and basic operations of a FIS. Following the theoretical introductions, a powerful tool, MATLAB® Fuzzy Logic Toolbox is introduced and discussed with some real examples. Also the MATLAB® Simulink, a powerful and useful simulation tool, is analyzed and discussed with a real DC motor control system. In this chapter, we will provide a more realistic implementation discussion about how to apply a FLC in a typical DC motor control system with a microcontroller programming project.

8.1 Introduction

A fuzzy logic controller belongs to a general-purpose controller that can handle both linear and nonlinear control systems. As we know, the PID controller can only handle the linear system or a linear time-invariant (LTI) system. However, all actual systems in our world are nonlinear systems. In order to develop a closely matched controller to control nonlinear systems, the fuzzy logic controller is a good candidate.

In order to effectively design and implement a FLC in a real closed-loop control system, the following operations or operational procedure steps should be followed:

1. Fuzzification process: Convert crisp variables to the fuzzy variables.
2. Control rule process: Design control rules.
3. Defuzzification process: Convert fuzzy variables back to crisp variables.

Figure 8.1 shows this fuzzy logic controller developing process.

To build a fuzzy logic controller (FLC), both input error $\boldsymbol{e}$ and error rate $\Delta\boldsymbol{e}$ need to be calculated in advance and both work as inputs to the FLC.

© Springer Nature Switzerland AG 2019

Y. Bai and Z. S. Roth, *Classical and Modern Controls with Microcontrollers*,
Advances in Industrial Control, https://doi.org/10.1007/978-3-030-01382-0_8

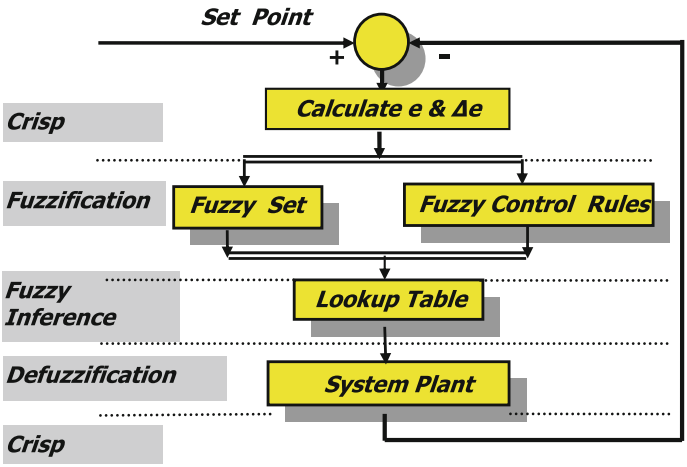


Fig. 8.1 The developing procedure of a fuzzy logic controller

Depending on the architecture of a FLC and the real applications, a Fuzzy-PI or a Fuzzy-PD controller can be designed and implemented for a specified system. Refer to Sect. 7.6 in Chap. 7 to get more details about the control-type selections.

8.2 Implementation of Fuzzy Logic Closed-Loop Control Systems

The first step of implementation of a FLC is to perform fuzzification process for both inputs and the output of a real control system or process.

8.2.1 The Fuzzification Process

The so-called fuzzification process is to convert the crisp variables to the corresponding fuzzy variables. The converted fuzzy variables are presented by a group of membership functions. Before doing this conversion, the ranges of the inputs (input error and error rate) and the output of the system should be determined first. In this way, the membership functions for the input error, the error rate, and the output of the system can be defined, respectively.

Recalled in Sect. 5.5 in Chap. 5, we discussed a PID controller designing and implementation process for a DC motor system **Mitsumi 448 PPR** with the transfer function.

$$G(s) = \frac{6.74}{s + 1.79} e^{-0.009s} \tag{8.1}$$

In that project, the input variable to the DC motor or the output of the PID controller is the PWM values. The output is the motor encoder. In this section, we will directly use the motor encoder values (feedback) as the input to this FLC system. The ranges of the input and the output of the system are given in Table 8.1.

These input errors, error rates, and output motor speeds are calculated based on the data collected from projects **CalibEncoder** (Sect. 4.7.4 in Chap. 4) and the **PID Control** (Sect. 5.5.2 in Chap. 5). For example, the minimum input encoder error is **−400**. The maximum input error rate is **179 − 49 = 130**, and the range of the motor PWM output values are **−2000 ~ 2000** (50%). A symmetry data structure is used. Although the **−2000** motor PWM value can never exist, for the fuzzy system design, a symmetry structure is recommended and applied for the fuzzy controller.

To represent these input and output variables using fuzzy sets, a set of linguistic variables is chosen to represent a five-*degree* of error, five-*degree* of error rate, and five-*degree* of control output. Membership functions are constructed to represent the input and output in which degree belongs to a different membership set or linguistic variable set.

The linguistic variables are defined as follows:

Inputs: LN—Large Negative, LP—Large Positive, SN—Small Negative, SP—Small Positive, ZE—Zero
Output: LN—Large Negative, LP—Large Positive, SN—Small Negative, SP—Small Positive, ZE—Zero

The membership functions of the input error, the error rate, and the controller output are shown in Fig. 8.2. The units for all of these variables are PPR and PWM value.

The MATLAB® Fuzzy Logic Toolbox™ is used to build and develop this fuzzy controller. Both inputs, error and error rate, used the Gaussian waveform as the shape of the membership functions (MFs). But the output **PWM** values used the triangular form as the shape of its MF.

Table 8.1 The range of the input error, error rate, and the output

Variable	Range of the variable	Units
Input error e	−400 to 400	PPR
Input error rate Δe	−130 to 130	PPR
Motor PWM value	−2000 to 2000	PWM

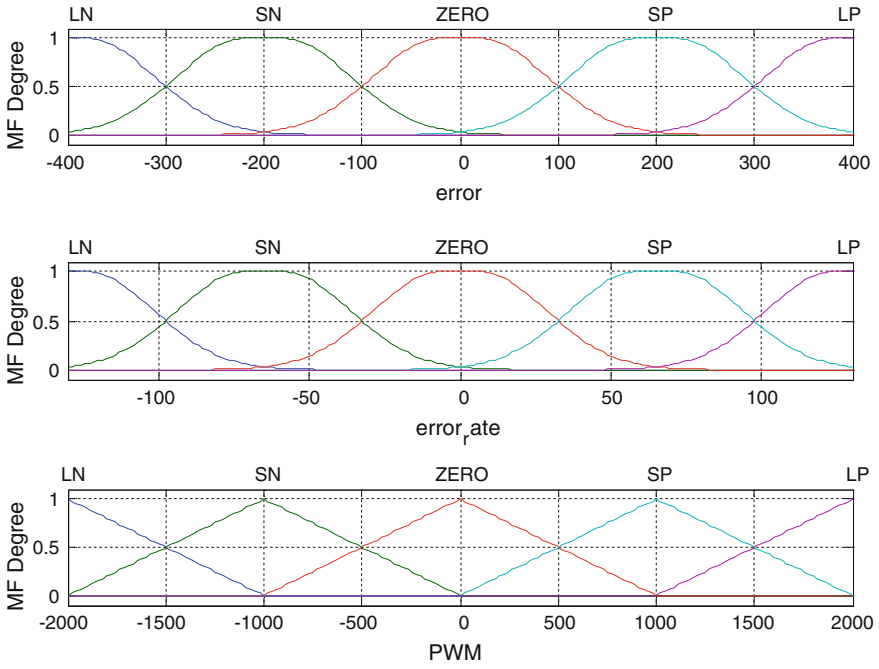


Fig. 8.2 Membership functions for input error, error rate, and the output

8.2.2 Design of Control Rules

Totally 25 control rules are designed for this fuzzy controller, as shown in Fig. 8.3.

These control rules are designed based on the common sense for a general control strategy by using the MATLAB® Fuzzy Logic Toolbox™. Go to the **Edit| Rules** menu to open the control Rule Editor wizard to build these rules.

8.2.3 The Defuzzification Process

To perform real-time control for any control system, a crisp control output is necessary. Therefore, the fuzzy outputs must be converted back to the crisp outputs by using the defuzzification process.

The core of the defuzzification process is the so-called Center-Of-Gravity (**COG**) method. Assume that u_i is the membership function, U_i the universe of discourse and m the number of contributions, the traditional output of the fuzzy inference system can be represented as

1. If (error is LN) and (error_rate is LN) then (PWM is LP) (1)
2. If (error is LN) and (error_rate is SN) then (PWM is LP) (1)
3. If (error is LN) and (error_rate is ZERO) then (PWM is SP) (1)
4. If (error is LN) and (error_rate is SP) then (PWM is SP) (1)
5. If (error is LN) and (error_rate is LP) then (PWM is ZERO) (1)
6. If (error is SN) and (error_rate is LN) then (PWM is LP) (1)
7. If (error is SN) and (error_rate is SN) then (PWM is SP) (1)
8. If (error is SN) and (error_rate is ZERO) then (PWM is SP) (1)
9. If (error is SN) and (error_rate is SP) then (PWM is ZERO) (1)
10. If (error is SN) and (error_rate is LP) then (PWM is SN) (1)
11. If (error is ZERO) and (error_rate is LN) then (PWM is LP) (1)
12. If (error is ZERO) and (error_rate is SN) then (PWM is SP) (1)
13. If (error is ZERO) and (error_rate is ZERO) then (PWM is ZERO) (1)
14. If (error is ZERO) and (error_rate is SP) then (PWM is SN) (1)
15. If (error is ZERO) and (error_rate is LP) then (PWM is LN) (1)
16. If (error is SP) and (error_rate is LN) then (PWM is LP) (1)
17. If (error is SP) and (error_rate is SN) then (PWM is SP) (1)
18. If (error is SP) and (error_rate is ZERO) then (PWM is SN) (1)
19. If (error is SP) and (error_rate is SP) then (PWM is SN) (1)
20. If (error is SP) and (error_rate is LP) then (PWM is LN) (1)
21. If (error is LP) and (error_rate is LN) then (PWM is ZERO) (1)
22. If (error is LP) and (error_rate is SN) then (PWM is SN) (1)
23. If (error is LP) and (error_rate is ZERO) then (PWM is LN) (1)
24. If (error is LP) and (error_rate is SP) then (PWM is SN) (1)
25. If (error is LP) and (error_rate is LP) then (PWM is LN) (1)

Fig. 8.3 Fuzzy control rules

$$u = \frac{\sum_{i=1}^{m} (u_i \times U_i)}{\sum_{i=1}^{m} u_i} \tag{8.2}$$

where u is the current crisp output of the fuzzy inference system and this equation is called the Center-Of-Gravity method.

With the help of the MATLAB® Fuzzy Logic Toolbox™, one can easily view the control rules and their functions based on a graphical output control wizard, as shown in Fig. 8.4. In the MATLAB® Fuzzy Logic Toolbox™ window, go to **View|Rules** menu item to open this graphical view for all control rules built for this fuzzy controller. The entire control output or envelope can also be viewed by going to **View|Surface** menu item, as shown in Fig. 8.5.

The results of the defuzzification process are the real crisp outputs, the motor rotation speeds in our project, can be derived or calculated in two different ways: the off-line and the online methods. The so-called off-line method is to build a lookup table before the project runs. In the lookup table, each crisp control output can be uniquely determined based on the current input error and error rate pair. The online method is to calculate the control output in real time based on the current error inputs and the control output in terms of the COG equation.

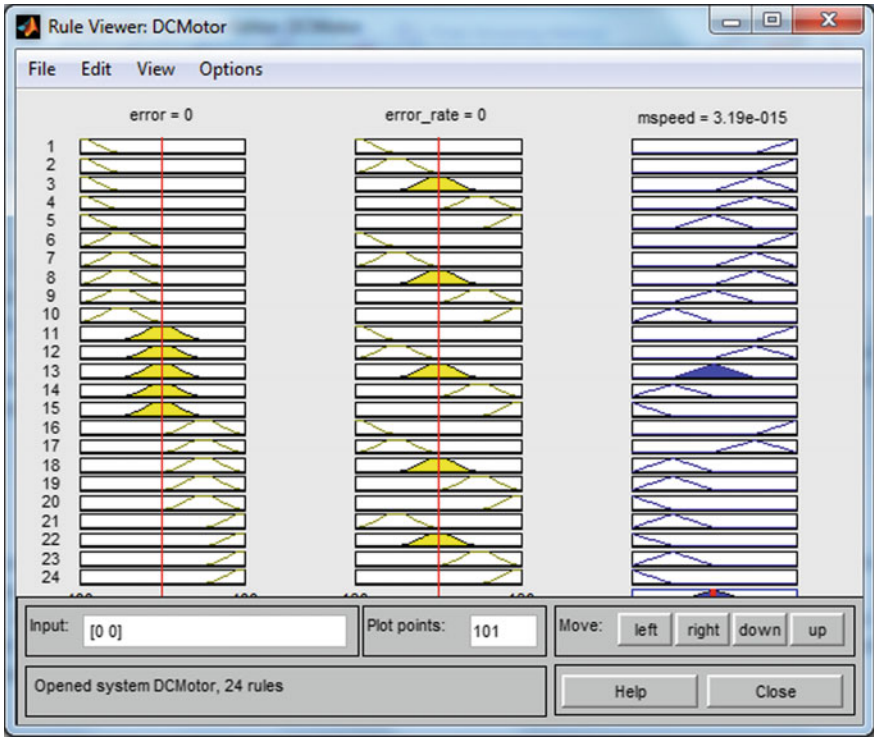


Fig. 8.4 Graphic representation of the control rules

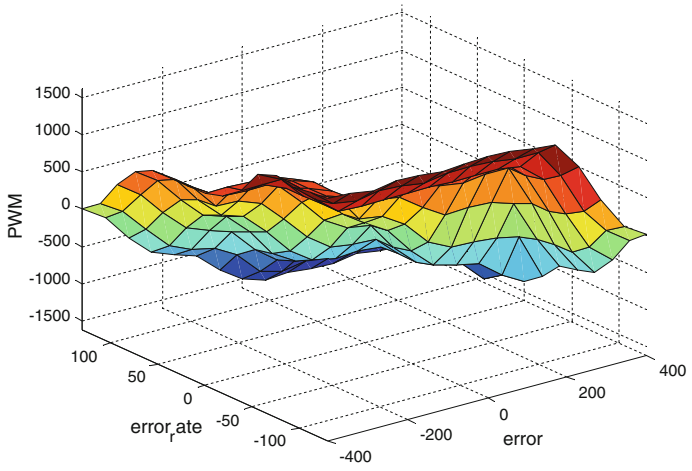


Fig. 8.5 The fuzzy output surface or envelope

In this project, we try to use the off-line method to calculate a lookup table to simplify this defuzzification process to obtain the control outputs.

In order to apply these control rules and defuzzification processes in our project, we need to convert them to the C codes with the help of an **IF** selection structure.

8.3 Apply the Fuzzy Logic Controller to the DC Motor Control System

To use the lookup table to get the desired control output, the **IF** selection structure should be used to divide the input errors and the output controls into the different ranges. Based on the different ranges, the control output can be determined from the lookup table. The control outputs in the lookup table are crisp variables.

It can be found from Table 8.2 that the control output PWM value is grouped based on their peak values, not the ranges, and these peak values are corresponded to the U_i values in the COG equation shown above since this is required by the COG method.

Based on control rules shown in Fig. 8.3, an input–output relationship table can be developed and shown in Table 8.3. This table can be considered as a 2-D matrix and each intersection cell is the control output **PWM** or **pwm(i, j)**.

According to the COG equation, each u_i should be the **AND** or the minimum value selected between the **MF_*e*** and **MF_**Δ***e*** pair: **u(i, j) = min(MF_*e*, MF_**Δ***e*)** if each u_i can be presented by an element in a 2D matrix, as shown in Table 8.4. The **MF_*e*** and the **MF_**Δ***e*** are the membership functions of the error ***e*** and the error rate Δ***e***. In this way, by using these two tables, we can easily calculate each crisp control output or PWM value based on the COG equation shown above since each cell in these two tables has a one-to-one relationship. For example, $u(0,1)$ in Table 8.4 should be multiplied by the cell **LP** in Table 8.3 if ***e*** is **LN** and Δ***e*** is **SN** to get **pwm(0,1)**. Similarly, $u(0,2)$ in Table 8.4 should be multiplied by the cell **SP** in Table 8.3 if ***e*** is **LN** and Δ***e*** is **ZE** to get **pwm(0,2)**. Finally, we can get the crisp control output u by adding all **pwm(i, j)** together as the numerator and adding all **u(i,j)** together as the denominator.

The pseudo codes for these two examples can be presented as follows:

Table 8.2 The ranges of the inputs and the output of the DC motor

Input velocity error	<−300	−300 to −100	−100 to 100	100 to 300	>300
MF	LN	SN	ZE	SP	LP
Input error rate	<−100	−100 to −50	−50 to 50	50 to 100	>100
MF	LN	SN	ZE	SP	LP
Output velocity	−1500	−1000	0	1000	1500
MF	LN	SN	ZE	SP	LP

Table 8.3 The control rules

e	Δe				
	LN	SN	ZE	SP	LP
LN	LP	LP	SP	SP	ZE
SN	LP	SP	SP	ZE	SN
ZE	LP	SP	ZE	SN	LN
SP	LP	SP	SN	SN	LN
LP	ZE	SN	LN	SN	LN

Table 8.4 The input errors

e	Δe				
	LN	SN	ZE	SP	LP
LN	$u(0,0)$	$u(0,1)$	$u(0,2)$	$u(0,3)$	$u(0,4)$
SN	$u(1,0)$	$u(1,1)$	$u(1,2)$	$u(1,3)$	$u(1,4)$
ZE	$u(2,0)$	$u(2,1)$	$u(2,2)$	$u(2,3)$	$u(2,4)$
SP	$u(3,0)$	$u(3,1)$	$u(3,2)$	$u(3,3)$	$u(3,4)$
LP	$u(4,0)$	$u(4,1)$	$u(4,2)$	$u(4,3)$	$u(4,4)$

If e = LN and Δe = SN, Then the $u_1 = u(0, 1) \times$ LP = min(**MF_*e***, **MF_**Δ***e***) $\times$ LP
If e = LN and Δe = ZE, Then the $u_2 = u(0, 2) \times$ SP = min(**MF_*e***, **MF_**Δ***e***) $\times$ SP

Furthermore, these two pseudo instructions can be written as follows:

If ***e*** < −300 and **−100** < Δ***e*** < **−50**, Then u_1 = **min(0.5, 0.8)** $\times$ **1500** = **0.5** $\times$ **1500** = **pwm(0, 1)** = 750.
If ***e*** < **−300** and **−50** < Δ***e*** < **50**, Then u_2 = **min(0.5, 1)** $\times$ **1000** = **0.5** $\times$ **1000** = **pwm(0, 2)** = 500.

Refer to Fig. 8.2 to get the values of the membership functions for each error and error rate. Finally, the control output can be calculated as

$$u = \frac{u(0,1) \times \mathrm{LP} + u(0,2) \times \mathrm{SP}}{u(0,1) + u(0,2)} = \frac{0.5 \times 1500 + 0.5 \times 1000}{0.5 + 0.5} = 1250 \qquad (8.3)$$

In this example, we used the approximated MF values as the real MF values to simplify this calculation process. To make our project easy and simple, we used MATLAB® Fuzzy Logic Toolbox™ to build our lookup table.

Basically, the core of using a lookup table is to predefine or pre-calculate all crisp control outputs based on the different ranges of input errors and error rates. In other words, each control output in the lookup table is not corresponded to the current error and error rate input value, but instead it is based on a range of the input error and error rate. All input errors and error rates can be divided into different ranges, and each range is associated to a crisp control output in the lookup table. The smaller the error and error rate ranges, the more accurate control outputs can be

obtained from the lookup table. However, the more memory space is needed to store those control outputs in a larger lookup table. For example, if both the errors and error rates in our project are divided into four ranges, such as −400 to 400 for errors and −120 to 120 for error rates with each range for the errors and the error rates being 200 and 60, the control output lookup table would be a 5 × 5 matrix with 25 elements. In the program, 25 **IF-THEN** instructions would be used to distinguish these errors and error rates to obtain desired control output from the lookup table.

To make a more accurate lookup table, one can divide the input velocity error ***e*** and error rate Δ***e*** into smaller equal segments or ranges. A trade-off needs to be made between the accuracy and space for any control system. In this project, we select to use a 5 × 5 lookup table to divide the inputs into four ranges.

The MATLAB® script used to calculate our lookup table is shown in Fig. 8.6. In this script, an MATLAB® function **evalfis()** in the Fuzzy Logic Toolbox is used to calculate this 5 × 5 lookup table. The finished lookup table is shown in Fig. 8.7.

Table 8.5 shows the relationship between the input errors, error rate ranges, and the control outputs in our lookup table. This table can be directly used by our program to determine and identify the desired control outputs based on the velocity error and error rate input ranges with the help of the **IF-THEN** selection instructions. One point to be noted is that some negative control outputs may never be used since we did not configure the motor to rotate in the CCW direction in our program.

Regularly you should use the MATLAB® Simulink® to perform simulations for this fuzzy controller. Because of the space limitation, we have to skip this step. Now let's build our fuzzy control program **Fuzzy Control** to control the DC motor with the fuzzy logic controller.

```
% Calculate the fuzzy lookup table for DC Motor control system – DCMotorLT.m
% Feb 18, 2018
% Y. Bai

MLT = zeros(5, 5);        % lookup table is a 5 by 5 matrix
e   = -400;               % most negative speed error: range: -400 ~ 400: step 200 (4 steps)
de  = -120;               % most negative speed error rate: range: -120 ~ 120: step: 60 (4 steps)

% the resulted lookup table is a 5 by 5 matrix.
a = readfis('DCMotor'); % get the DCMotor fuzzy inference system – DCMotor.fis

for i = 1:5
  for j = 1:5
  MLT(i, j) = evalfis([(e + (i-1)*200) (de + (j-1)*60)], a);
 end
end

MLT                       % display the resulted lookup table
```

Fig. 8.6 The script used to calculate the lookup table

```
MLT =

   1616.5   1573.4   958.2    919.7     76.8
   1513.0    958.2    889.5     30.0     818.5
   1513.0    860.0      0.000 -860.0   -1419.6
   1289.9    809.9   -889.5   -958.2   -1513.0
    -74.6   -889.6  -1364.8  -1002.6   -1513.0
```

Fig. 8.7 The finished lookup table

Table 8.5 The relationship of input errors and control outputs

e	Δe				
	<−120	−120 to −60	−60 to 0	0 to 60	60 to 120
<−400	1617	1573	958	920	77
−400 to −200	1513	958	890	30	819
−200 to 0	1513	860	0	−860	−1420
0 to 200	1290	810	−890	−958	−1513
200 to 400	−75	−890	−1365	−1003	−1513

8.3.1 Build the Fuzzy Logic Control Project Fuzzy Control

Create a new folder **Fuzzy Control** under the folder **C:\C-M Control Class Projects\Chapter 8** and a new Keil® μVersion®5 project named **Fuzzy Control** under that folder. Since this project is a little large, thus a header file and a C source file are needed to be developed.

8.3.1.1 Create the Header File Fuzzy Control.h

Create a new header file **Fuzzy Control.h** and add it into the project **Fuzzy Control** that is just created above. Enter the codes shown in Fig. 8.8 into this header file.

Let's have a closer look at this header file to see how it works.

(1) The codes in lines 4 ~ 9 are used to declare some system header files to be used in this project.
(2) The codes in lines 10 ~ 12 are used to define three constants to be used in the project, the upper bound and the lower bound of the PWM output values as well as the conversion coefficient HS that is used to convert the feedback speed value to the corresponded PWM value.
(3) The mapping addresses for the GPIO Ports D and F as well as QEI0 related GPIO pins are defined in lines 13 ~ 17.

```
1  //*****************************************************************************
2  // Fuzzy-Control.h - Header File for Fuzzy-Control Program
3  //*****************************************************************************
4  #include <stdint.h>
5  #include <stdbool.h>
6  #include "driverlib/sysctl.h"
7  #include "driverlib/gpio.h"
8  #include "driverlib/qei.h"
9  #include "TM4C123GH6PM.h"
10 #define  PWMMAX          3999
11 #define  PWMMIN          5
12 #define  HS              2.22                    // HS = 1/K = 1/0.45 = 2.22
13 #define  GPIO_PD6_PHA0           0x00031806
14 #define  GPIO_PD7_PHB0           0x00031C06
15 #define  GPIO_PORTD_BASE         0x40007000      // GPIO Port D
16 #define  GPIO_PORTF_CR_R         (*((volatile uint32_t *)0x40025524))
17 #define  GPIO_PORTD_CR_R         (*((volatile uint32_t *)0x40007524))
18 void InitPWM(void);
19 void InitQEI(void);
20 int GetLookup(double error, double error_rate);
21 int E_LN = -400, E_SN = -200, E_ZE = 0, E_SP = 200, E_LP = 400;
22 int ER_LN = -120, ER_SN = -60, ER_ZE = 0, ER_SP = 60, ER_LP = 120;
23 static int MLT[5][5] = {{1617, 1573, 958,  920,   77},
24                         {1513, 958,  890,   30,  819},
25                         {1513, 860,  0,   -860, -1420},
26                         {1290, 810, -890,  -958, -1513},
27                         {-75, -890, -1365,-1003, -1513}};
```

Fig. 8.8 The header file of the project Fuzzy Control

(4) Three user-defined functions, **InitPWM()**, **InitQEI()**, and **GetLookup()** are declared in lines 18 ~ 20. The **GetLookup()** function is used to get the crisp control output value from the fuzzy lookup table based on the current input velocity error and the error rate values.
(5) The ranges of the fuzzy input velocity errors and error rates are defined in lines 21 and 22. Five points are defined to divide the input errors and error rates into four ranges, and these points are obtained from Table 8.5 directly.
(6) The lookup table **MLT[]** is declared in lines 23 ~ 27, and it is a 5 × 5 matrix. The values on this lookup table can be obtained from the Table 8.5.

Next, let's build the C source file for this project.

8.3.1.2 Create the C Source File Fuzzy Control.c

The size of this source file is a little large since it contains the codes for the fuzzy inference process. Thus, this source file can be divided into three parts: the **main()** program part, the fuzzy inference process part and the initialization processes for the PWM1 and QEI0 modules. Because the initialization process functions for the PWM and QEI modules, **InitPWM()** and **InitQEI()**, are identical with those functions built for the previous projects, such as **CalibEncoder** (Sect. 4.7.4 in Chap. 4) and **PID Control** (Sect. 5.5.2 in Chap. 5), we will not discuss them in this section. Refer to those projects to get more details about the codes for these two functions.

The first part of this source file is shown in Fig. 8.9.

Basically, the codes in this part are identical with those in the **PID Control** project built in Sect. 5.5.2 in Chap. 5, therefore, we only discuss the codes that are different from those codes in that project. Let's have a closer look at these codes to see how they work.

(1) The type of the PWM output variable **pw** is changed from **uint32_t** to **int** in line 9 since both positive and negative PWM output values exist in the fuzzy lookup table.
(2) The code in line 33 is replaced by calling a user-defined function **GetLookup()** to perform the fuzzy inference calculation to get the crisp PWM output from the lookup table, instead of using the PID algorithm to get PID control output values.

All of the rest codes in this part are identical with those used in the **PID Control** project.

Now let's take a look at the second part codes for this project, which is a user-defined function **GetLookup()**. The detailed codes for this function are shown in Fig. 8.10.

```
1  //*****************************************************************************
2  // Fuzzy-Control.c - Motor Fuzzy Logic Control Program - QEI0
3  //*****************************************************************************
4  #include "Fuzzy-Control.h"
5  int main(void)
6  {
7     uint32_t es, index, upper = 1000, s, n = 0, motor[100];
8     double de, e[2];
9     int pw;
10    SysCtlClockSet(SYSCTL_SYSDIV_25|SYSCTL_USE_PLL|SYSCTL_XTAL_4MHZ|SYSCTL_OSC_MAIN);
11    SYSCTL->RCGC2 = 0x2A;                  // enable & clock GPIO Ports B, D & F
12    GPIOB->DEN = 0xF;                      // enable PB3 ~ PB0 as digital function pins
13    GPIOB->DIR = 0xF;                      // configure PB3 ~ PB0 as output pins
14    GPIOD->DEN = 0xC0;                     // enable PD7 ~ PD6 as digital function pins
15    GPIOD->DIR = ~0xC;                     // configure PD7 ~ PD6 as input pins
16    GPIOF->DEN = 0xF;                      // enable PF3 ~ PF0 as digital function pins
17    GPIOF->DIR = 0xF;                      // configure PF3 ~ PF0 as output pins
18    InitPWM();                             // configure PWM module 1
19    InitQEI();                             // configure QEI module 0
20    while(1)
21    {
22       PWM1->_2_CTL = 0x1;                 // enable PWM1_2B or M1PWM5
23       PWM1->ENABLE = 0x20;
24       s = 1000;                           // s = 1000 makes es = 0.45*PWM = 0.45*1000 = 450 ppr
25                                           // the 450 ppr can be considered as a setup input velocity
26       PWM1->_2_CMPB = s;                  // send desired PWM value or 450 ppr to motor
27       e[1] = s;
28       for (index = 0; index < upper; index++)   // output a square wave form to motor
29       {
30          es = QEI0->SPEED;                // get the current encoder speed value
31          e[0] = s - es*HS;                // convert speed to corresponding PWM value
32          de = e[1] - e[0];                // get differential error
33          pw = GetLookup(e[0],  de);       // get crisp control output from lookup table
34          e[1] = e[0];
35          PWM1->_2_CMPB = pw;              // with fuzzy logic  control
36          SysCtlDelay(5);
37          if (n < 100)  { motor[n] = es; }
38          n++;
39       }
40       PWM1->_2_CMPB = 0;                  // 0 to motor to stop motor
41       PWM1->_2_CTL = 0x0;                 // disable PWM1_2B or M1PWM5
42       PWM1->ENABLE &= ~0x20;
43       for (index = 0; index < upper; index++)
44         SysCtlDelay(10);
45    }
46 }
```

Fig. 8.9 The first part codes for the project Fuzzy Control

(1) The **GetLookup()** function starts at line 1 with two arguments, the motor encoder feedback velocity error and error rate, **error** and **error_rate**, and both of them are **double** data type. The returned variable data type is **int**.
(2) The returned variable **u** is declared in line 3.
(3) Totally we have 25 control outputs located in a 5 × 5 lookup table. In order to identify and select the desired control output from the lookup table, we need to use the feedback velocity error and error rate as inputs. In other words, we need to do this selection based on the different error and error rate ranges defined in the header file **Fuzzy Control.h** and these ranges are defined based on the error and error rate ranges defined in Table 8.5. Therefore, a sequence of **if** and **else if** selection structures is used to do this selection job.
(4) We do this selection by dividing error ranges into five segments, and perform five sub-selections based on the error rate ranges inside each error segment.

```
1  int GetLookup(double error, double error_rate)
2  {
3     int u;
4     if (error < E_LN)
5     {
6        if (error_rate < ER_LN) { u = MLT[0][0]; }
7        else if (error_rate < ER_SN && error_rate >= ER_LN) { u = MLT[0][1]; }
8        else if (error_rate < ER_ZE && error_rate >= ER_SN) { u = MLT[0][2]; }
9        else if (error_rate < ER_SP && error_rate >= ER_ZE)  { u = MLT[0][3]; }
10       else if (error_rate < ER_LP && error_rate >= ER_SP)  { u = MLT[0][4]; }
11    }
12    else if (error < E_SN && error >= E_LN)
13    {
14       if (error_rate < ER_LN) { u = MLT[1][0]; }
15       else if (error_rate < ER_SN && error_rate >= ER_LN) { u = MLT[1][1]; }
16       else if (error_rate < ER_ZE && error_rate >= ER_SN) { u = MLT[1][2]; }
17       else if (error_rate < ER_SP && error_rate >= ER_ZE)  { u = MLT[1][3]; }
18       else if (error_rate < ER_LP && error_rate >= ER_SP)  { u = MLT[1][4]; }
19    }
20    else if (error < E_ZE && error >= E_SN)
21    {
22       if (error_rate < ER_LN) { u = MLT[2][0]; }
23       else if (error_rate < ER_SN && error_rate >= ER_LN) { u = MLT[2][1]; }
24       else if (error_rate < ER_ZE && error_rate >= ER_SN) { u = MLT[2][2]; }
25       else if (error_rate < ER_SP && error_rate >= ER_ZE)  { u = MLT[2][3]; }
26       else if (error_rate < ER_LP && error_rate >= ER_SP)  { u = MLT[2][4]; }
27    }
28    else if (error < E_SP && error >= E_ZE)
29    {
30       if (error_rate < ER_LN) { u = MLT[3][0]; }
31       else if (error_rate < ER_SN && error_rate >= ER_LN) { u = MLT[3][1]; }
32       else if (error_rate < ER_ZE && error_rate >= ER_SN) { u = MLT[3][2]; }
33       else if (error_rate < ER_SP && error_rate >= ER_ZE)  { u = MLT[3][3]; }
34       else if (error_rate < ER_LP && error_rate >= ER_SP)  { u = MLT[3][4]; }
35    }
36    else if (error < E_LP && error >= E_SP)
37    {
38       if (error_rate < ER_LN) { u = MLT[4][0]; }
39       else if (error_rate < ER_SN && error_rate >= ER_LN) { u = MLT[4][1]; }
40       else if (error_rate < ER_ZE && error_rate >= ER_SN) { u = MLT[4][2]; }
41       else if (error_rate < ER_SP && error_rate >= ER_ZE)  { u = MLT[4][3]; }
42       else if (error_rate < ER_LP && error_rate >= ER_SP)  { u = MLT[4][4]; }
43    }
44    return u;
45 }
```

Fig. 8.10 The detailed codes for the function GetLookup()

Starting at line 4, the first error segment is: if the **error** is less than **E_LN**, we continue to check five error rate ranges, **< ER_LN**, **ER_LN** ∼ **ER_SN**, **ER_SN** ∼ **ER_ZE**, **ER_ZE** ∼ **ER_SP**, and **ER_SP** ∼ **ER_LP**, in lines 6∼10. Based on each error rate range, the corresponded control output can be identified by the related element located at the lookup table **MLT[0, n]** and assigned to the returned variable **u**.

(5) The second error segment, **error** < **E_SN** && **error** > = **E_LN**, starts at line 12. If the velocity error is in this range, the five sub-selections are performed to check five error rate ranges that are identical with those inside the last error segment to select the corresponded control output **MLT[1, n]** from the lookup table in lines 14∼18, respectively.

(6) In a similar way, the following three error segments and related five sub-selections can be performed to select the corresponded control outputs

MLT[2, n], **MLT[3, n]**, and **MLT[4, n]** and assigned them to the returned variable **u** in lines 22 ~ 26, 30 ~ 34, and 38 ~ 42.

(7) Finally, the identified control output stored in the returned variable **u** is returned to the main program in line 44.

One point to be noted is that the motor rotation direction and the direction we defined in our program. In this project, we defined the motor positive rotating direction as Clockwise (**CW**) and the membership functions and control outputs defined in the lookup table are based on this definition. If you find that the direction is opposite or your simulation result is negative, you need to change the direction in your program to redefine this direction. Also, the membership function input definitions, such as **LN**, **SN**, **SP**, and **LP**, need to be inverted in polarity.

8.3.1.3 Setup Environments to Build and Run the Project

Before one can run this project to test the fuzzy logic controller, the following environments need to be set:

- Add the path for all system header files used in this project into the **Include Paths** box in the **C/C++** tab under the **Project|Options for Target 'Target 1'** menu item. The correct path should be: **C:\ti\TivaWare_C_Series-2.1.4.178**.
- Select the correct debug driver **Stellaris ICDI** in the **Debug** tab under the **Project|Options for Target "Target 1"** menu item.
- Add the TivaWare™ Peripheral Driver Library **driverlib.lib** file into the project. This library file is located at: **C:\ti\TivaWare_C_Series-2.1.4.178\driverlib\rvmdk**. By right clicking on the **Source Group 1** in the **Project** pane and selecting the **Add Existing File…** item, one can add this library file into the project.

Now run the project to test the fuzzy logic controller for our DC motor closed-loop control system. You can find that the motor is periodically rotating and stopping. The step response of this fuzzy logic control system is shown in Fig. 8.11, which is plotted based on the collected motor output velocity feedback values **motor[]** acquired from the **QEI0** module.

8.4 Case Study

In Sect. 7.8.1 in Chap. 7, we discussed how to design a FLC for a DC motor system with a transfer function

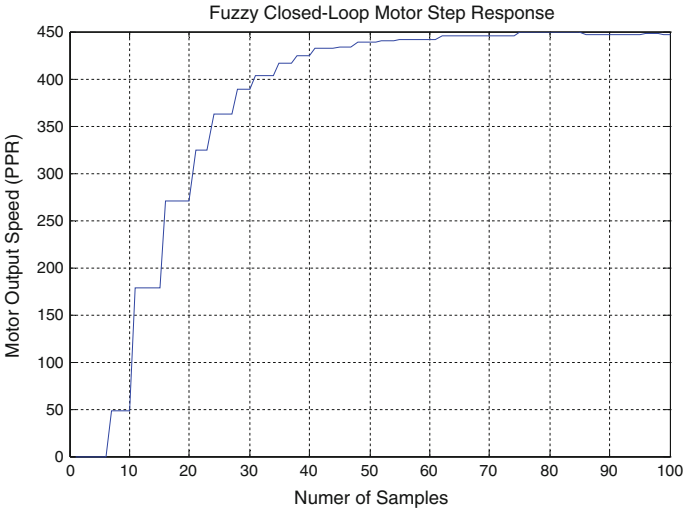


Fig. 8.11 The step response of the fuzzy logic control system

$$G(s) = \frac{6520}{s(s+430.6)} e^{-0.005s} \tag{8.4}$$

In this section, we will continue our discussion and build a real microcontroller programming project to perform the closed-loop control to this motor system.

We still use a lookup table to get the desired control output based on inputs, and the **IF** selection structure should be used to divide the input errors and the output controls into the different ranges. Based on the different ranges, the control output can be determined from the lookup table. The control outputs in the lookup table are crisp variables.

Based on Table 7.3 and Fig. 7.23 in Chap. 7, the input error, error_rate and the output can be divided into seven ranges, as shown in Table 8.6. It can be found from Table 8.6 that the control output PWM value is grouped based on their peak

Table 8.6 The ranges of the inputs and the output of the DC motor

Input error	<−0.1	−0.1 ~ −0.05	−0.05 ~ −0.01	−0.01 ~ 0.01	0.01 ~ 0.05	0.05 ~ 0.1	>0.1
MF	LN	MN	SN	ZE	SP	MP	LP
Input error rate	< −0.06	−0.06 ~ −0.03	−0.03 ~ −0.01	−0.01 ~ 0.01	0.01 ~ 0.03	0.03 ~ 0.06	>0.06
MF	LN	MN	SN	ZE	SP	MP	LP
Output velocity	−600	−400	−200	0	200	400	600
MF	LN	MN	SN	ZE	SP	MP	LP

Table 8.7 The control rules

e	Δe						
	LN	MN	SN	ZE	SP	MP	LP
LN	LP	LP	LP	SP	SP	SP	ZE
SN	LP	MP	SP	SP	ZE	SN	SN
ZE	LP	MP	SP	ZE	SN	MN	LN
SP	LP	MP	SP	SN	SN	MN	LN
LP	ZE	SN	SN	LN	SN	MN	LN

values, not the ranges, and these peak values are corresponded to the U_i values in the COG equation shown in Eq. (8.2) since this is required by the COG method.

Based on control rules shown in Fig. 8.3, an input–output relationship table can be developed and shown in Table 8.7. This table can be considered as a 2D matrix and each intersection cell is the control output **PWM** or **pwm(i, j)**.

According to the COG equation, each u_i should be the **AND** or the minimum value selected between the **MF_*e*** and **MF_Δ*e*** pair: **u(i, j) = min(MF_*e*, MF_Δ*e*)** if each u_i can be presented by an element in a 2D matrix, as shown in Table 8.8. The **MF_*e*** and the **MF_Δ*e*** are the membership functions of the error ***e*** and the error rate Δ***e***. In this way, by using these two tables, we can easily calculate each crisp control output or PWM value based on the COG equation shown in Eq. (8.2) since each cell in these two tables has a one-to-one relationship. For example, $u(0,1)$ in Table 8.8 should be multiplied by the cell **LP** in Table 8.7 if ***e*** is **LN** and Δ***e*** is **MN** to get **pwm(0,1)**. Similarly, $u(0,3)$ in Table 8.8 should be multiplied by the cell **SP** in Table 8.7 if ***e*** is **LN** and Δ***e*** is **ZE** to get **pwm(0,3)**. Finally, we can get the crisp control out u by adding all **pwm(i, j)** together as the numerator and adding all **u(i,j)** together as the denominator.

The pseudo codes for these two examples can be presented as

If e = LN and Δe = MN, Then the $u_1 = u(0, 1) \times$ LP = min(**MF_*e***, **MF_Δ*e***) × LP
If e = LN and Δe = ZE, Then the $u_3 = u(0, 3) \times$ SP = min(**MF_*e***, **MF_Δ*e***) × SP

Table 8.8 The input errors

e	Δe						
	LN	MN	SN	ZE	SP	MP	LP
LN	$u(0,0)$	$u(0,1)$	$u(0,2)$	$u(0,3)$	$u(0,4)$	$u(0,5)$	$u(0,6)$
SN	$u(1,0)$	$u(1,1)$	$u(1,2)$	$u(1,3)$	$u(1,4)$	$u(1,5)$	$u(1,6)$
ZE	$u(2,0)$	$u(2,1)$	$u(2,2)$	$u(2,3)$	$u(2,4)$	$u(2,5)$	$u(2,6)$
SP	$u(3,0)$	$u(3,1)$	$u(3,2)$	$u(3,3)$	$u(3,4)$	$u(3,5)$	$u(3,6)$
LP	$u(4,0)$	$u(4,1)$	$u(4,2)$	$u(4,3)$	$u(4,4)$	$u(4,5)$	$u(4,6)$

Furthermore, these two pseudo instructions can be written as:

If $\boldsymbol{e} < -\mathbf{0.1}$ and $-\mathbf{0.06} < \Delta\boldsymbol{e} < -\mathbf{0.03}$, Then $u_1 = \mathbf{min(0.5,\ 0.8)} \times \mathbf{600} = \mathbf{0.5} \times \mathbf{600} = \mathbf{pwm(0,\ 1)} = 300$.
If $\boldsymbol{e} < -\mathbf{0.1}$ and $-\mathbf{0.01} < \Delta\boldsymbol{e} < \mathbf{0.01}$, Then $u_3 = \mathbf{min(0.5,\ 1)} \times \mathbf{200} = \mathbf{0.5} \times \mathbf{200} = \mathbf{pwm(0,\ 3)} = 100$.

Refer to Fig. 7.23 in Chap. 7 to get the values of the membership functions for each error and error rate. Finally, the control output can be calculated as

$$u = \frac{u(0,1) \times \mathrm{LP} + u(0,3) \times \mathrm{SP}}{u(0,1) + u(0,3)} = \frac{0.5 \times 600 + 0.5 \times 200}{0.5 + 0.5} = 400 \tag{8.5}$$

We used the approximated MF values as the real MF values to simplify this calculation process. To make our project easy and simple, we used MATLAB® Fuzzy Logic Toolbox™ to build our lookup table.

Basically, the core of using a lookup table is to predefine or pre-calculate all crisp control outputs based on the different ranges of input errors and error rates. All input errors and error rates can be divided into different ranges, and each range is associated to a crisp control output in the lookup table. The smaller the error and error rate ranges, the more accurate control outputs can be obtained from the lookup table. However, the more memory space is needed to store those control outputs in a larger lookup table. For example, if both the errors and error rates in our project are divided into six equal ranges, such as −0.1 to 0.1 for errors and −0.06 to 0.06 for error rates with each range for the errors and the error rates being 0.03 and 0.02, the control output lookup table would be a 7 × 7 matrix with 49 elements. In the program, 49 **IF-THEN** instructions would be used to distinguish these errors and error rates to obtain desired control output from the lookup table.

To make a more accurate lookup table, one can divide the input velocity error $\boldsymbol{e}$ and error rate $\Delta\boldsymbol{e}$ into smaller equal segments or ranges. A trade-off needs to be made between the accuracy and space for any control system. In this project, we select to use a 7 × 7 lookup table to divide the inputs into six ranges.

The MATLAB® script used to calculate our lookup table is shown in Fig. 8.12. In this script, a MATLAB® function **evalfis()** in the Fuzzy Logic Toolbox is used to calculate this 7 × 7 lookup table. The finished lookup table is shown in Fig. 8.13.

Table 8.9 shows the relationship between the input errors, error rate ranges, and the control outputs in our lookup table. This table can be directly used by our program to determine and identify the desired control outputs based on the velocity error and error rate input ranges with the help of the **IF-THEN** selection instructions. One point to be noted is that some negative control outputs may never be used since we did not configure the motor to rotate in the CCW direction in our program.

```
% Calculate the fuzzy lookup table for DC Motor control system – FZMotorLT.m
% Feb 3, 2018
% Y. Bai

FZMLT = zeros(7, 7);      % resulted lookup table is a 7 by 7 matrix
e  = -0.1;                % most negative speed error: range: -0.1 ~ 0.1: step 0.033 (6 steps)
de  = -0.06;              % most negative speed error rate: range: -0.06 ~ 0.06: step: 0.02 (6 steps)

a = readfis('FZMotor');   % get the FZMotor fuzzy inference system – FZMotor.fis

for i = 1:7
  for j = 1:7
  FZMLT(i, j) = evalfis([(e + (i-1)*0.033) (de + (j-1)*0.02)], a);
 end
end

FZMLT                     % display the resulted lookup table
```

Fig. 8.12 The script used to calculate the lookup table

```
FZMLT =

 -502.6755  -502.2861  -461.1693  -389.0839  -275.2139  -111.9937   25.6576
 -465.1543  -404.9152  -399.8878  -301.4340  -165.0299   -57.1639   83.9465
 -357.6980  -354.8496  -199.6049  -148.0946   -47.6655   130.0324  180.0034
 -354.5252  -224.0123  -145.1449   -3.6552    122.5543   196.3798  348.3112
 -200.2323  -151.8705   34.1608    133.3607   192.5916   350.4985  351.5642
 -114.8222    8.1741    145.8789   300.8185   398.0874   400.0460  462.9850
   -6.8332   114.4380   293.0933   400.9215   459.8715   506.6105  506.9236
```

Fig. 8.13 The finished lookup table

Table 8.9 The relationship of input errors and control outputs

e	Δe						
	<-0.06	$-0.06\sim-0.04$	$-0.04\sim-0.02$	$-0.02\sim0.02$	$0.02\sim0.04$	$0.04\sim0.06$	>0.06
<-0.1	−502	−502	−461	−389	−275	−112	26
$-0.1\sim-0.067$	−465	−405	−400	−301	−165	−57	84
$-0.067\sim-0.034$	−358	−355	−200	−148	−48	130	180
$-0.034\sim0.032$	−355	−224	−145	−3	123	196	348
$0.032\sim0.065$	−200	−152	34	133	193	350	352
$0.065\sim0.1$	−115	8	146	301	398	400	463
>0.1	−7	114	293	401	460	507	507

Regularly, you should use the MATLAB® Simulink® to perform simulations for this fuzzy controller. Since we have provided this simulation study in Sect. 7.10 in Chap. 7, so we can skip this step. Now let's build our fuzzy control program **FZControl** to control the DC motor.

8.4.1 Build the Fuzzy Logic Control Project FZControl

Create a new folder **FZControl** under the folder **C:\C-M Control Class Projects\Chapter 8** and a new Keil® μVersion®5 project named **FZControl** under that folder. Since this project is a little large, thus a header file and a C source file are needed to be developed.

8.4.1.1 Create the Header File FZControl.h

Create a new header file **FZControl.h** and add it into the project **FZControl** that is just created above. Enter the codes shown in Fig. 8.14 into this header file.

```
1  //*****************************************************************************
2  // FZControl.h - Header File for Fuzzy Logic Control Program
3  //*****************************************************************************
4  #include <stdint.h>
5  #include <stdbool.h>
6  #include "driverlib/sysctl.h"
7  #include "driverlib/gpio.h"
8  #include "driverlib/qei.h"
9  #include "TM4C123GH6PM.h"

10 #define  PWMMAX          3999
11 #define  PWMMIN          5
12 #define  HS              2.22                    // HS = 1/K = 1/0.45 = 2.22

13 #define  GPIO_PD6_PHA0          0x00031806
14 #define  GPIO_PD7_PHB0          0x00031C06
15 #define  GPIO_PORTD_BASE        0x40007000      // GPIO Port D
16 #define  GPIO_PORTF_CR_R        (*((volatile uint32_t *)0x40025524))
17 #define  GPIO_PORTD_CR_R        (*((volatile uint32_t *)0x40007524))

18 void InitPWM(void);
19 void InitQEI(void);
20 int GetLookup(float  error, float  error_rate);

21 float E_LN = -0.1, E_MN = -0.067, E_SN = -0.034, E_ZE = 0, E_SP = 0.034,  E_MP = 0.067, E_LP = 0.1;
22 float ER_LN = -0.06, ER_MN = -0.04, ER_SN = -0.02, ER_ZE = 0, ER_SP = 0.02, ER_MP = 0.04, ER_LP = 0.06;

23 static int MLT[7][7] = {{-502, -502,  -461, -389, -275, -111,   26},
24                         {-465, -405,  -400, -301, -165,  -57,   84},
25                         {-358, -355,  -200, -148, -48,   130,  180},
26                         {-355, -224,  -145,  -4,   123,  196,  348},
27                         {-200, -152,  34,   133,  193,   350,  352},
28                         {-115,    8,  146,  301,  398,  400,  463},
29                         {-7,     114, 293,  401,  460,  507,  507}};
```

Fig. 8.14 The header file of the project FZControl

Let's have a closer look at this header file to see how it works.

(1) The codes in lines 4 ~ 9 are used to declare some system header files to be used in this project.
(2) The codes in lines 10 ~ 12 are used to define three constants to be used in the project, the upper bound and the lower bound of the PWM output values as well as the conversion coefficient HS that is used to convert the feedback speed value to the corresponded PWM value.
(3) The mapping addresses for the GPIO Ports D and F as well as QEI0 related GPIO pins are defined in lines 13 ~ 17.
(4) Three user-defined functions, **InitPWM()**, **InitQEI()**, and **GetLookup()**, are declared in lines 18 ~ 20. The **GetLookup()** function is used to get the crisp control output value from the fuzzy lookup table based on the current input velocity error and the error rate values.
(5) The ranges of the fuzzy input velocity errors and error rates are defined in lines 21 and 22. Seven points are defined to divide the input errors and error rates into six ranges, and these points are obtained from Table 8.9 directly.
(6) The lookup table **MLT[]** is declared in lines 23 ~ 27, and it is a 7 × 7 matrix. The values on this lookup table can be obtained from the Table 8.9.

Next, let's build the C source file for this project.

8.4.1.2 Create the C Source File FZControl.c

The size of this source file is large since it contains the codes for the fuzzy inference process. Thus, this source file can be divided into three parts; the **main()** program part, the fuzzy inference process part and the initialization processes for the PWM1 and QEI0 modules. Because the initialization process functions for the PWM and QEI modules, **InitPWM()**, and **InitQEI()**, are identical with those functions built for the previous projects, such as **CalibEncoder** (Sect. 4.7.4 in Chap. 4) and **PID Control** (Sect. 5.5.2 in Chap. 5), we will not discuss them in this section. Refer to those projects to get more details about the codes for these two functions.

The first part of this source file is shown in Fig. 8.15.

Basically, the codes in this part are identical with those in the **Fuzzy Control** project built in Sect. 8.3.1.2, therefore, we only discuss the codes that are different from those codes in that project. Let's have a closer look at these codes to see how they work.

(1) The upper bound value for the PWM **upper** is changed to **600** in line 7 since this number is identified from the previous project **FZMotor.fis** in Sect. 7.8.1 in Chap. 7.
(2) The data type for both input encoder error **e[]** and error rate **de** is changed from *double* to *float* to match it to the function **GetLookup()** defined in the header file.

```
1  //*****************************************************************************
2  // FZControl.c - Motor Fuzzy Logic Control Program - QEI0
3  //*****************************************************************************
4  #include "FZControl.h"
5  int main(void)
6  {
7     uint32_t es, index, s, upper = 600, n = 0, motor[100];
8     float  de, e[2];
9     int pw;
10    SysCtlClockSet(SYSCTL_SYSDIV_25|SYSCTL_USE_PLL|SYSCTL_XTAL_4MHZ|SYSCTL_OSC_MAIN);
11    SYSCTL->RCGC2 = 0x2A;                  // enable & clock GPIO Ports B, D & F
12    GPIOB->DEN = 0xF;                      // enable PB3 ~ PB0 as digital function pins
13    GPIOB->DIR = 0xF;                      // configure PB3 ~ PB0 as output pins
14    GPIOD->DEN = 0xC0;                     // enable PD7 ~ PD6 as digital function pins
15    GPIOD->DIR = ~0xC;                     // configure PD7 ~ PD6 as input pins
16    GPIOF->DEN = 0xF;                      // enable PF3 ~ PF0 as digital function pins
17    GPIOF->DIR = 0xF;                      // configure PF3 ~ PF0 as output pins
18    InitPWM();                             // configure PWM module 1
19    InitQEI();                             // configure QEI module 0
20    while(1)
21    {
22       PWM1->_2_CTL = 0x1;                 // enable PWM1_2B or M1PWM5
23       PWM1->ENABLE = 0x20;
24       s = 1000;                           // s = 1000 makes es = 0.45*PWM = 0.45*1000 = 450 ppr
25                                           // the 450 ppr can be considered as a setup input velocity
26       PWM1->_2_CMPB = s;                  // send desired PWM value or 450 ppr to motor
27       e[1] = s;
28       for (index = 0; index < upper; index++)   // output a square wave form to motor
29       {
30          es = QEI0->SPEED;                // get the current encoder speed value
31          e[0] = s - es*HS;                // convert speed to corresponding PWM value
32          de = e[1] - e[0];                // get differential error
33          pw = GetLookup(e[0],  de);       // get crisp control output from lookup table
34          e[1] = e[0];
35          PWM1->_2_CMPB = pw;              // with fuzzy logic  control
36          SysCtlDelay(5);
37          if (n < 100)  { motor[n] = es; }
38          n++;
39       }
40       PWM1->_2_CMPB = 0;                  // 0 to motor to stop motor
41       PWM1->_2_CTL = 0x0;                 // disable PWM1_2B or M1PWM5
42       PWM1->ENABLE &= ~0x20;
43       for (index = 0; index < upper; index++)
44         SysCtlDelay(10);
45    }
46 }
```

Fig. 8.15 The first part codes for the project FZControl

All of the rest codes in this part are identical with those built in the **Fuzzy Control** project.

Now let's take a look at the second part codes for this project, which is a user-defined function **GetLookup()**. The detailed codes for this function are shown in Fig. 8.16.

(1) The **GetLookup()** function starts at line 1 with two arguments, the motor encoder feedback velocity error and error rate, **error** and **error_rate**, and both of them are **float** data type. The returned variable data type is **int**.
(2) The returned variable **u** is declared in line 2.
(3) Totally we have 49 control outputs located in a 7 × 7 lookup table. In order to identify and select the desired control output from the lookup table, we need to use the feedback velocity error and error rate as inputs. In other words, we need

```
int GetLookup(float  error, float  error_rate)
{  int u;

   if (error < E_LN) {
      if (error_rate < ER_LN) { u = MLT[0][0]; }
      else if (error_rate < ER_MN && error_rate >= ER_LN) { u = MLT[0][1]; }
      else if (error_rate < ER_SN && error_rate >= ER_MN) { u = MLT[0][2]; }
      else if (error_rate < ER_ZE && error_rate >= ER_SN) { u = MLT[0][3]; }
      else if (error_rate < ER_SP && error_rate >= ER_ZE)  { u = MLT[0][4]; }
      else if (error_rate < ER_MP && error_rate >= ER_SP)  { u = MLT[0][5]; }
      else if (error_rate < ER_LP && error_rate >= ER_MP)  { u = MLT[0][6]; } }
   else if (error < E_MN && error >= E_LN) {
      if (error_rate < ER_LN) { u = MLT[1][0]; }
      else if (error_rate < ER_MN && error_rate >= ER_LN) { u = MLT[1][1]; }
      else if (error_rate < ER_SN && error_rate >= ER_MN) { u = MLT[1][2]; }
      else if (error_rate < ER_ZE && error_rate >= ER_SN)  { u = MLT[1][3]; }
      else if (error_rate < ER_SP && error_rate >= ER_ZE)  { u = MLT[1][4]; }
      else if (error_rate < ER_MP && error_rate >= ER_SP)  { u = MLT[1][5]; }
      else if (error_rate < ER_LP && error_rate >= ER_MP)  { u = MLT[1][6]; }  }
   else if (error < E_SN && error >= E_MN) {
      if (error_rate < ER_LN) { u = MLT[2][0]; }
      else if (error_rate < ER_MN && error_rate >= ER_LN) { u = MLT[2][1]; }
      else if (error_rate < ER_SN && error_rate >= ER_MN) { u = MLT[2][2]; }
      else if (error_rate < ER_ZE && error_rate >= ER_SN)  { u = MLT[2][3]; }
      else if (error_rate < ER_SP && error_rate >= ER_ZE)  { u = MLT[2][4]; }
      else if (error_rate < ER_MP && error_rate >= ER_SP)  { u = MLT[2][5]; }
      else if (error_rate < ER_LP && error_rate >= ER_MP)  { u = MLT[2][6]; }  }
   else if (error < E_ZE && error >= E_SN) {
      if (error_rate < ER_LN) { u = MLT[3][0]; }
      else if (error_rate < ER_MN && error_rate >= ER_LN) { u = MLT[3][1]; }
      else if (error_rate < ER_SN && error_rate >= ER_MN) { u = MLT[3][2]; }
      else if (error_rate < ER_ZE && error_rate >= ER_SN)  { u = MLT[3][3]; }
      else if (error_rate < ER_SP && error_rate >= ER_ZE)  { u = MLT[3][4]; }
      else if (error_rate < ER_MP && error_rate >= ER_SP)  { u = MLT[3][5]; }
      else if (error_rate < ER_LP && error_rate >= ER_MP)  { u = MLT[3][6]; }  }
   else if (error < E_SP && error >= E_ZE) {
      if (error_rate < ER_LN) { u = MLT[4][0]; }
      else if (error_rate < ER_MN && error_rate >= ER_LN) { u = MLT[4][1]; }
      else if (error_rate < ER_SN && error_rate >= ER_MN) { u = MLT[4][2]; }
      else if (error_rate < ER_ZE && error_rate >= ER_SN)  { u = MLT[4][3]; }
      else if (error_rate < ER_SP && error_rate >= ER_ZE)  { u = MLT[4][4]; }
      else if (error_rate < ER_MP && error_rate >= ER_SP)  { u = MLT[4][5]; }
      else if (error_rate < ER_LP && error_rate >= ER_MP)  { u = MLT[4][6]; }  }
   else if (error < E_MP && error >= E_SP) {
      if (error_rate < ER_LN) { u = MLT[5][0]; }
      else if (error_rate < ER_MN && error_rate >= ER_LN) { u = MLT[5][1]; }
      else if (error_rate < ER_SN && error_rate >= ER_MN) { u = MLT[5][2]; }
      else if (error_rate < ER_ZE && error_rate >= ER_SN)  { u = MLT[5][3]; }
      else if (error_rate < ER_SP && error_rate >= ER_ZE)  { u = MLT[5][4]; }
      else if (error_rate < ER_MP && error_rate >= ER_SP)  { u = MLT[5][5]; }
      else if (error_rate < ER_LP && error_rate >= ER_MP)  { u = MLT[5][6]; }  }
   else if (error < E_LP && error >= E_MP) {
      if (error_rate < ER_LN) { u = MLT[6][0]; }
      else if (error_rate < ER_MN && error_rate >= ER_LN) { u = MLT[6][1]; }
      else if (error_rate < ER_SN && error_rate >= ER_MN) { u = MLT[6][2]; }
      else if (error_rate < ER_ZE && error_rate >= ER_SN)  { u = MLT[6][3]; }
      else if (error_rate < ER_SP && error_rate >= ER_ZE)  { u = MLT[6][4]; }
      else if (error_rate < ER_MP && error_rate >= ER_SP)  { u = MLT[6][5]; }
      else if (error_rate < ER_LP && error_rate >= ER_MP)  { u = MLT[6][6]; }  }
      return u;
}
```

Fig. 8.16 The detailed codes for the function GetLookup()

to do this selection based on the different error and error rate ranges defined in the header file **FZControl.h** and these ranges are defined based on the error and error rate ranges defined in Table 8.9. Thus, a sequence of **if** and **else if** selection structures is used to do this selection job.

(4) We do this selection by dividing error ranges into seven segments, and perform seven sub-selections based on the error rate ranges inside each error segment. Starting at line 4, the first error segment is: if the **error** is less than **E_LN**, we continue to check seven error rate ranges, **< ER_LN**, **ER_LN ~ ER_MN**, **ER_MN ~ ER_SN**, **ER_SN ~ ER_ZE**, **ER_ZE ~ ER_SP**, **ER_SP ~ ER_MP** and **ER_MP ~ ER_LP**, in lines 5 ~ 11. Based on each error rate range, the corresponded control output can be identified by the related element located at the lookup table **MLT[0, n]** and assigned to the returned variable **u**.

(5) The second error segment, **error** < **E_MN** && **error** > = **E_LN**, starts at line 12. If the velocity error is in this range, the seven sub-selections are performed to check seven error rate ranges that are identical with those inside the last error segment to select the corresponded control output **MLT[1, n]** from the lookup table in lines 13 ~ 19, respectively.

(6) In a similar way, the following five error segments and related seven sub-selections can be performed to select the corresponded control outputs **MLT[2, n]**, **MLT[3, n]**, **MLT[4, n]**, **MLT[5, n]** and **MLT[6, n]**, and assigned them to the returned variable **u** in lines 21 ~ 27, 29 ~ 35, 37 ~ 43, 45 ~ 51, and 53 ~ 59.

(7) Finally, the identified control output stored in the returned variable **u** is returned to the main program in line 61.

One point to be noted is that the motor rotation direction and the direction we defined in our program. In this project, we defined the motor positive rotating direction as Clockwise (**CW**) and the membership functions and control outputs defined in the lookup table are based on this definition. If you find that the direction is opposite or your simulation result is negative, you need to change the direction in your program to redefine this direction. Also, the membership function input definitions, such as **LN**, **MN**, **SN**, **SP**, **MP**, and **LP**, need to be inverted in polity.

Now one can set up environments to build and run the project in a similar sequence as we did in Sect. 8.3.1.3.

So far for all FLC related projects, we only discussed how to use off-line fuzzy inference system to get the fuzzy control outputs with a predefined or pre-calculated lookup table. This method has some shortcoming for some accurate and high-speed control applications. In the next section, we will concentrate on how to use an online fuzzy inference system to get real-time fuzzy control output to improve the controllability and control accuracy.

8.5 Apply Online Fuzzy Inference System to Obtain the Real-Time Outputs

As we discussed in Sect. 7.5.5 in Chap. 7, the online method has the real-time controllability. Both inputs and outputs membership functions are developed during the real-time processing of an actual application. Also, the lookup table elements are calculated in real-time based on the current actual inputs and outputs. In this method, only fuzzy rules are developed prior to the real applications. The advantage of this method is that higher control accuracy can be obtained for a process and the fuzzy output can be computed in real-time. The disadvantage of this method is that a longer processing time is needed and it is a little time-consuming process. But as the development of new computer technologies, today much faster CPUs are available, and the processing time is no longer a big issue for applying this method.

The key issue when using this online fuzzy inference system is to build an interface between the C and the MATLAB, and to be able to call a critical fuzzy evaluation function **evalfis()** from the C environment based on the current inputs in real time. Figure 8.17 shows a functional block diagram for this kind of accessing or calling of a MATLAB function from a C source file environment. The operational procedure for this kind of accessing includes:

(1) Build a user interface between the C source file and MATLAB environment.
(2) Design and build a FIS based on the actual control system.
(3) Using the following MATLAB scripts to build the target MATLAB function that will be called by C source file:

```
a = readfis('FZMotor');
u = evalfis([err, err_rate], a);
```

where **err** and **err_rate** are input variables coming from the C source file, and the variable **u** is the calculation result of the FIS based on both inputs and it is to be returned to the C source file. The **FZMotor** is the name of the selected FIS.

There are quite a few of methods that can be used to design and build a user interface between the C/C++ and MATLAB. Four different methods have been reported by the author of this book (**Ying Bai**, "*Applications Interface Programming Using Multiple Languages—A Windows Programmer's Guide*", ISBN: 0131003135, Prentice Hall, March 2003). Please refer to this reference to get more details about these methods.

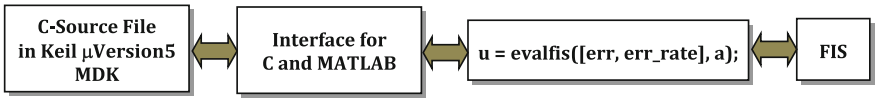


Fig. 8.17 Access MATLAB functions from the C source file

In the following section, we will concentrate one of those four methods, MATLAB engine method, to discuss how to build an interface between C/C++ and MATLAB to call a MATLAB function to access the FIS to get the fuzzy output value in real time.

8.5.1 *Build a Visual C++ Interface Project to Access MATLAB FIS*

The MATLAB engine method is similar to an Active-X method to dynamically access the MATLAB domain to setup, manipulate and retrieve variables' values via an Active-X engine library.

By using MATLAB engine library method, the interface between the C/C++ and MATLAB is relatively easier. Just like the Active-X control, after calling into the MATLAB environment to execute some functions in MATLAB, the C/C++ still keeps the controllability on the execution of the program. Another advantage for using MATLAB engine library is that we don't need to store any data into a data file in the C/C++ environment, and to retrieve this data by opening the data file in the MATLAB environment. Instead, we can create the data in the C/C++ programming and directly call the M-functions in the MATLAB by passing the data as an array string into the MATLAB domain.

We want to show an example project to illustrate how to use the MATLAB engine library method to access and manipulate data in MATLAB environments. However, before we can do that make sure that the following environments and installations have been setup:

- A Microsoft Visual Studio should have been installed in the computer. In this example, Microsoft Visual Studio.NET 2012 is used.
- A MATLAB has been installed in the computer. In this example, MATLAB R2015b that including Fuzzy Logic Toolbox has been installed. The default folder should be: **C:\Program Files\MATLAB\R2015b**. A fuzzy inference system **FZMotor.fis** has been built by using the Fuzzy Logic Toolbox (refer to Sect. 7.8.1 in Chap. 7 to build it).
- A Windows OS should be installed. In this example, Windows 7 SP 1 is used.

Now let's build our interface project **C_MATLAB_UI** by using the software Visual Studio.NET 2012 and MATLAB R2015b with Fuzzy Logic Toolbox.

8.5.1.1 Create and Build VC++ Interface Project C_MATLAB_UI

Open Visual Studio.NET 2012 to create a new Visual C++ Win32 console project named **C_MATLAB_UI**. As the **C_MATLAB_UI.cpp** file is opened, modify the main entry program header line to **int main(void)**.

```
1  //*****************************************************************************
2  // C_MATLAB_UI.h – Header file for project C_MATLAB_UI
3  //*****************************************************************************
4  #include "stdafx.h"
5  #include <stdlib.h>
6  #include <stdio.h>
7  #include <string.h>
8  #include <conio.h>
9  #include "engine.h"
10 #define BUFSIZE 256
11 double err = 0.08, err_rate = 0.04;          // testing err and err_rate values
12 static double Error[2] = { err, err_rate };
```

Fig. 8.18 The header file C_MATLAB_UI.h

Since the program is a little large in presentation, thus we divide it into two parts: a header file and a source file. In our real project, both files are combined together to have only one source file **C_MATLAB_UI.cpp**.

Enter codes that are shown in Fig. 8.18 into the header file **C_MATLAB_UI.h**. Save and add this header file as **C_MATLAB_UI.h** into the project.

In fact, the first or the top **#include** clause can be created and added into the source file **C_MATLAB_UI.cpp** automatically. The clause **<conio.h>** is used for the testing purpose and it can be deleted from the final normal project file. The **err** and **err_rate** are used to represent the input error and input error rate to the FLC, which will be placed into a data array **Error[]** that can be easily passed into the interface MATLAB function later.

Enter the codes that are shown in Fig. 8.19 into the source file **C_MATLAB_UI.cpp**. Let's take a look at this piece of codes to see how it works although most codes have been illustrated by related comments:

- The project header file **C_MATLAB_UI.h** is included in line 7 to make sure that the compiler can find the related symbols and prototypes for all structures and functions developed or built by the system, and used in this project.
- The MATLAB engine pointer ***ep** and two empty mxArray ***a** and ***u** are declared in lines 10 ~ 13. Both arrays are used to place and retrieve variables in/from MATLAB domain later. The double pointer variable ***result** is used to get back and store the returned fuzzy control output value **u** later. The **buffer[]** is a data array used to store all variable values to be transferred and placed into the MATLAB workspace later. All these variables must be declared as pointers since they are memory addresses, not variables.
- In line 17, a 1 × 2 double array is created and it is used to hold the **err** and **err_rate** values to be passed to the MATLAB workspace later. The related memory space for that data array is created and reserved for the **Error** array (source) including the **err** and the **err_rate** in line 18. All values in this array are copied into the destination memory space starting at address **a**. The function **mxGetPr(a)** gets the starting address of the target memory space.
- This **Error** data array, including its values (**err** and **err_rate**), is placed into the MATLAB workspace in line 20.

```
1  /*****************************************************************************
2  *  NAME: C_MATLAB_UI.cpp
3  *  FUNC: Interface between C++ and MATLAB to get FZ output
4  *  PRGM: Y. Bai
5  *  DATE: 2-8-2018
6  *****************************************************************************/
7  #include "C_MATLAB_UI.h"
8  int main(void)
9  {
10   Engine *ep;
11   mxArray *a = NULL, *u = NULL;
12   double *result;
13   char buffer[BUFSIZE+1];
14   /* start the MATLAB engine  */
15   if (!(ep = engOpen(NULL))) { printf("Cannot start MATLAB engine!\n");  exit(-1); }
16   /* create 1 by 2 array to store err & err_rate  */
17   a = mxCreateDoubleMatrix(1, 2, mxREAL);
18   memcpy((char *) mxGetPr(a), (char *) Error, 2*sizeof(double));
19   /* place err & err_rate array in MATLAB workspace  */
20   engPutVariable(ep, "E", a);
21   /* call MATLAB function to calculate the fz output value */
22   engEvalString(ep, "u = getFZOutput(E);");
23   /* use engOutputBuffer to capture MATLAB output. Ensure the buffer is always NULL terminated  */
24   buffer[BUFSIZE] = '\0';
25   engOutputBuffer(ep, buffer, BUFSIZE);
26   /* the evaluate string returns the result into the output buffer  */
27   engEvalString(ep, "whos");
28   printf("MATLAB - whos\n");
29   /* get the fuzzy control output u in mxArray  */
30   u = engGetVariable(ep, "u");
31   if (u == NULL) { printf("Get Array Failed\n"); }
32   else
33   {
34       result = mxGetPr(u);
35       printf("Control ouput u =: %f",result[0]);
36       mxDestroyArray(u);
37    }
38   engClose(ep);
39   mxDestroyArray(a);
40   while( !_kbhit() ){}                    // testing purpose, can be removed in normal program
41   return 0;
42 }
```

Fig. 8.19 The detailed codes for the source file C_MATLAB_UI.cpp

- A MATLAB function **getFZOutput()** is called in line 22 to obtain the desired control output value for the FIS named **FZMotor.fis**. The MATLAB Command used to call this function is transferred from the VC++ to MATLAB environment via a Text String, and the variable **u** is used to retrieve the collected fuzzy control output value.
- The codes in lines 26 ~ 28 are only used for the testing purpose, and they can be removed from the final normal project to speed up the program running.
- In line 30, the calculated fuzzy control output is picked up from the MATLAB workspace by using **engGetVariable()** function. A point to be noted is that this value is an address or a pointer, not a data value.
- After checking of this pointer, the starting address of this returned array is assigned to the variable pointer **result** in line 34 via the function **mxGetPr()**. The first data value stored in that array is displayed in line 35. In fact, this

coding line can be ignored in the final normal project source file since it is only for the testing purpose.

- Some cleaning jobs are performed in lines 36 ~ 39.
- The coding line 40 is only for the testing purpose to enable the Prompt Command Window to be stayed to show the fuzzy control output value for a while until a key is pressed by the user. This line can be ignored in the final program file.

Now we need to setup the appropriate environment variables to build and run this project to successfully access MATLAB workspace later.

8.5.1.2 Setup Environmental Variables to Build the Project

Perform the following operational steps to setup these environmental variables:

(1) Go to **PROJECT→C_MATLAB_UI Properties** menu item to open the **C_MATLAB_UI Property Pages** dialog. Under the **Configuration Properties** item in the left pane, select the **VC++ Directories** item. In the Include Directories line, click on the drop-down arrow and select the **<Edit…>** to open the Include Directories dialog, as shown in Fig. 8.20. Click on the **New Line** button and three dots button to go to the desired Include folder that contains MATLAB engine library header file, **engine.h**, which is **C:\Program Files\MATLAB\R2015b\extern\include** in your computer.

(2) Go to **PROJECT→C_MATLAB_UI Properties** menu item to open the **C_MATLAB_UI Property Pages** dialog. Under the **Configuration Properties** item in the left pane, select the **Linker→General** item, and go to

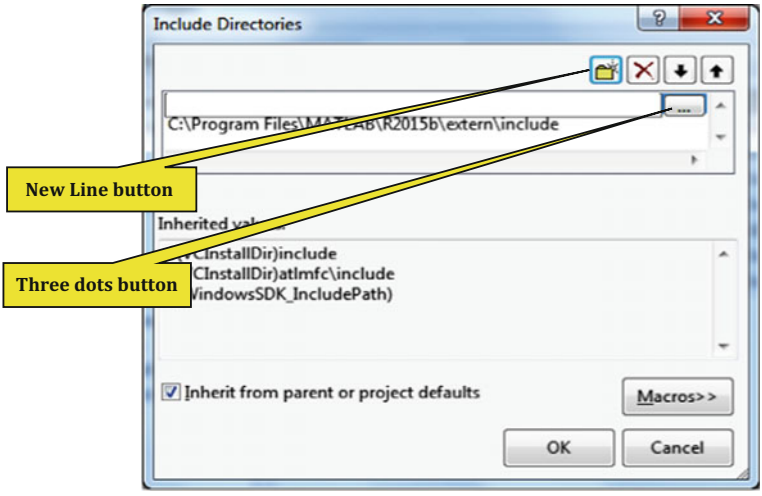


Fig. 8.20 The opened include Directories dialog

Additional Library Directories line. Click on the drop-down arrow and select the **<Edit… >** to open the **Additional Library Directories** dialog, which is similar to the **Include Directories** dialog shown in Fig. 8.20. Click on the **New Line** button and three dots button to go to the desired Library folder that contains MATLAB engine library files, **libeng.lib** and **libmx.lib**, which is **C:\Program Files\MATLAB\R2015b\extern\lib\win32\microsoft** in your computer.

(3) Still in the **C_MATLAB_UI Property Pages** dialog, select the **Linker→Input** item and go to the **Additional Dependencies** line on the top. Click on the dropdown arrow and select the **<Edit…>** to open the **Additional Dependencies** dialog. Enter **libeng.lib** and **libmx.lib** into the top box. Press the **OK** button to complete these VC++ setups.

(4) The last setup is the most important one and must be executed in the System Environment Variables setup dialog. This setup will allow VC++ to locate and find the dynamic libraries used in the project, such as **libeng.dll** and **libmx.dll**. Click on the **Start** icon located at the lower-left corner on your computer to open the **Control Panel→ System and Security→System** dialog. Click on the **Advanced system settings** link to open the **System Properties** dialog, and click on the **Environment Variables** button to open the Environment Variables dialog. Under the **System variables** group, click on the **New** button to add a new variable, **Variable name**: **PATH**, **Variable value**: **C:\Program Files\MATLAB\R2015b\bin\win32**.

The path setup in step 4 is used to reference on the location where your library files **libeng.dll** and **libmx.dll** are located. An important point is that it may require you to restart your Visual Studio.NET 2012 or even restart your computer to make these setups effective.

Now you can build the project and link all required libraries with no problem. However, before you can run this project, you need to first create a MATLAB interface function **getFZOutput()**. Next, let's take care of building the MATLAB interface function.

8.5.1.3 Build MATLAB Interface Function GetFZOutput()

The major role of this function is to work as an interface or a bridge between VC++ and MATLAB domain to transfer necessary information between two domains. In fact, this function needs to read a FIS "**FZMotor.fis"** and execute an evaluation function **evalfis()** to calculate the fuzzy output based on the current two inputs, **err** and **err_rate**.

Open MATLAB with a new script file and enter the codes shown in Fig. 8.21 into this function. Save this file as **getFZOutput.m** to the default MATLAB folder.

Fig. 8.21 Detailed codes for the MATLAB function getFZOutput()

```
% C-MATLAB UI Function getFZOutput()
% Inputs: err & err_rate
% Output: u – current fuzzy output
function u = getFZOutput(eInput)
a = readfis('FZMotor');
u = evalfis(eInput, a);
end
```

The first point to be noted for this piece of codes is that the name of this script file and the name of this function must be identical, which is a requirement by MATLAB coding process.

The second point to be noted is that the Input to this function is a 1 × 2 array containing the **err** and **err_rate** values. The FIS **FZMotor.fis** should have already been built with Fuzzy Logic Toolbox and Exported into a file named **FZMotor.fis**.

Now you can run the VC++ project **C_MATLAB_UI** in the Visual Studio.NET 2012 IDE to test this interface function. Based on the current testing **err** and **err_rate** values, 0.08 and 0.04, the calculated fuzzy output should be **u = 475.1369**. You can test and confirm this value by using the Fuzzy Logic Toolbox with **FZMotor** FIS.

8.5.2 Build a Complete Project to Get Online Fuzzy Output

With the help of the interface function built in the last section, we can develop that interface function as a single or independent function and insert it into the complete main fuzzy logic control program to obtain the online fuzzy output in real time.

Let's develop that interface function as an independent C++ function file to be called by the main fuzzy control program built later.

First, create a main C++ console Win32 project **MC_MATLAB_UI** with Visual Studio.NET 2012. Now you do not need to worry about the main C++ file **MC_MATLAB_UI.cpp** and we will take care of it later. Right click on the new created project **MC_MATLAB_UI** in the Solution Explorer window, and select **Add New Item** link to add a new C++ File named **CMFZOutput.cpp**. Enter the codes shown in Fig. 8.22 into this source file.

All of the codes in this function are identical or similar to those in the project **C_MATLAB_UI** discussed in the last section. Only the following points need to be noted:

(1) The system header file **stdafx.h** is needed for each C++ file, and regularly it is created by the system automatically when a new C++ project is created. But for any new created C++ or Header file, you need to add it yourself.
(2) The user header file **MC_MATLAB_UI.h** will be built later and it only contains some other useful system header files and some global variables.

```
/*********************************************************************************
* NAME:    CMFZOutput() function
* PRGMR:  Y. Bai
* DATE:    2-11-2018
*********************************************************************************/
#include "stdafx.h"
#include "MC_MATLAB_UI.h"

double CMFZOutput(Engine *ep, double errorInput[])
{
    double *result = NULL, fvalue;
    char buffer[BUFSIZE+1];
    mxArray *a = NULL, *u = NULL;

    a = mxCreateDoubleMatrix(1, 2, mxREAL);          // create 1 by 2 array to store err & err_rate
    memcpy((char *) mxGetPr(a), (char *) errorInput, 2*sizeof(double));
    engPutVariable(ep, "E", a);                      // place err & err_rate array in MATLAB wkspace

    /* Call MATLAB function to calculate the fz output value */
    engEvalString(ep, "u = getFZOutput(E);");

    /* Use engOutputBuffer to capture MATLAB output. Ensure the buffer is always NULL terminated  */
    buffer[BUFSIZE] = '\0';
    engOutputBuffer(ep, buffer, BUFSIZE);

    /* Get the fuzzy control output u in mxArray  */
    u = engGetVariable(ep, "u");

    if (u == NULL) { printf("Get Array Failed\n"); }
    else {  result = mxGetPr(u);
            printf("Control ouput u =: %f",result[0]);
            fvalue = result[0];
            mxDestroyArray(u); }
    mxDestroyArray(a);

    return fvalue;
}
```

Fig. 8.22 The complete codes for the function CMFZOutput()

(3) A MATLAB Engine pointer ***ep** is passed as an argument into this function. This is necessary since this pointer should be created in the main program and will be used in this function, therefore, it must be passed into this function as a pointer.

(4) The double variable **fvalue** works as an intermediate variable to return the obtained fuzzy output value stored in a pointer variable **result** to the main program later. One cannot directly return the variable **result[0]** to the main program since it should have been destroyed in the *else* clause when the instruction **mxDestroyArray(u)** is executed. Therefore, we must reserve its value to the **fvalue** before that destroying to be executed.

Now add a new header file into this project and name it as **MC_MATLAB_UI.h**, and enter the codes shown in Fig. 8.23 into this header file.

It is the time for us to develop our main program file **MC_MATLAB_UI.cpp**. Since we used a function to perform the major job to get fuzzy output, thus the main program is very simple. Enter the codes shown in Fig. 8.24 into this main C++ file.

The explanations for this piece of codes have been provided in the last section when we developed the project **C_MATLAB_UI**. The following points should be paid attention to:

```
/*****************************************************************************
  NAME: MC_MATLAB_UI.h
*  FUNC: Header file for the main program MC_MATLAB_UI
*  PRGM: Y. Bai
*  DATE: 2-11-2018
*****************************************************************************/
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <conio.h>
#include "engine.h"
#define BUFSIZE 256
```

Fig. 8.23 The header file MC_MATLAB_UI.h

```
/*****************************************************************************
  NAME: MC_MATLAB_UI.cpp
*  FUNC: Main program to call interface function CMFZOutput()
*  PRGM: Y. Bai
*  DATE: 2-11-2018
*****************************************************************************/
#include "stdafx.h"
#include "MC_MATLAB_UI.h"
Engine *ep;
extern double CMFZOutput(Engine *ep, double Error[]);
int main(void)
{
    double ret = 0;
    double err = 0.08, err_rate = 0.04;                  // testing err and err_rate values
    static double Error[2] = { err, err_rate };
    /* Start the MATLAB engine   */
    if (!(ep = engOpen(NULL))) { printf("Cannot start MATLAB engine!\n");
       exit(-1);
    }
    /* Call function CMFZOutput() to calculate the fz output value */
    ret = CMFZOutput(ep, Error);
    if (ret == 0)
       printf("Error in calling function CMFZOutput()!\n");
    else
       printf("\nReturned calculated fuzzy output value is: %f", ret);
    engClose(ep);
    while( !_kbhit() ){}                                 // testing purpose, can be removed in normal program
    return 0;
}
```

Fig. 8.24 The detailed codes for the main program MC_MATLAB_UI.cpp

(1) The MATLAB Engine pointer ***ep** should be declared as a global variable since both the main program and the function file will use it.
(2) Since the function **CMFZOutput()** is located at a separate single C++ file **CMFZOutput.cpp**, therefore the system keyword **extern** must be prefixed before this function declaration in the main program to tell the compiler where this function is located and how can find it in this project when the entire project is compiled later.
(3) Similarly, the system header file **stdafx.h** must be added and included by the user since this header file is needed by each C++ file.

Now one needs to set up the environment variables for this project as we did in Sect. 8.5.1.2, then the project can be run to access the MATLAB FIS to obtain the desired fuzzy output value via MATLAB Engine library.

8.6 Summary

The major topic on this chapter is the implementation of fuzzy logic controller (FLC) on any closed-loop control system by using an actual microcontroller with its programming technologies. Starting Sect. 8.2, three main operational steps on how to successfully implement a fuzzy logic controller on any closed-loop system are introduced and discussed: Fuzzifications, Control Rule Designs, and Defuzzifications. An actual DC motor closed-loop control system is used as an example to illustrate how to design and implement a FLC on that system with real C programming control codes step by step in Sect. 8.3. A mapping technique used between a real fuzzy logic output lookup table and a 2D matrix in C Programming language is discussed with detailed illustrations and example coding process.

To make this kind of implementation more impressive and practical in applications, another DC motor system is studied and analyzed in Sect. 8.4 as a special case study. All fuzzy control strategies used in Sects. 8.3 and 8.4 belong to off-line fuzzy technique and the fuzzy output lookup table must be calculated in advanced before the project runs.

Starting Sect. 8.5, an online fuzzy output strategy is introduced and implemented by designing a user interface (UI) between the VC++ and MATLAB via MATLAB Engine library method to enable the VC++ to access the FIS built in the MATLAB environments. Both the programming codes in the VC++ side and the interface function designed in the MATLAB domain are presented and discussed step by step with very detailed descriptions. A complete control program is designed and presented with both a main program and an interface function to make the control more objective and integrated.

Homework and Labs

Lab Project 8.1
Develop an off-line lookup table based on MFs and control rules given in HW7.2 in Chap. 7. The relationships between both inputs, temperature, and humidity, and the output, fan speed, are shown in Table 8.10.

Based on 12 control rules shown in HW7.2, an input–output relationship table can be developed and shown in Tables 8.11 and 8.12. This table can be considered as a 2D matrix and each intersection cell is the control output **fan_speed** or **f_sp (i, j)**.

Refer to Fig. 8.12 to develop this lookup table matrix using MATLAB Script **Lab8_1.m**.

Table 8.10 Ranges of inputs and the output of the heating-fan system

Input temperature (°F)	<30	30 ~ 60	45 ~ 75	>75
MF	Cold	Cool	Warm	Hot
Input humidity	<50	50 ~ 180		>180
MF	Low	Med		High
Output fan velocity	10	20	30	40
MF	ZE	LOW	MED	HIGH

Table 8.11 The control rules

T	H		
	Low	Med	High
Cold	MED	MED	HIGH
Cool	LOW	LOW	MED
Warm	ZE	ZE	LOW
Hot	ZE	ZE	ZE

Table 8.12 The input errors

T	H		
	Low	Med	High
Cold	$u(0,0)$	$u(0,1)$	$u(0,2)$
Cool	$u(1,0)$	$u(1,1)$	$u(1,2)$
Warm	$u(2,0)$	$u(2,1)$	$u(2,2)$
Hot	$u(3,0)$	$u(3,1)$	$u(3,2)$

Lab Project 8.2
Refer to Fig. 8.16 to develop a C-Function **GetLookUpTable()** to obtain the desired fuzzy output from the lookup table based on two inputs, **Temp**, and **Humd**. Show your complete codes for this function. One can create and build a VC ++ file to save this function named **Lab8_2.cpp**.

Lab Project 8.3
Refer to Sect. 8.5.2 to build a VC++ interface function **VC_GetFZOutput()** to call a MATLAB interface function **GetFuzzyOutput()** (Sect. 8.5.1.3) to access the FIS **evalfis()** function to get desired fuzzy online output based on inputs, **Temp** and **Humd**. Provide the complete VC++ codes and MATLAB Script codes for both functions. This VC++ function can be added into the main project **VC_MATLAB** built in the next Lab Project. One can first do Lab Project 8.4 and then add this function as an independent C++ file into that main project.

Lab Project 8.4
Refer to Sect. 8.5.2 to build a VC++ main project **VC_MATLAB** to call the interface function **VC_GetFZOutput()** built in the last Lab Project. Provide the complete VC++ codes for this project. One can combine the function **VC_GetFZOutput()** built in the last Lab Project with this main project together.

Chapter 9
Interval Type-2 Fuzzy Logic Controllers



In the point of view of control types, the fuzzy logic control technology can be divided into two categories: type-1 and type-2. So far, the fuzzy logic control technologies discussed in all previous chapters in this book belong to the type-1 fuzzy logic controls. In type-1 FLC, each membership function has a definite value or a membership degree based on the current input or output value. Also, the control rules have a two-to-one, multi-to-one, or multi-to-multi definite single-value relationship, and the fuzzy inference system output is a crisp single-value output. However, some shortcomings are involved with this kind of fuzzy inference system. Since the actual membership functions may be varied in a range, but not a definite single degree value. Therefore, a type-2 fuzzy inference system may be used to provide a more accurate or more practical approach to the real situations. Among type-2 fuzzy logic control technologies, the interval type-2 fuzzy inference system (IT2FIS) is a relatively simple and practical technique to be studied and implemented in most systems, which is the main topic of this chapter.

9.1 Introduction to Type-1 and Interval Type-2 Fuzzy Sets

At Sect. 7.2 in Chap. 7, we have provided detailed discussions about the type-1 fuzzy set and its three basic operations. All members of a type-1 fuzzy set are definite numbers with certain values. For example, for a room heating type-1 fuzzy inference system, an **L** (LOW) fuzzy set that is shown in Fig. 9.1a includes the following members in the f domain:

$$\mathbf{L} = \{0/20, 0.5/30, 1/40, 0.5/50, 0/60\} \tag{9.1}$$

where the first number 0/20 means that 20 °F has a membership degree of 0 in this type-1 fuzzy set f, and the second number 0.5/30 indicates that the 30 °F has a membership degree of 0.5 in this set f, etc. In other words, for each member in this

© Springer Nature Switzerland AG 2019

Y. Bai and Z. S. Roth, *Classical and Modern Controls with Microcontrollers*,
Advances in Industrial Control, https://doi.org/10.1007/978-3-030-01382-0_9

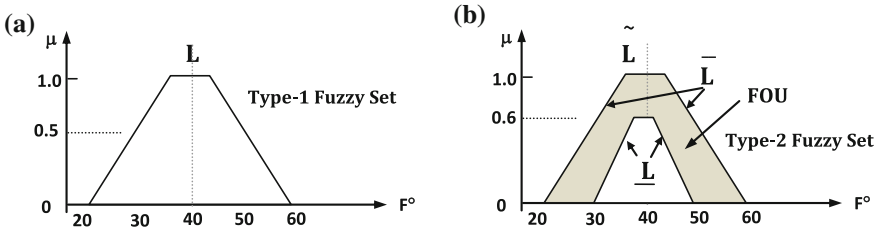


Fig. 9.1 An example of type-1 and type-2 fuzzy sets

set, it has its own definite crisp membership degree value. Now let us take a look at a similar type-2 fuzzy set that is shown in Fig. 9.1b.

Comparing with type-1 fuzzy set L, the type-2 fuzzy set is indicated as $\tilde{L}$ with a tilde symbol on the top of the set name. Also, each member in the fuzzy set is no longer a crisp value, but instead it is an interval or a range. For example, as shown in Fig. 9.1b, the membership for 30 °F is [0, 0.6], and the membership for 40 °F is [0.6, 1]. Observe also that the interval type-2 (IT2) fuzzy set (FS) shown in Fig. 9.1b is bounded from the above and below by two type-1 fuzzy set (T1 FS) $\overline{L}$ and $\underline{L}$, which are called upper membership function (UMF) and lower membership function (LMF), respectively. The area between $\overline{L}$ and $\underline{L}$ is called footprint of uncertainty (FOU).

Type-2 fuzzy sets are used for modeling uncertainty and imprecision in a better way compared with T1 FS. These type-2 fuzzy sets were originally presented by Zadeh in 1975 and are essentially "fuzzy fuzzy" sets, where the fuzzy degree of membership is a type-1 fuzzy set. The new concepts introduced by Mendel and Liang allow the characterization of an interval type-2 fuzzy set with an UMF and an LMF; these two functions can be represented each one by a type-1 fuzzy set membership function.

The uncertainty is the imperfection of knowledge about the natural process or natural system. The statistical uncertainty is the randomness or error that comes from different sources as we use it in a statistical methodology.

In fact, an IT2 FS expresses the non-deterministic truth degree with imprecision and uncertainty for an element that belongs to a set. An IT2 FS denoted by $\tilde{A}$ is characterized by a type-2 membership function $\mu_{\tilde{A}}(x,u)$, where $x \in X$, $u \in J_x^u \subseteq [0,1]$, and $0 \leq \mu_{\tilde{A}}(x,u) \leq 1$, as defined in Eq. (9.2)

$$\tilde{A} = \{(x, \mu_{\tilde{A}}(x) | x \in X\}; \quad \tilde{A} = \{(x, u, \mu_{\tilde{A}}(x,u)) | \forall x \in X, \forall u \in J_x^u \subseteq [0,1]\} \tag{9.2}$$

If $f_x(u) = 1$, $\forall u \in [\underline{J_x^u}, \overline{J_x^u}] \subseteq [0,1]$, this is an IT2 FS and the membership function $\mu_{\tilde{A}}(x,u)$ can be expressed by one type-1 UMF, $\overline{J_x^u} = \overline{\mu}_{\tilde{A}}(x)$ and one type-1 LMF, $\underline{J_x^u} = \underline{\mu}_{\tilde{A}}(x)$ as shown in Fig. 9.2. Then, this type-2 fuzzy set can be denoted by Eq. (9.3)

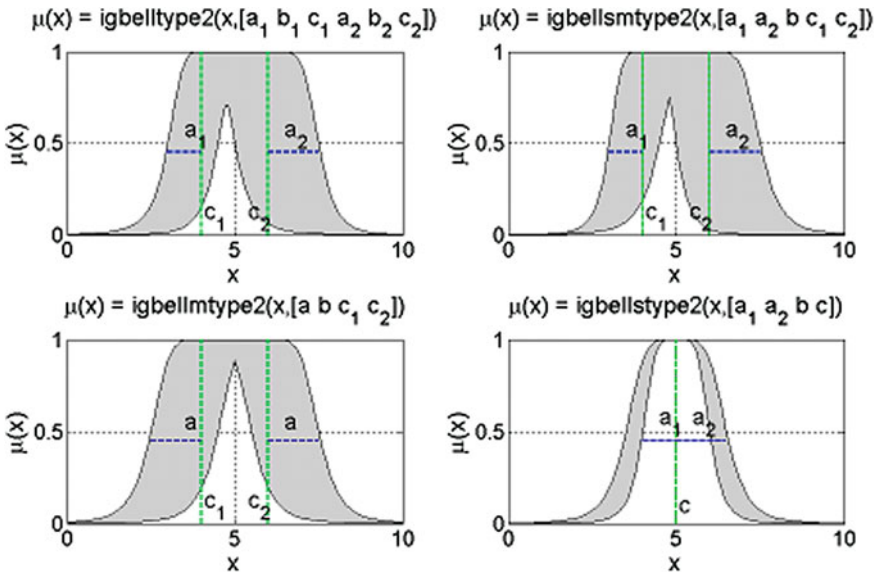


Fig. 9.2 FOU of GBell interval type-2 membership functions

$$\tilde{A} = \begin{cases} (x, u, 1) | \forall x \in X \\ \forall u \in \left[\underline{\mu_A}(x), \overline{\mu_A}(x) \right] \subseteq [0, 1] \end{cases} \tag{9.3}$$

In Fig. 9.2, a Gaussian Bell curve is used for this membership function and the related FOU is presented with different parameters, such as a_i, b_i, and c_i.

9.2 Interval Type-2 Fuzzy Sets Operations

Most fundamental operations in the interval type-2 fuzzy sets are Union (9.4), Intersection (9.5), and Complement (9.6). The union for interval type-2 fuzzy sets $\tilde{A}$ and $\tilde{B}$ is

$$\tilde{A} \sqcup \tilde{B} = \left\{ \int_{x \in X} \mu_{\tilde{A}}(x) \sqcup \mu_{\tilde{B}}(x) / x \right\} = \left\{ \int_{x \in X} \left[\int_{\alpha \in \left[\underline{\mu}_{\tilde{A}(x)} \vee \underline{\mu}_{\tilde{B}(x)}, \overline{\mu}_{\tilde{A}(x)} \vee \overline{\mu}_{\tilde{B}(x)} \right]} 1/\alpha \right] / x \right\} \tag{9.4}$$

The intersection for interval type-2 fuzzy sets $\tilde{A}$ and $\tilde{B}$ is

$$\tilde{A} \sqcap \tilde{B} = \left\{ \int_{x \in X} \mu_{\tilde{\tilde{A}}}(x) \sqcap \mu_{\tilde{\tilde{B}}}(x) / x \right\} = \left\{ \int_{x \in X} \left[\int_{\alpha \in \left[\underline{\mu}_{\tilde{A}(x)} \wedge \underline{\mu}_{\tilde{B}(x)}, \overline{\mu}_{\tilde{A}(x)} \wedge \overline{\mu}_{\tilde{B}(x)} \right]} 1/\alpha \right] / x \right\} \tag{9.5}$$

The complement for interval type-2 fuzzy set $\tilde{A}$ is

$$\neg \tilde{A} \left\{ \int_{x \in X} \mu_{\neg \tilde{\tilde{A}}}(x) / x \right\} = \left\{ \int_{x \in X} \left[\int_{\alpha \in \left[1 - \overline{\mu}_{\tilde{A}}(x), 1 - \underline{\mu}_{\tilde{A}}(x) \right]} 1/\alpha \right] / x \right\} \tag{9.6}$$

As we mentioned, for an interval type-2 fuzzy set that third dimension value is the same (e.g., 1) everywhere, which means that no new information is provided in the third dimension of an interval type-2 fuzzy set. Thus, for such a set, the third dimension is ignored, and only the FOU is used to describe an interval or a range of a MF with UMF and LMF. It is for this reason that an interval type-2 fuzzy set is sometimes called a *first-order uncertainty* fuzzy set model, whereas a general type-2 fuzzy set (with its useful third-dimension) is sometimes referred to as a *second-order uncertainty* fuzzy setuncertainty fuzzy set model.

The FOU represents the blurring of a type-1 membership function, and is completely described by its two bounding functions, a lower membership function (LMF) and an upper membership function (UMF), both of which are type-1 fuzzy sets. Consequently, it is possible to use type-1 fuzzy set mathematics to characterize and work with interval type-2 fuzzy sets.

9.3 Interval Type-2 Fuzzy Logic System

As a type-1 fuzzy logic system, an interval type-2 fuzzy logic system also needs a three-operational sequence to build a fuzzy logic control system:

(1) Fuzzificator-Fuzzifications
(2) Fuzzy Control Rule-Fuzzy Inference System
(3) Type Reducer-Defuzzificator-Defuzzifications

Figure 9.3 shows a functional block diagram of a typical interval type-2 fuzzy logic system. Both inputs and output to this IT2 Fuzzy Logic System (FLC) are

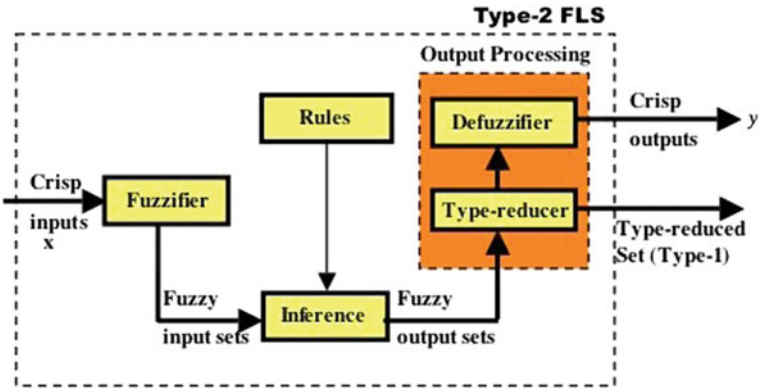


Fig. 9.3 A functional block diagram of an interval type-2 fuzzy logic system

crisp variables. However, unlike type-1 FLC, in which all MFs and control rule variables are crisp variables, in an IT2 FLC, all MFs and control rule variables, including both inputs and output, are fuzzy sets described and ranged by an Upper bound and a Lower bound membership functions. Hence, the outputs of the inference engine are IT2 FSs, and a type-reducer is needed to convert them into a T1 FS before defuzzification process can be carried out.

In practice, the computations in an IT2 FLC can be significantly simplified. Consider the rule base of an IT2 FLC consisting of N rules assuming the following form:

$$R^n : \text{IF } x_1 \text{ is } \tilde{X}_1^{\,n} \text{ and} \ldots \text{ and } x_i \text{ is } \tilde{X}_i^{\,n}, \text{ then } y \text{ is } Y^n; \quad n = 1, 2 \ldots N,$$

where $\tilde{X}_i^n (i = 1 \sim I)$ are IT2 Fuzzy sets, and $Y^n = \left[\underline{y^n}, \overline{y^n}\right]$ is an interval, which can be understood as the centroid of a consequent Interval Type-2 fuzzy set, or the simplest TSK model, for its simplicity.

Assume the input vector is $x' = (x'_1, x'_2, \ldots, x'_1)$. Similar to a type-1 FLC, typical computations in an IT2 FLC involve the following steps:

(1) Compute the membership of x'_i on each X_i^n to get the LMF and UMF, $\left\{\underline{\mu}(x'), \overline{\mu}(x')\right\}$.

(2) Compute the firing interval of the nth rule, $F^n(\mathbf{x}')$, which is

$$F^n(\mathbf{x}') = \left[\underline{\mu}(x'_1) \times \underline{\mu}(x'_2) \times \cdots \underline{\mu}(x'_I), \overline{\mu}(x'_1) \times \overline{\mu}(x'_2) \times \cdots \overline{\mu}(x'_I)\right] = \left[\underline{f^n}, \overline{f^n}\right].$$

(3) Perform type-reduction to combine $F^n(\mathbf{x}')$ and the corresponding rule consequents with the center-of-sets type-reducer:

$$Y_{\cos}(\mathbf{x}') = \bigcup_{\substack{f^n \in F^n(\mathbf{x}') \\ y^n \in Y^n}} \frac{\sum_{n=1}^{N} f^n y^n}{\sum_{n=1}^{N} f^n} = [y_l, y_r] \tag{9.7}$$

(4) Compute the defuzzified output as

$$y = \frac{y_l + y_r}{2} \tag{9.8}$$

On step (3), in order to calculate y_l and y_r, the following equations are used:

$$\begin{aligned} y_l &= \min_{k \in [1, N-1]} \frac{\sum_{n=1}^{k} \overline{f}^n \underline{y}^n + \sum_{n=k+1}^{N} \underline{f}^n \underline{y}^n}{\sum_{n=1}^{k} \overline{f}^n + \sum_{n=k+1}^{N} \underline{f}^n} \equiv \frac{\sum_{n=1}^{L} \overline{f}^n \underline{y}^n + \sum_{n=L+1}^{N} \underline{f}^n \underline{y}^n}{\sum_{n=1}^{L} \overline{f}^n + \sum_{n=L+1}^{N} \underline{f}^n} \\ y_r &= \max_{k \in [1, N-1]} \frac{\sum_{n=1}^{k} \underline{f}^n \overline{y}^n + \sum_{n=k+1}^{N} \overline{f}^n \overline{y}^n}{\sum_{n=1}^{k} \underline{f}^n + \sum_{n=k+1}^{N} \overline{f}^n} \equiv \frac{\sum_{n=1}^{R} \underline{f}^n \overline{y}^n + \sum_{n=R+1}^{N} \overline{f}^n \overline{y}^n}{\sum_{n=1}^{R} \underline{f}^n + \sum_{n=R+1}^{N} \overline{f}^n} \end{aligned} \tag{9.9}$$

where the switch points, **L** and **R**, can be determined by using the Karnik–Mendel (**KM**) algorithms

$$\begin{aligned} \underline{y}^L \leqslant y_l \leqslant \underline{y}^{L+1} \\ \overline{y}^R \leqslant y_r \leqslant \overline{y}^{R+1} \end{aligned} \tag{9.10}$$

9.4 Karnik–Mendel (KM) Algorithms

As we mentioned, the purpose of using the Karnik–Mendel (KM) algorithm is to effectively find the switch points for the left and right bounds in an output set. This algorithm can be divided into two separate processes based on left and right bound. First, let us take care of finding the left bound y_l. Basically, this algorithm is composed of the following operational steps:

(1) Sort $\underline{y}^n (n = 1, 2, \ldots N)$ in an increment order to make it as a new set, $\underline{y}^1 \leq \underline{y}^2 \leq \cdots \leq \underline{y}^N$. Match the firing weights $\mathbf{F^n}(x')$ with their corresponding $\underline{y}^n$ in the above new set and renumber them to make their index correspond to the renumbered new set $\underline{y}^n$.

(2) Initialize f^n by setting

$$f^n = \frac{\underline{f}^n + \overline{f}^n}{2} \quad n = 1, 2, 3. \ldots N \tag{9.11}$$

(3) Calculate the output set y as

$$y = \frac{\sum_{n=1}^{N} \underline{y}^n f^n}{\sum_{n=1}^{N} f^n} \tag{9.12}$$

(4) Find the switch point k ($1 \leq k \leq N-1$) such that

$$\underline{y}^k \leq y \leq \underline{y}^{k+1} \tag{9.13}$$

(5) Now set

$$f^n = \begin{cases} \underline{f}^n, n \leq k \\ \overline{f^n}, n > k \end{cases} \tag{9.14}$$

(6) Calculate the value for y' as

$$y' = \frac{\sum_{n=1}^{N} \underline{y}^n f^n}{\sum_{n=1}^{N} f^n} \tag{9.15}$$

(7) Check if $y' = y$. If yes, stop and set $y_l = y$ and $L = k$. If no, go to Step 8.
(8) Set $y = y'$ and go to Step 4.

Now let us take care of the algorithm to find the right bound y_r. The algorithm is also composed of the following steps:

(1) Sort $\overline{y}^n (n = 1, 2, \ldots N)$ in an incrementing order to make it as a new set, $\overline{y}^1 \leq \overline{y}^2 \leq \cdots \leq \overline{y}^N$. Match the firing weights $\mathbf{F^n}(x')$ with their corresponding $\underline{y}^n$ in the above new set and renumber them to make their index corresponding to the renumbered new set $\overline{y}^n$.
(2) Initialize f^n by setting

$$f^n = \frac{\underline{f}^n + \overline{f}^n}{2} \quad n = 1, 2, 3 \ldots N \tag{9.16}$$

(3) Calculate the output set y as

$$y = \frac{\sum_{n=1}^{N} \overline{y}^n f^n}{\sum_{n=1}^{N} f^n} \tag{9.17}$$

(4) Find the switch point k ($1 \leq k \leq N-1$) such that

$$\overline{y}^k \leq y \leq \overline{y}^{k+1} \tag{9.18}$$

(5) Now set

$$f^n = \begin{cases} \underline{f}^n, n \le k \\ f^n, n > k \end{cases} \tag{9.19}$$

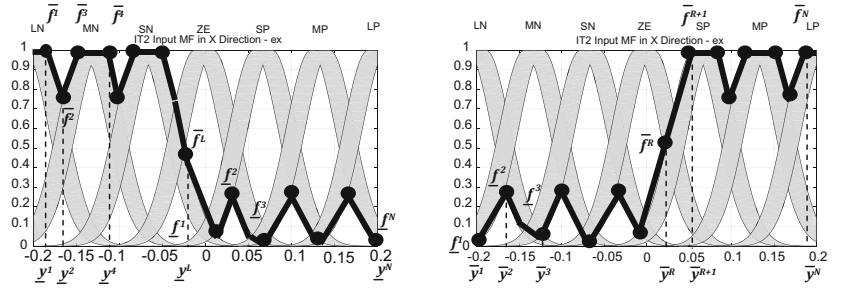
(6) Calculate the value for y' as

$$y' = \frac{\sum_{n=1}^{N} \bar{y}^n f^n}{\sum_{n=1}^{N} f^n} \tag{9.20}$$

(7) Check if $y' = y$. If yes, stop and set $y_r = y$ and $R = k$. If no, go to Step 8.
(8) Set $y = y'$ and go to Step 4.

The main idea of the **KM** algorithm is to find the switch points for y_l and y_r. Let us consider y_l and use an IT2 input MF (error) with seven elements as an example. The y_l is the minimum of $Y_{\cos}(\mathbf{x}')$. Since $\underline{y}^n$ increases from the left to the right along the horizontal axis of Fig. 9.4a, we should choose a large weight (upper membership grade) for $\underline{y}^n$ on the left and a small weight (lower membership grade) for $\underline{y}^n$ on the right. The KM algorithm finds the switch point L. For $n \le L$, the upper membership grades are used to calculate y_l; for $n > L$, the lower membership grades are used. This will ensure y_l be the minimum.

Similar process to y_r, the y_r should be the maximum of $Y_{\cos}(\mathbf{x}')$. Since $\bar{y}^n$ increases from the left to the right along the horizontal axis of Fig. 9.4b, we should choose a small weight (lower membership grade) for $\bar{y}^n$ on the left and a large weight (upper membership grade) for $\bar{y}^n$ on the right. The KM algorithm finds the switch point R. For $n \le R$, the lower membership grades are used to calculate y_r; for $n > R$, the upper membership grades are used. This will ensure y_r be the maximum.



(a) Compute y_l – switch from UMF to LMF. (b) Compute y_r – switch from LMF to UMF.

Fig. 9.4 Illustration of the switch points in computing y_l and y_r

9.5 An Example of an Interval Type-2 Fuzzy Logic System

To make the above IT2 FLS more senses, we try to use an example to illustrate how an IT2 FLC or FLS performs its inference function based on input MFs and control rules to derive the output fuzzy sets, and furthermore to obtain the crisp fuzzy output by using a type reducer process.

Consider an IT2 FLS with two inputs, **Input1** and **Input2**, and one **output**, output. Each input domain contains two IT2 fuzzy sets, **W**, **E** and **N**, **S**, as shown in Fig. 9.5.

Both fuzzy sets on two inputs are described by using the Gaussian type-2 curve, as shown in Fig. 9.5. Four control rules are applied to this IT2 FLS, and they are

(1) IF Input1 is W and Input2 is N, the output is Y^1.
(2) IF Input1 is W and Input2 is S, the output is Y^2.
(3) IF Input1 is E and Input2 is N, the output is Y^3.
(4) IF Input1 is E and Input2 is S, the output is Y^4.

The complete relationship between control rules and the corresponding outputs is shown in Table 9.1.

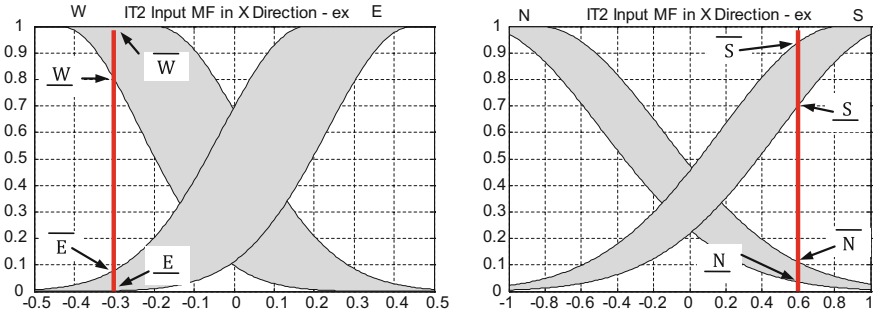
Suppose we have an input vector Input = {**Input1**, **Input2**} = {−0.3, 0.6}, based on above four control rules, the firing intervals of the four IT2 Fuzzy Sets are

$$[\mu_{\underline{W}}(-0.3), \mu_{\overline{W}}(-0.3)] = [0.8, 1.0] \tag{9.21}$$

$$[\mu_{\underline{E}}(-0.3), \mu_{\overline{E}}(-0.3)] = [0.0, 0.08] \tag{9.22}$$

$$[\mu_{\underline{N}}(0.6), \mu_{\overline{N}}(0.6)] = [0.04, 0.1] \tag{9.23}$$

$$[\mu_{\underline{S}}(0.6), \mu_{\overline{S}}(0.6)] = [0.7, 0.95] \tag{9.24}$$



(a) MF for Input1. (b) MF for Input2.

Fig. 9.5 Input membership functions for ITS example system

Table 9.1 Control rules and consequents of the IT2 FLS

Input1	Input2	
	N	**S**
W	$Y^1 = [\underline{y}^1, \overline{y}^1] = [-0.6, -0.3]$	$Y^2 = [\underline{y}^2, \overline{y}^2] = [-0.2, 0.2]$
E	$Y^3 = [\underline{y}^3, \overline{y}^3] = [0.2, 0.6]$	$Y^4 = [\underline{y}^4, \overline{y}^4] = [0.4, 0.8]$

According to step (2) in Sect. 9.3, the firing intervals of the four rules $F^n(\mathbf{x}') = [\underline{f}^n, \overline{f}^n]$ can be calculated as

(1) $\left[\underline{f}^1, \overline{f}^1\right] = \left[\mu_{\underline{W}}(-0.3) \times \mu_{\underline{N}}(0.6), \mu_{\overline{W}}(-0.3) \times \mu_{\overline{N}}(0.6)\right] = [0.8 \times 0.04, 1.0 \times 0.1] = [0.032, 0.1]$
$\rightarrow \left[\underline{y}^1, \overline{y}^1\right] = [-0.6, -0.3].$

(2) $\left[\underline{f}^2, \overline{f}^2\right] = \left[\mu_{\underline{W}}(-0.3) \times \mu_{\underline{S}}(0.6), \mu_{\overline{W}}(-0.3) \times \mu_{\overline{S}}(0.6)\right] = [0.8 \times 0.7, 1.0 \times 0.95] = [0.56, 0.95]$
$\rightarrow \left[\underline{y}^2, \overline{y}^2\right] = [-0.2, 0.2].$

(3) $\left[\underline{f}^3, \overline{f}^3\right] = \left[\mu_{\underline{E}}(-0.3) \times \mu_{\underline{N}}(0.6), \mu_{\overline{E}}(-0.3) \times \mu_{\overline{N}}(0.6)\right] = [0.0 \times 0.04, 0.08 \times 0.1] = [0.0, 0.008]$
$\rightarrow \left[\underline{y}^3, \overline{y}^3\right] = [0.2, 0.6].$

(4) $\left[\underline{f}^4, \overline{f}^4\right] = \left[\mu_{\underline{E}}(-0.3) \times \mu_{\underline{S}}(0.6), \mu_{\overline{E}}(-0.3) \times \mu_{\overline{S}}(0.6)\right] = [0.0 \times 0.7, 0.08 \times 0.95] = [0.0, 0.076]$
$\rightarrow \left[\underline{y}^4, \overline{y}^4\right] = [0.4, 0.8].$

Based on the KM algorithms, we find that $L = 1$ and $R = 2$ for this example. Thus,

$$y_l = \frac{\overline{f}^1 \underline{y}^1 + \underline{f}^2 \underline{y}^2 + \underline{f}^3 \underline{y}^3 + \underline{f}^4 \underline{y}^4}{\overline{f}^1 + \underline{f}^2 + \underline{f}^3 + \underline{f}^4} = \frac{0.1 \times (-0.6) + 0.56 \times (-0.2) + 0 \times 0.2 + 0 \times 0.4}{0.1 + 0.56} = -0.2606$$

$$y_r = \frac{\underline{f}^1 \overline{y}^1 + \underline{f}^2 \overline{y}^2 + \overline{f}^3 \overline{y}^3 + \overline{f}^4 \overline{y}^4}{\underline{f}^1 + \underline{f}^2 + \overline{f}^3 + \overline{f}^4} = \frac{0.032 \times (-0.3) + 0.56 \times 0.2 + 0.008 \times 0.6 + 0.076 \times 0.8}{0.032 + 0.56 + 0.008 + 0.076} = 0.2485$$

Therefore, the final crisp output of this IT2 FLS y is

$$y = \frac{y_l + y_r}{2} = \frac{-0.2606 + 0.2485}{2} = -0.0061$$

Next let us discuss how to design and implement a real IT2 FLC for a DC motor system.

9.6 Design and Build an IT2 Fuzzy Logic Controller for a DC Motor System

Recalled in Sect. 8.4 in Chap. 8, we discussed and designed a type-1 FLC for a DC motor system with transfer function

$$G(s) = \frac{6520}{s(s+430.6)} e^{-0.005\,s} \tag{9.25}$$

In fact, this DC motor system is implemented in a Laser Tracking System (LTS), and it works as a small motor assembled with a big DC motor to build a tracking gimbal system.

In this section, we will discuss how to design and implement an interval type-2 FLC for this DC motor system and compare the control performances for both types of FLCs with a MATLAB Simulation process.

First let us build an IT2 FLC for this DC motor system with an IT2 Fuzzy Logic Toolbox provided by Dr. Oscar Castillo and his group [1, 2].

9.6.1 Build an IT2 Fuzzy Logic Control System for a DC Motor System

Because of the software compatibility issue, this IT2 Fuzzy Logic Toolbox can only work on previous MATLAB® versions, such as MATLAB® R2010b. Therefore, we will use this version of the software to build our IT2 FLC system.

Open the IT2 Fuzzy Inference System by typing **it2fuzzy** in the MATLAB Command window after the prompt sign **>>**. Perform the following operational steps to complete this IT2 Fuzzy Logic Inference System building.

9.6.1.1 Create a New IT2 Fuzzy Logic Inference System IT2FL_LTS

When the IT2 Fuzzy Inference System is opened, a default IT2 Fuzzy Inference System Editor is displayed, as shown in Fig. 9.6.

Go to **Edit→Add Variable→Input** menu item to add one more input since we need both inputs to this DC motor control system, error **ex** and error_rate **dex**.

Now go to **File→Export→To Disk** to save this FIS as **IT2FL_LTS.fis** in your appropriate folder.

Modify the **input1** variable' name by clicking on **input1** box and change its name to **ex** on the Name box. Perform similar operations to change the **input2** and the **output1** variable's name to **dex** and **yit2**, respectively.

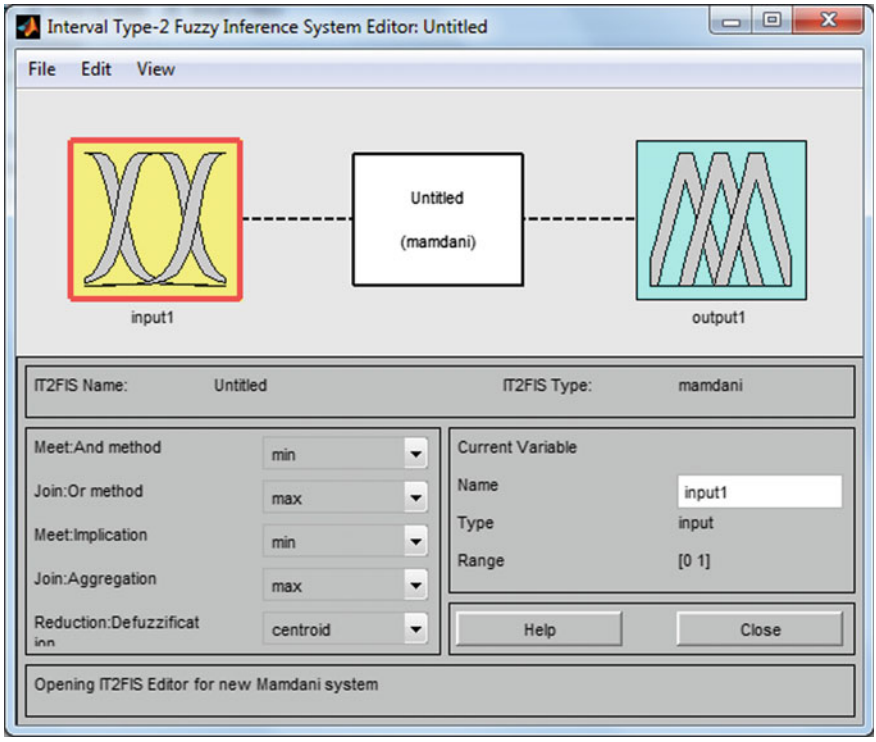


Fig. 9.6 The opened IT2 Fuzzy Inference System Editor

9.6.1.2 Build the Input and the Output Membership Functions

Perform the following operational steps to build both inputs and the output membership functions:

(1) Double click on the first input **ex** box to open its IT2 Membership Function Editor. Then go to **Edit→Remove All MFs** to delete all default MFs. Then go to **Edit→Add MFs** item to open the IT2 Membership Function dialog to add seven (7) MFs since totally we need seven (7) elements for this MF, **LN**, **MN**, **SN**, **ZE**, **SP**, **MP**, and **LP**, respectively.

(2) On the opened dialog, click on the drop-down arrow on the **Number of MFs** box, and select **7** from the list. Keep the default **MF type**, **igausstype2** as default with no changes, and click on the **OK** button.

(3) Click on **mf1** and change its name to **LN** from the **Name** box. Then perform similar operations to change the rest six (6) elements to: **mf2** → **MN**, **mf3** → **SN**, **mf4** → **ZE**, **mf5** → **SP**, **mf6** → **MP**, and **mf7** → **LP**, respectively.

(4) Go to the **Range** box located at the lower left pane, and change the range to **−0.2** and **0.2**.

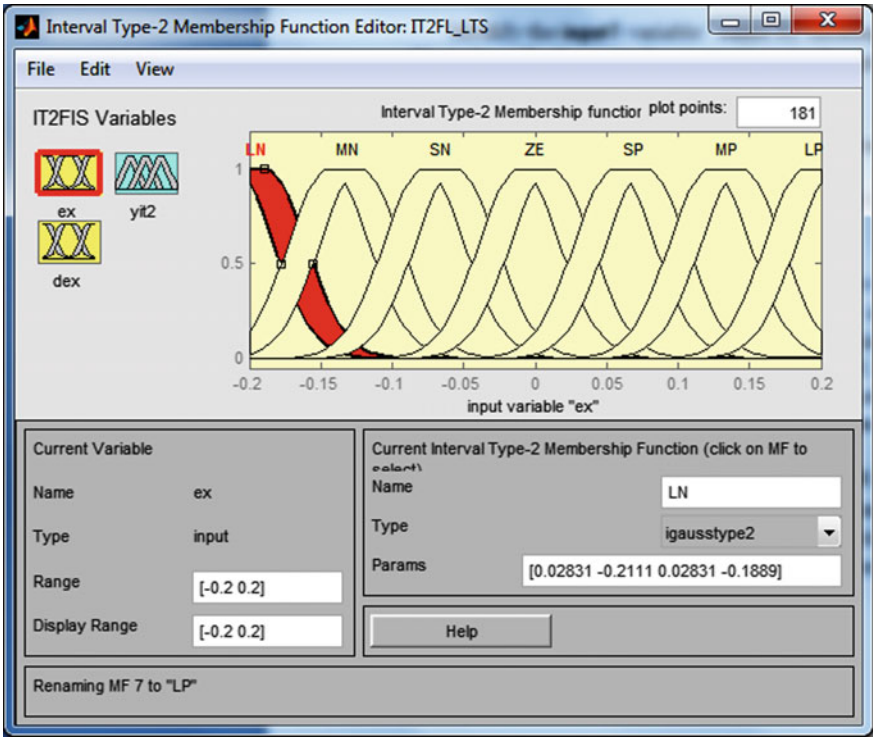


Fig. 9.7 The finished MF for the first input ex

Your finished MF for the input **ex** should match the one that is shown in Fig. 9.7.

Perform the similar operational steps shown above to create and modify the related MF for the second input dex. Basically, both MFs are identical and the only difference is that the **Range** for the **dex** is from −**0.15** to **0.15**.

Now, perform similar operations to the output MF **yit2** with seven (7) elements on this MF. The **Range** for this MF is from −**600** to **600**.

9.6.1.3 Design and Build Control Rules

Since we have two inputs, each has seven (7) elements, thus totally we have 49 control rules to be designed. Refer to Sect. 7.8.1.2 in Chap. 7 to get more details about these 49 rules. In this section, we only take care of implementing these rules in this FIS.

Go to **Edit→Interval Type-2 Rules** menu item to open the **Interval Type-2 Rule Editor**, and add the following 49 rules into this editor:

(1) If (ex is LN) and (dex is LN), then (yit2 is LN)
(2) If (ex is LN) and (dex is MN), then (yit2 is LN)
(3) If (ex is LN) and (dex is SN), then (yit2 is LN)
(4) If (ex is LN) and (dex is ZE), then (yit2 is LN)
(5) If (ex is LN) and (dex is SP), then (yit2 is MN)
(6) If (ex is LN) and (dex is MP), then (yit2 is SN)
(7) If (ex is LN) and (dex is LP), then (yit2 is ZE)
(8) If (ex is MN) and (dex is LN), then (yit2 is LN)
(9) If (ex is MN) and (dex is MN), then (yit2 is LN)
(10) If (ex is MN) and (dex is SN), then (yit2 is LN)
(11) If (ex is MN) and (dex is ZE), then (yit2 is MN)
(12) If (ex is MN) and (dex is SP), then (yit2 is SN)
(13) If (ex is MN) and (dex is MP), then (yit2 is ZE)
(14) If (ex is MN) and (dex is LP), then (yit2 is SP)
(15) If (ex is SN) and (dex is LN), then (yit2 is MN)
(16) If (ex is SN) and (dex is MN), then (yit2 is MN)
(17) If (ex is SN) and (dex is SN), then (yit2 is SN)
(18) If (ex is SN) and (dex is ZE), then (yit2 is SN)
(19) If (ex is SN) and (dex is SP), then (yit2 is ZE)
(20) If (ex is SN) and (dex is MP), then (yit2 is SP)
(21) If (ex is SN) and (dex is LP), then (yit2 is SP)
(22) If (ex is ZE) and (dex is LN), then (yit2 is MN)
(23) If (ex is ZE) and (dex is MN), then (yit2 is SN)
(24) If (ex is ZE) and (dex is SN), then (yit2 is SN)
(25) If (ex is ZE) and (dex is ZE), then (yit2 is ZE)
(26) If (ex is ZE) and (dex is SP), then (yit2 is SP)
(27) If (ex is ZE) and (dex is MP), then (yit2 is SP)
(28) If (ex is ZE) and (dex is LP), then (yit2 is MP)
(29) If (ex is SP) and (dex is LN), then (yit2 is SN)
(30) If (ex is SP) and (dex is MN), then (yit2 is SN)
(31) If (ex is SP) and (dex is SN), then (yit2 is ZE)
(32) If (ex is SP) and (dex is ZE), then (yit2 is SP)
(33) If (ex is SP) and (dex is SP), then (yit2 is SP)
(34) If (ex is SP) and (dex is MP), then (yit2 is MP)
(35) If (ex is SP) and (dex is LP), then (yit2 is MP)
(36) If (ex is MP) and (dex is LN), then (yit2 is SN)
(37) If (ex is MP) and (dex is MN), then (yit2 is ZE)
(38) If (ex is MP) and (dex is SN), then (yit2 is SP)
(39) If (ex is MP) and (dex is ZE), then (yit2 is MP)
(40) If (ex is MP) and (dex is SP), then (yit2 is LP)
(41) If (ex is MP) and (dex is MP), then (yit2 is LP)
(42) If (ex is MP) and (dex is LP), then (yit2 is LP)
(43) If (ex is LP) and (dex is LN), then (yit2 is ZE)

(44) If (ex is LP) and (dex is MN), then (yit2 is SP)
(45) If (ex is LP) and (dex is SN), then (yit2 is MP)
(46) If (ex is LP) and (dex is ZE), then (yit2 is LP)
(47) If (ex is LP) and (dex is SP), then (yit2 is LP)
(48) If (ex is LP) and (dex is MP), then (yit2 is LP)
(49) If (ex is LP) and (dex is LP), then (yit2 is LP)

Your finished control rules list should like the one that is shown in Fig. 9.8.

9.6.1.4 View the Complete IT2 Fuzzy Inference System

Now, we can have a detailed view about this IT2 fuzzy inference system by going to the menu item **View→Interval Type-2 Rules**. The completed view for this IT2 FIS is shown in Fig. 9.9. If one moves one of input vertical bars to obtain the different input values, the output **yit2** can be calculated immediately in real time and displayed on the top of the output column.

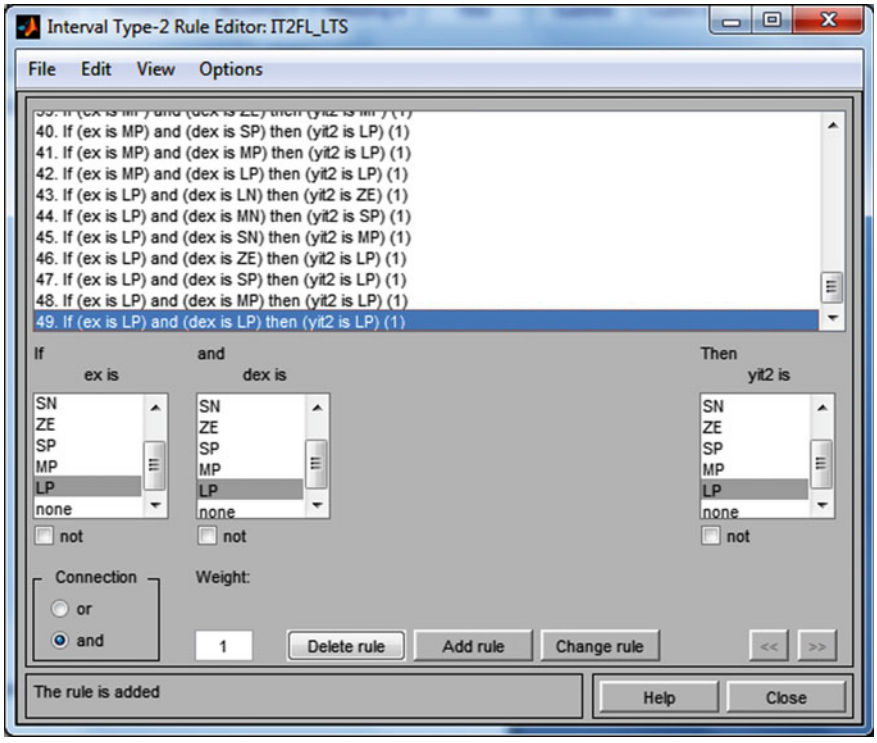


Fig. 9.8 The finished 49 control rules

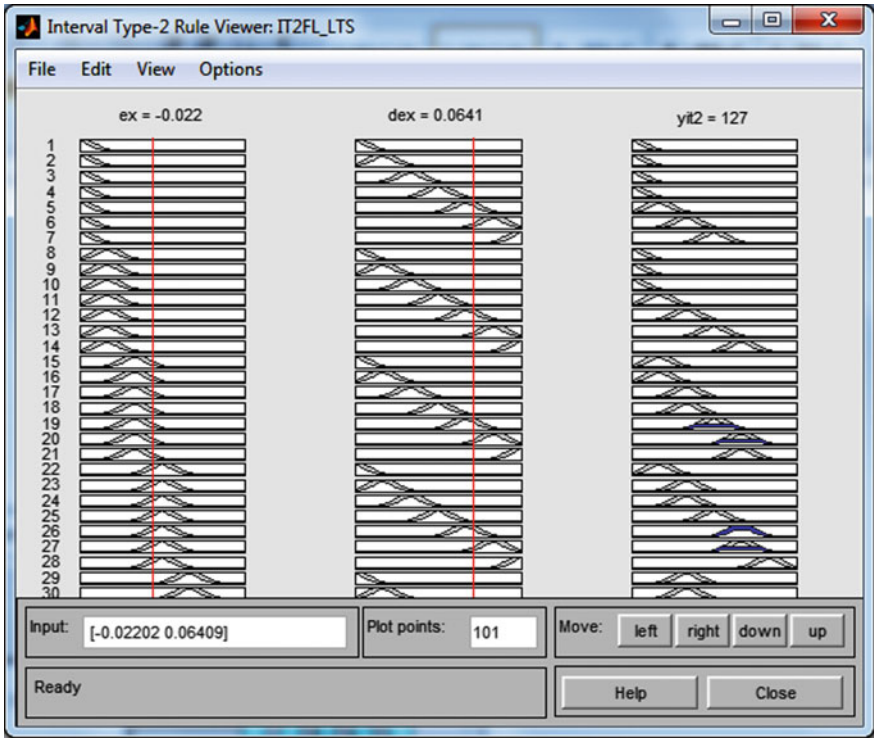


Fig. 9.9 The completed view for the IT2 FIS—IT2FL_LTS.fis

An example is shown in Fig. 9.9. Both inputs are **ex = −0.022**, **dex = 0.0641**, and the output **yit2 = 127**. One can also use the IT2 FIS command **gensurftype2()** to plot the envelope of this IT2 FIS with the following scripts:

```
a = readfistype2('IT2FL_LTS');
gensurftype2(a)
```

The plotted surface or envelope of this FIS is shown in Fig. 9.10.

9.7 Simulation Study for This IT2 Fuzzy Logic Control System

In this section, we try to use MATLAB® Simulink® to perform a simulation study for this IT2 FLC to confirm the control performances of this closed-loop DC motor control system. A comparison study between a type-1 and this IT2 FLC will be given in the next section.

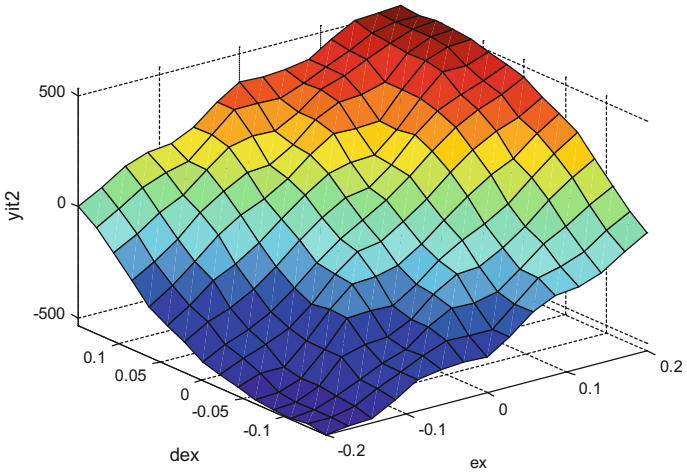


Fig. 9.10 The plotted surface or envelope of the FIS—IT2FL_LTS.fis

Before starting this simulation study, we need to export our IT2 FIS, **IT2FL_LTS.fis**, to the MATLAB® Workspace to make our study easier. Of course, export it to a file is also working for this simulation study.

Open our **IT2FL_LTS.fis** if it is not opened, go to **File→Export→To Workspace** item, and keep the original name unchanged, click on the **OK** button to complete this exporting operation.

9.7.1 Create a Simulation Block Diagram Model for FIS—IT2FL_LTS

First let us create a Simulink® model named **IT2FL_Model**.

Open MATLAB® Simulink® by typing **simulink** in the MATLAB® R2010b Command window after the prompt symbol **>>** to open the **Simulink Library Browser**. Then build this Simulink model by adding different blocks into this model container, and your finished model block diagram should match one that is shown in Fig. 9.11. Save this model as a model file **IT2FL_Model.mdl**.

When building this model, the following points should be paid special attention to:

(1) The **MATLAB Function** block is located at the **User-Defined Functions** group in the Simulink Library Browser. We need this function block to provide an IT2 FIS function to evaluate the IT2 fuzzy outputs based on two inputs.
(2) The Signal Generator (**Generator**) is used to work as a noise generator to create a simulated noise to test the noise reduction ability for this FLC later.

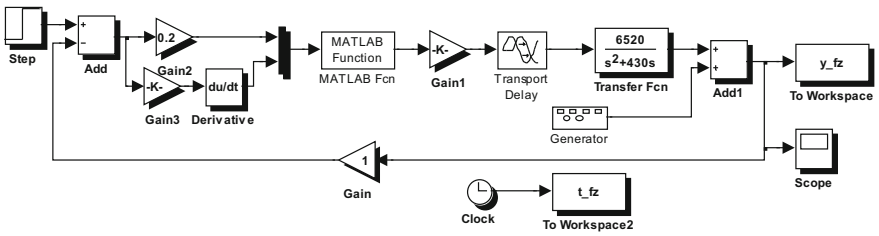


Fig. 9.11 The finished block diagram for the Simulink model

Right now one needs to set the **Amplitude** of this Generator to 0 since we will not handle any noise in this part.

(3) Double click on the **MATLAB Function** block and enter a MATLAB function: **evalifistype2([u(1) u(2)], IT2FL_LTS)** into the MATLAB function box. The two inputs are **u(1)** and **u(2)**, and the third argument, **IT2FL_LTS**, which is our IT2 FIS that has been exported into the MATLAB Workspace at the end of the last section.

(4) Two variable sets, **y_fz** and **t_fz**, are exported to the Workspace during the simulation process, and they will be used later to plot the step response of this IT2 FLC.

(5) Three Voltage Gains, **Gain1**, **Gain2**, and **Gain3**, will be used to adjust the amplifying coefficients when tuning this IT2 FLC to get the optimal control performance with a step response during the simulation study process.

Before the simulation process can be started, one needs to set up an appropriate simulation environment to perform this study. Go to **Simulation→Configuration Parameters** menu item to open the Configuration Parameters wizard. Perform the following operations to set up the environment:

(1) Enter 0.3 into the **Stop time** box under the **Simulation time** group at the top of this wizard since we need a short period of time for this simulation study.

(2) Make sure to select the **Variable-step** from the **Type** box under the **Solver options** group, and keep all other default setups with no changes. Click the **OK** button to complete this setup.

Now one can start the simulation process by going to **Simulation→Start** item.

During the simulation process, one needs to tune three gains, **Gain1**, **Gain2**, and **Gain3**. The optimal three gains are

- Gain1: **0.01**
- Gain2: **0.20**
- Gain3: **0.00005**

Next, let us take a look at our simulation study results.

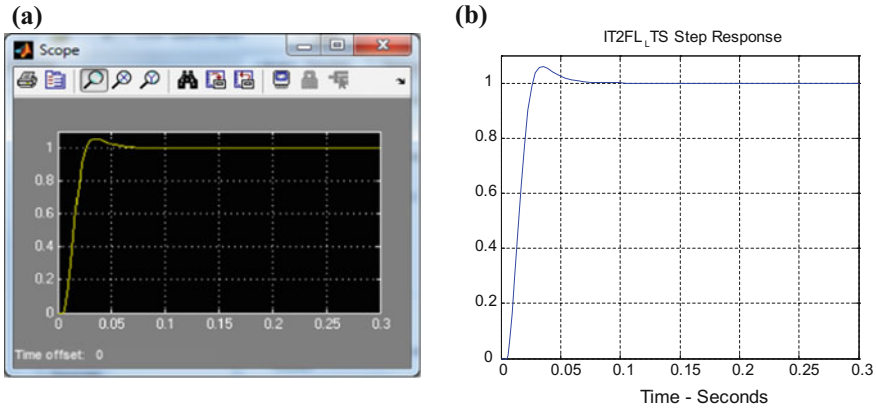


Fig. 9.12 Simulated step response for the IT2FL_LTS control system

9.7.2 Simulation Results for IT2FL_LTS Model

Based on the optimal parameters for three control gains, Gain1 ~ Gain3, the corresponding step response is shown in Fig. 9.12.

The step response shown in Fig. 9.12a is the output of the scope built and connected in simulation block diagram shown in Fig. 9.11. While the step response displayed in Fig. 9.12b is plotted with a separate Script file based on the output variables **y_fz** and **t_fz** that were output to the Workspace. Both variables are 66 × 1 arrays and the plotting script codes are shown in Fig. 9.13.

Next, let us concentrate on a comparison study between a PID controller and this IT2 FLC for this DC motor closed-loop control system.

```
% Plot IT2FL_LTS Step Response
% Y. Bai
% 2-25-2018

t = t_fz.signals.values;
y = y_fz.signals.values;
plot(t, y, 'b-');
grid;
title('IT2FL_LTS Step Response');
xlabel('Time - Seconds');
axis([0 0.3 0 1.1]);
```

Fig. 9.13 Script codes to plot step response for IT2FL_LTS system

9.8 Simulation Study for Comparison Among PID, Type-1 and IT2 FLC

Recalled in Lab Project 5.2 in Chap. 5 and Sect. 7.10 in Chap. 7, we asked students to design and simulate a PID controller and an FLC for the same DC motor system as we did in the last section to design an IT2 FLC. From that Lab Project and section, the optimal PID control parameters, the control gains for the type-1 FLC and IT2 FLC are

- For PID Controller: **P** = 10, **I** = 317, and **D** = 0.02.
- For Type-1 FLC: **Gain1** = 0.05, **Gain2** = 0.037, and **Gain3** = 0.00007.
- For IT2 FLC: **Gain1** = 0.01, **Gain2** = 0.20, and **Gain3** = 0.00005.

Now in this section, we need to compare the control performances for these three controllers, a PID controller, a type-1 FLC, and an IT2 FLC, with a simulation study. The comparison criteria include the control performances for a system with varying time delays, for a system with uncertain or random noises and for a system with uncertain dynamic models.

9.8.1 Simulation Study Comparison for Systems with Normal Time Delays

First, lets build our simulation block with Simulink® Library Browser. Open this Browser by typing **simulink** in the MATLAB® Command window after the prompt symbol **>>**.

Then create a new simulation model and build the simulation model by adding related blocks into this model platform. Your finished simulation model should match one that is shown in Fig. 9.14. Save this model as **T1IT2PIDComp.mdl** to an appropriate folder in your computer.

After finished this simulation block building and before the simulation process can be started, the following operations are needed to be performed:

(1) The type-1 fuzzy inference system **T1_FLC.fis** should have been built (refer to Sect. 7.8.1 in Chap. 7 to build it if it has not been built). The FIS named **T1_FLC.fis** is similar to the **FIS FZMotor.fis**.
(2) The type-1 FIS **T1_FLC.fis** should have been exported to the Workspace. If not, please open this FIS and export it to the Workspace.
(3) Double click on the **Fuzzy Logic Controller** block on Fig. 9.14, and enter **T1_FLC** into the **FIS file** box.

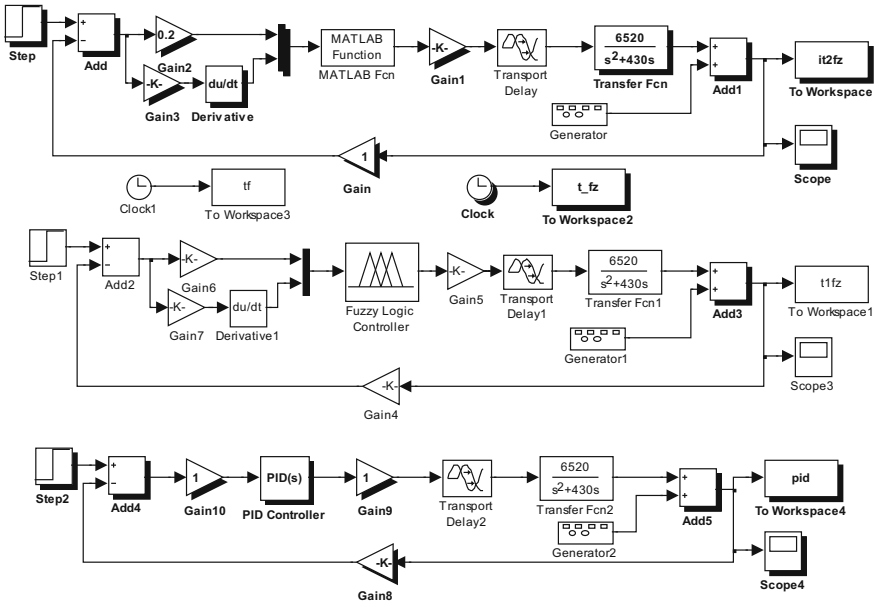


Fig. 9.14 The completed simulation block diagram model **T1IT2PIDComp.mdl**

(4) Perform a similar operation to the block **MATLAB Function** on the top in Fig. 9.14 to enter **evalifistype2([u(1) u(2)], IT2FL_LTS)** into the MATLAB function box.
(5) Double click on the PID Controller box and enter three optimal **PID control** gains shown above into three boxes: **P** = 10, **I** = 317, and **D** = 0.02.
(6) Double click on three Step input blocks, **Step**, **Step1**, and **Step2**, and enter **0.001** into the **Sample time** box to define a fixed sampling period.
(7) Go to **Simulation**→**Configuration Parameters** menu item to open the **Configuration Parameters** dialog. Select the **Fixed-step** from the Type combo box under **Solver options** group.
(8) Set three Transport Delay blocks, **Transport Delay**, **Transport Delay1**, and **Transport Delay2**, as 0.005 (5 ms) time delay.
(9) Set **Time range** property as 0.3 s for three Scopes, **Scope**, **Scope3**, and **Scope4**.
(10) Right click on each Scope to open its **Scope properties** dialog and set the **Y-main** as 0, and **Y-max** as 1.2 for all three Scope blocks.

Now one can start this simulation study by going to **Simulation**→**Start** menu item. During the simulation process, all three Scopes should be opened to watch the step responses and all control parameters, including the PID control gains, **P**, **I**, and **D**, as well as six voltage gains, **Gain1**–**Gain8**, are needed to be tuned to get the optimal step responses for all three kinds of controllers. Based on the simulation study, the optimal values for these parameters are

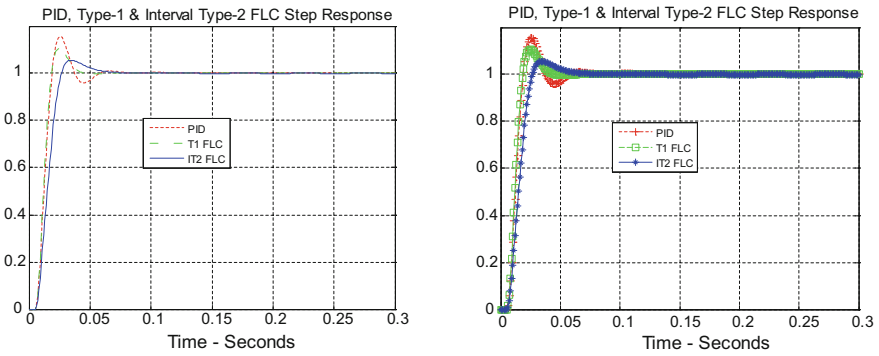


Fig. 9.15 Simulated step responses for three controllers

- For PID Controller: **P** = 6, **I** = 8, and **D** = 0.01.
- For Type-1 FLC: **Gain4** = 0.885, **Gain5** = 0.05, and **Gain6** = 0.07, **Gain7** = 0.00008.
- For IT2 FLC: **Gain1** = 0.01, **Gain2** = 0.20, and **Gain3** = 0.00005.

The simulated step responses are shown in Fig. 9.15.

It can be found from the simulation result that the setting times for three kinds of controllers are basically identical without a big difference. However, the overshoots for type-1 and PID controllers are relatively larger compared with that of IT2 FLC.

9.8.2 Simulation Study Comparison for Systems with Long Time Delays

Now, let us perform some simulation studies for longer time delay for three kinds of control strategies. Double click on three **Transport Delay** blocks, **Transport Delay**, **Transport Delay1**, and **Transport Delay2**, and change the delay time to 0.01 (10 ms), 0.1 (100 ms), 0.5 (500 ms), and 0.8 (800 ms).

For each time delay setup, all control parameters, including PID and all Gains, are needed to be retuned to perform an individual simulation study to obtain the optimal step responses for three control strategies. The simulated step responses for three controllers with 10 ms time delay are shown in Fig. 9.16.

For 10 ms time delay, the retuned optimal control parameters are

- For PID Controller: **P** = 3.6, **I** = 2.0, and **D** = 0.01, **Gain8** = 1.02.
- For Type-1 FLC: **Gain4** = 0.885, **Gain5** = 0.03, and **Gain6** = 0.07, **Gain7** = 0.00008.
- For IT2 FLC: **Gain1** = 0.006, **Gain2** = 0.20, and **Gain3** = 0.00005.

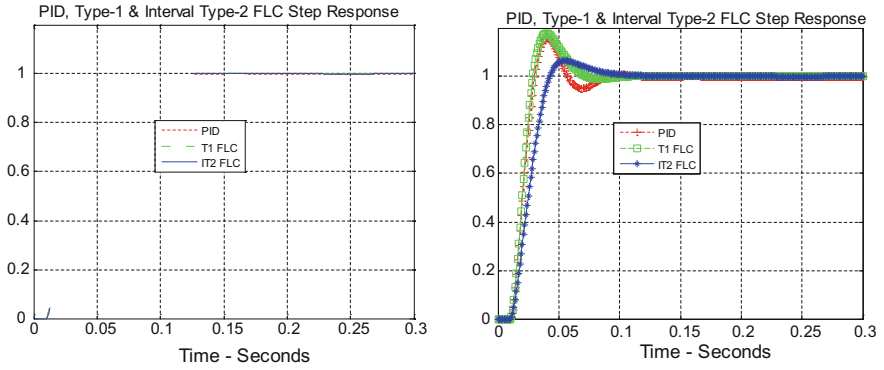


Fig. 9.16 Simulated step responses for three controllers with 10 ms delay

For 100 ms time delay, the retuned optimal control parameters are

- For PID Controller: **P** = 0.5, **I** = 0.08, and **D** = 0.01, **Gain8** = 1.01.
- For Type-1 FLC: **Gain4** = 0.885, **Gain5** = 0.0034, and **Gain6** = 0.07, **Gain7** = 0.00008.
- For IT2 FLC: **Gain1** = 0.0008, **Gain2** = 0.20, and **Gain3** = 0.00005.

The simulated step responses for three controllers with 100 ms time delay are shown in Fig. 9.17. Do not forget to change the value in the **Stop time** box, which is in the **Simulation time** group under the **Configuration Parameters** dialog, to 1 to increase the simulation time period.

Now let us take care of 500 ms time delay. Double click on three **Transport Delay** blocks and enter 0.5 into the **Time delay** box. Change the value in the **Stop time** box, which is in the **Simulation time** group under the **Configuration Parameters** dialog, to 5 to increase the simulation time period. Also change the value in the **Time range** box, which is in the **Parameters** menu item on three Scopes, to 5. For 500 ms time delay, the retuned optimal control parameters are

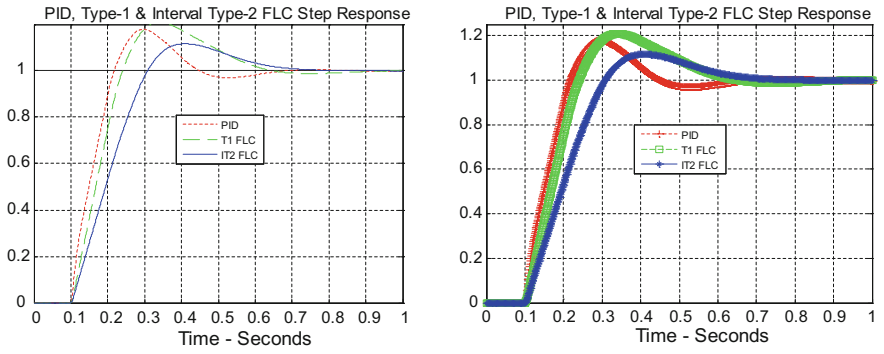


Fig. 9.17 Simulated step responses for three controllers with 100 ms delay

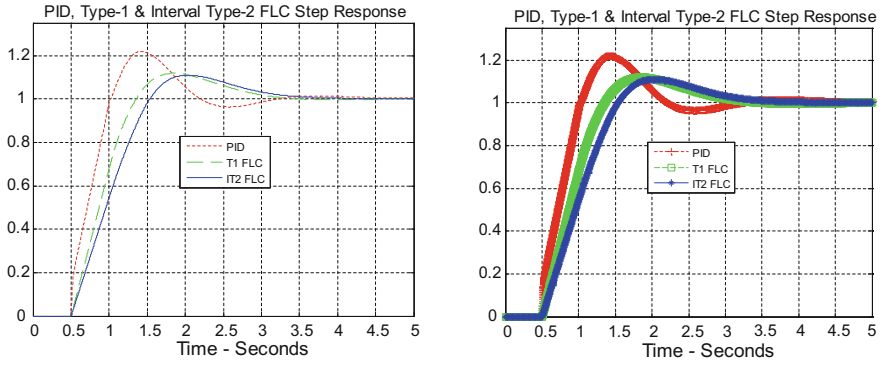


Fig. 9.18 Simulated step responses for three controllers with 500 ms delay

- For PID Controller: **P** = 0.11, **I** = 0.001, and **D** = 0.01, **Gain8** = 1.
- For Type-1 FLC: **Gain4** = 0.885, **Gain5** = 0.0006, and **Gain6** = 0.07, **Gain7** = 0.00008.
- For IT2 FLC: **Gain1** = 0.00016, **Gain2** = 0.20, and **Gain3** = 0.00005.

The simulated step responses for three controllers with 500 ms time delay are shown in Fig. 9.18. It can be found from Fig. 9.18 that the step response of the PID controller has larger overshoot and undershoot with longer settling time (>4 s). But both type-1 and IT2 FLC has shorter settling time (about 3.5 s) with smaller overshoots.

For 800 ms time delay, the retuned optimal control parameters are

- For PID Controller: **P** = 0.065, **I** = 0.0005, and **D** = 0.01, **Gain8** = 1.
- For Type-1 FLC: **Gain4** = 0.885, **Gain5** = 0.0004, and **Gain6** = 0.07, **Gain7** = 0.00008.
- For IT2 FLC: **Gain1** = 0.0001, **Gain2** = 0.20, and **Gain3** = 0.00005.

The simulated step responses for three controllers with 800 ms time delay are shown in Fig. 9.19.

9.8.3 Simulation Study Comparison for Systems with Random Noises

Now, let us take care of all three controllers for this DC motor control system with random noises that can be considered as disturbances to the system.

Perform the following operations to recover all control parameters and control gains to the original optimal values with Transport Delay as 5 ms:

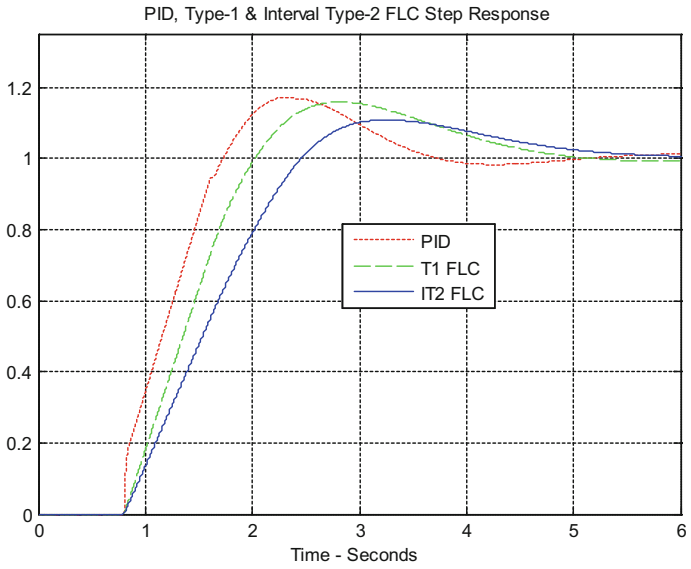


Fig. 9.19 Simulated step responses for three controllers with 800 ms delay

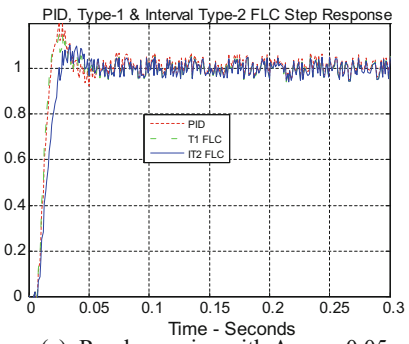
(1) Set three Transport Delay blocks, **Transport Delay**, **Transport Delay1**, and **Transport Delay2**, as 0.005 (5 ms) time delay.
(2) Open the **Parameters** dialog for each Scope block and set **Time range** property as 0.3 s for three Scopes, **Scope**, **Scope3**, and **Scope4**.
(3) Right click on each Scope to open its **Scope properties** dialog and set the **Y-main** as 0, and **Y-max** as 1.2 for all three Scope blocks.
(4) For PID Controller set **P** = 6, **I** = 8, and **D** = 0.01, **Gain8** = 1.01.
(5) For Type-1 FLC, set **Gain4** = 0.885, **Gain5** = 0.05, and **Gain6** = 0.07, **Gain7** = 0.00008.
(6) For IT2 FLC, set **Gain1** = 0.01, **Gain2** = 0.20, and **Gain3** = 0.00005.

Now, double click on each **Signal Generator** block to set up parameters to create random noise as below:

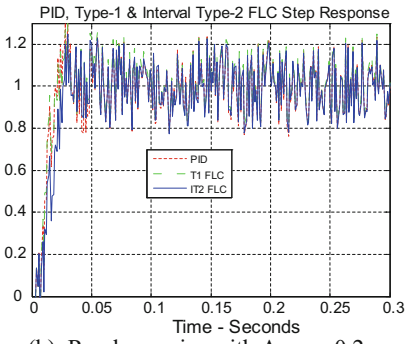
- Select random from the **Waveform** combo box.
- Enter 0.05 into the **Amplitude** box.

The simulated step responses for three kinds of controllers are shown in Fig. 9.20a, b. The step responses shown in Fig. 9.20b have an amplitude of random noise as 0.2.

It can be found from Fig. 9.20 that there is no significant difference among these three control strategies. But relatively speaking, both type-1 and IT2 FLC have a better noise reduction ability compared with that of PID controller.



(a) Random noise with Amp = 0.05.



(b) Random noise with Amp = 0.2.

Fig. 9.20 Simulated step responses for three controllers with random noises

9.8.4 *Simulation Study Comparison for Systems with Model Uncertainty*

Now, let us perform a simulation study for three controllers with some model uncertainty, which means that the dynamic model of the DC motor system has some variations with the time elapsing.

In Sect. 7.11.1.3 in Chap. 7, we made a similar simulation study for systems with uncertain models. In fact, the complete dynamic model for that DC motor system is

$$G(s) = 10.68\frac{s+610.5}{s(s+430.6)}\mathrm{e}^{-0.005\,s} \tag{9.26}$$

The difference between this model and one shown in Eq. (9.25) is that a zero = −610.5 is added into this model. The reason we utilized the model shown in (9.25) instead of model in (9.26) in the above simulation studies is because this complete model can be simplified from (9.26) to (9.25).

In order to introduce some uncertainty to the system, first the zero of this transfer function (z = −610.5) is modified from 5, 10 to 50%, and the pole is fixed. Then, the pole s = −430.6 will be modified from 10, 30 to 50% and the zero is kept unchanged.

A list of modified system models and related change-percentage-rate on zero and pole was already given in Table 7.8, but now we duplicate it here as our Table 9.2. The model shown in Eq. (9.26) is the original system model identified by using the MATLAB® Identification Toolbox for this DC motor system. The simplified model was used when the design of the conventional PID controllers in Chap. 5.

The simulated results are shown in Fig. 9.21 for the modified system models due to some uncertainties existed in the original DC motor system model.

Table 9.2 Modified system models and related change-rate on zero and pole

Zero/pole	Change-rate (%)	Modified system model
Zero	5	$10.68\frac{s+641.0}{s(s+430.6)}e^{-0.005\,s}$
Zero	10	$10.68\frac{s+671.5}{s(s+430.6)}e^{-0.005\,s}$
Zero	50	$10.68\frac{s+915.8}{s(s+430.6)}e^{-0.005\,s}$
Pole	10	$10.68\frac{s+610.5}{s(s+473.7)}e^{-0.005\,s}$
Pole	30	$10.68\frac{s+610.5}{s(s+301.4)}e^{-0.005\,s}$
Pole	50	$10.68\frac{s+610.5}{s(s+215.3)}e^{-0.005\,s}$

It can be found from Fig. 9.21 that for small zero uncertain models such as 5% or 10% shown in Fig. 9.21a, b, there are no significant differences among the three controllers. However, this difference becomes significant when the zero uncertainty is close to 50% shown in Fig. 9.21c. At that situation, the PID controller generated a large overshoot (≈30%) with a longer settling time (≈0.08 s) compared with type-1 FLC (22% overshoot and 0.04 s settling time) and IT2 FLC (18% overshoot and 0.04 s settling time).

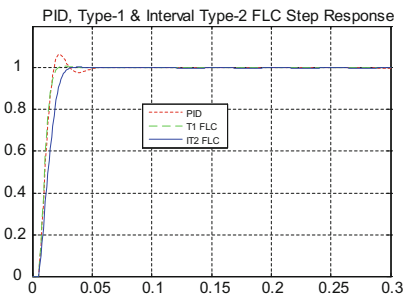
For uncertain poles, similar results are obtained. There is no significant difference among the three controllers as the pole uncertainty is small (10%) shown in Fig. 9.21d. However, as the pole uncertainty becomes larger and larger, more and more significant differences occur among the three control strategies. Especially when the pole uncertainty is 30%, the PID controller generated a 30% overshoot with a longer settling time (0.07 s) compared with the type-1 FLC (20% overshoot and 0.04 s settling time) and IT2 FLC (17% overshoot and 0.05 s settling time). These differences become much more significant as the pole uncertainty is 50%. At that situation, the PID controller produces a 60% overshoot with a much longer settling time (0.15 s) compared with the type-1 FLC (43% overshoot and 0.08 s settling time) and the IT2 FLC (38% overshoot and 0.08 s settling time).

Based on these simulation studies, we can conclude that relatively speaking, both type-1 FLC and IT2 FLC provide better controllability on long time delay, random noises, and uncertainty in dynamic models. Therefore, both type-1 and IT2 FLC are more robust controllers compared with PID controllers.

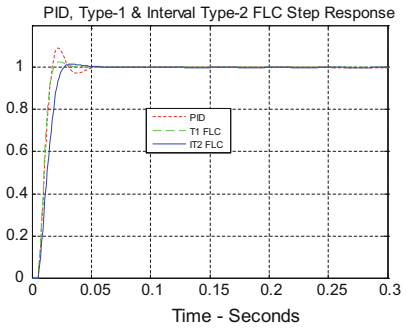
When performing above simulations, one needs to place correct numerator and denominator for three dynamic models of DC motor, **Transfer Fcn**, **Transfer Fcn1**, and **Transfer Fcn2**, in the simulation block diagram shown in Fig. 9.14 based on values shown in Table 9.2.

9.9 Summary

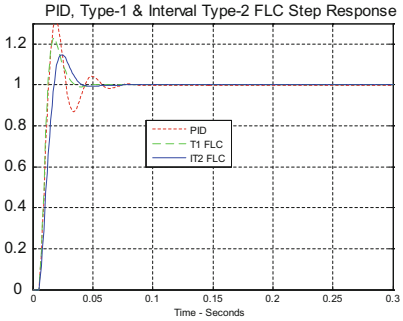
The main topic discussed in this chapter is an Interval Type-2 Fuzzy Logic Control (IT2 FLC) strategy applied in most popular control systems, including linear and nonlinear systems.



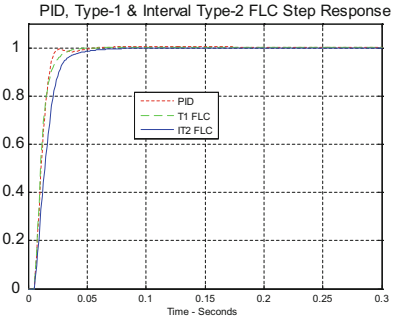
(a) 5% uncertainty on zero.



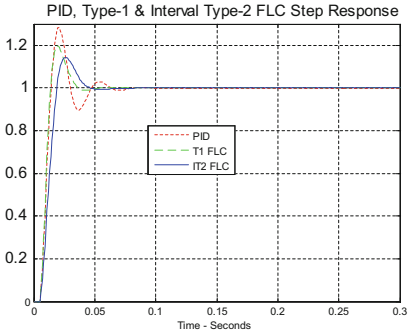
(b) 10% uncertainty on zero.



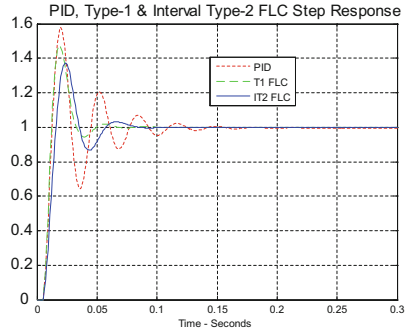
(c) 50% uncertainty on zero.



(d) 10% uncertainty on pole.



(e) 30% uncertainty on pole.



(f) 50% uncertainty on pole.

Fig. 9.21 Simulated step responses for uncertain system models

Starting at Sect. 9.1, an introduction about the differences between a type-1 fuzzy set and an interval type-2 fuzzy set is presented to provide readers a clear and complete picture about these two kinds of fuzzy sets and inference systems. Compared with type-1 fuzzy set, an interval type-2 fuzzy set provides more controllability and flexibility for more complicated or nonlinear systems applied in our real world. Following this introduction, more detailed and deeper discussions about

the interval type-2 fuzzy set (IT2 FS) operations and IT2 fuzzy inference system (FIS) are presented in Sects. 9.2 and 9.3, respectively.

Basically, it is similar between a type-1 FIS and an IT2 FIS in most aspects, and the only difference between them is that for membership functions and control rule outputs, they are not crisp values, but instead they are fuzzy sets. Therefore, a type reducer is needed to convert those fuzzy sets to the related crisp values with the help of Karnik–Mendel (KM) algorithms that is introduced in Sect. 9.4.

To provide a more crystal clear picture about how to implement this IT2 FIS in a real system, an example is given in Sect. 9.5 to illustrate how to use an IT2 fuzzy set and FIS to design a real IT2 control system. A more practical example is provided in Sect. 9.6 to direct readers to design and build an actual IT2 FLC for a DC motor control system step by step by using an IT2 Fuzzy Logic Toolbox.

A simulation study is discussed in Sect. 9.7 to help users to familiarize them with the MATLAB Simulink tool to perform simulation studies with an IT2 FLC built in Sect. 9.6. In Sect. 9.8, a comparison study is presented to compare three control strategies: PID controllers, type-1, and IT2 FLCs with a simulation study. Various control factors and situations are simulated to illustrate the advantages of type-1 and IT2 FLC over PID controllers, which including the time delay, long time delays, random noises, and model uncertainty.

Homework and Labs

Lab Project 9.1

Using IT2 Fuzzy Logic Toolbox to design and build an FLC for a DC motor system that is identified in Sect. 4.7.5 in Chap. 4. The identified dynamic model of that DC motor is

$$G(\mathrm{s}) = \frac{3.776}{(1 + 0.56\ \mathrm{s})} \mathrm{e}^{-0.09\ \mathrm{s}} \cong \frac{6.74}{(s + 1.79)} \mathrm{e}^{-0.09\ \mathrm{s}} \tag{9.27}$$

In Sect. 4.7.5, the input to the DC motor is the PWM values and the output is the motor encoder rotating speed (Pulse Per Revolution—**PPR**). The relationship between the motor encoder speed (PPR) and the control output PWM values is shown in Table 4.12 and repeated in Table 9.3. The ranges of inputs and output relationship are shown in Table 9.4.

Two inputs, the error value **err** and the error rate **err_rate**, work as inputs and the output is the motor encoder speed **m_sp**.

The range for the error, **err** is about −0.3 to 0.3, it can be divided into five MFs, **LN**, **SN**, **ZE**, **SP**, and **LP**. Similar to error, the **err_rate** can also be divided into five MFs. The range for the output **m_sp** is from −800 to 800 PPR, and it can be divided into five MFs, LN, SN, ZE, SP, and LP. The waveforms of all MFs should be **igausstype2** type.

Totally, one may need twenty-five (25) control rules for this FIS. Try to build these rules based on the common senses and general control principles.

The DC motor model shown in Eq. (9.27) is only used later for the simulation study and comparison between an IT2 FLC and a PID controller.

Table 9.3 The collected QEI encoder speed values and PWM values

Number	PWM values (u)	Encoder velocity values (es) PPR	
		Hexadecimal	Decimal
1	100	49	73
2	300	F4	244
3	500	15C	348
4	700	19F	415
5	900	1E0	480
6	1100	1EF	495
7	1300	1FA	506
8	1500	201	513
9	1700	206	518
10	1900	20C	524
12	2100	20D	525
13	2300	20C	524
14	2500	20E	526
15	2700	20E	526

Table 9.4 Ranges of the input–output variables

Variables	Variables ranges	Units
Input error (err)	−0.3 to 0.3	V
Input error rate (err_rate)	−0.08 to 0.08	V
Motor output speed (m_sp)	−800 to 800	PPR

Lab Project 9.2
Using MATLAB® R2010b and Simulink® to perform a simulation study for the IT2 FLC obtained from Lab Project 9.1 to obtain an optimal step response for a step input. Tune all control gains to obtain an optimal step response. The Transport Delay should be 90 ms (0.09 s).

Lab Project 9.3
Using MATLAB® R2010b and Simulink® to perform a simulation study to compare the control performances (step response) of this IT2 FLC built in Lab Project 9.1 and a PID controller designed in Sect. 5.3.6.2 in Chap. 5. The optimal PID control gains are $P = 2.76$, $I = 1.13$, and $D = 1.69$. Use these gains as the initial PID values for the PID controller in that simulation study.

Lab Project 9.4
Redo Lab Project 9.3 with different uncertain models. The original model is shown in Eq. (9.27). Change the model by modifying the location of the pole, $s = -1.79$, to 10, 30, 50, and 80% of the original location, and this is equivalent to model uncertainty. Keep all control parameters unchanged to see the control performances to compare control robustness between the PID controller and the IT2 FLC. Plot the step responses for these model uncertainties.

References

1. Castro, J. R., Castillo, O., Melin, P., Martínez, L. G., Escobar, S., & Camacho, I. (2007). *Building fuzzy inference systems with the interval Type-2 fuzzy logic toolbox*. IFSA.
2. Castro, J. R., Castillo, O., & Melin, P. (2007). *An interval Type-2 fuzzy logic toolbox for control applications*. FUZZ-IEEE.

Appendix A
Download and Install Keil MDK-ARM 5.24a IDE

1. Go to the URL: https://www.keil.com/download/product to begin this process. Click on the **MDK-Arm** button to begin this download process (Fig. A.1).
2. Enter your information to finish this registration process. Then click on the **Submit** button to go to the next screen (Fig. A.2).
3. Click on **MDK_524A.EXE** link to start this download process (Fig. A.2).
4. Click on the **Save** button to save this file to your C: drive. Then click on the **Run** and **Yes** buttons to download this software.
5. A confirmation page is shown in Fig. A.3. Click **Next** button to go to the next screen.
6. Check on **I agree to all the terms of the preceding License Agreement** checkbox, and click on the **Next** button.

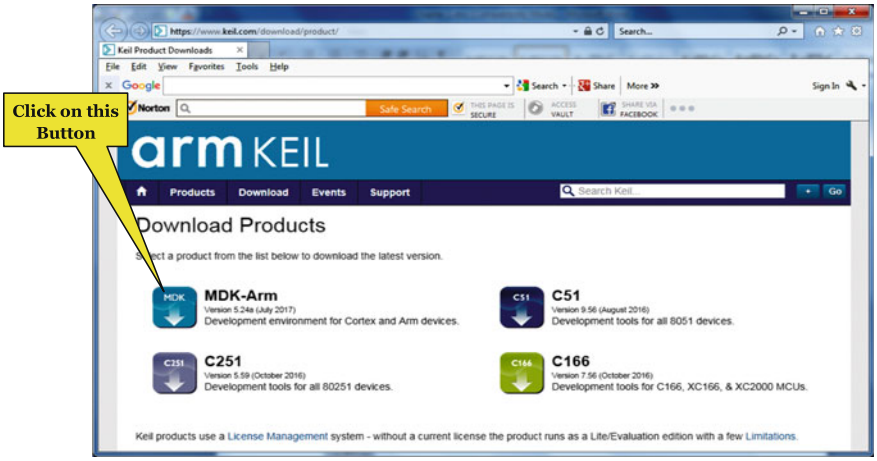


Fig. A.1 The download starting page

© Springer Nature Switzerland AG 2019

Y. Bai and Z. S. Roth, *Classical and Modern Controls with Microcontrollers*, Advances in Industrial Control, https://doi.org/10.1007/978-3-030-01382-0

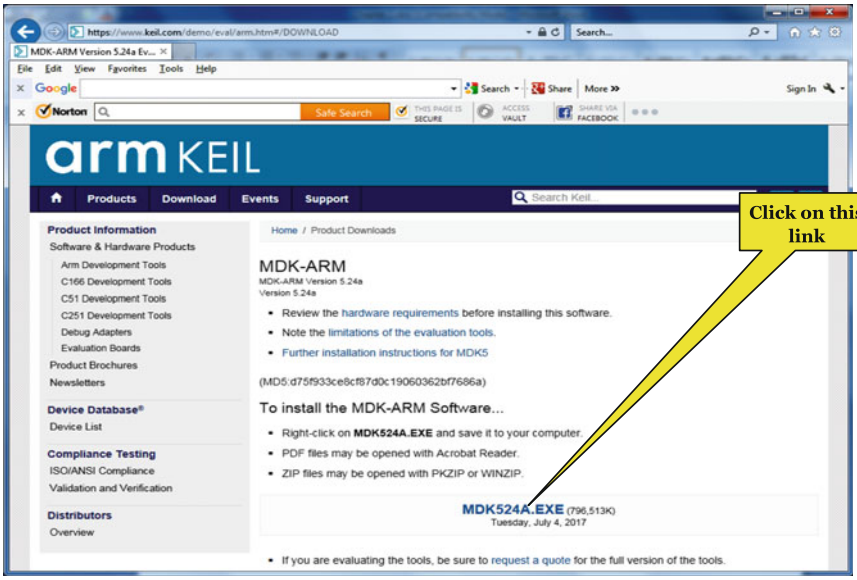


Fig. A.2 The information page

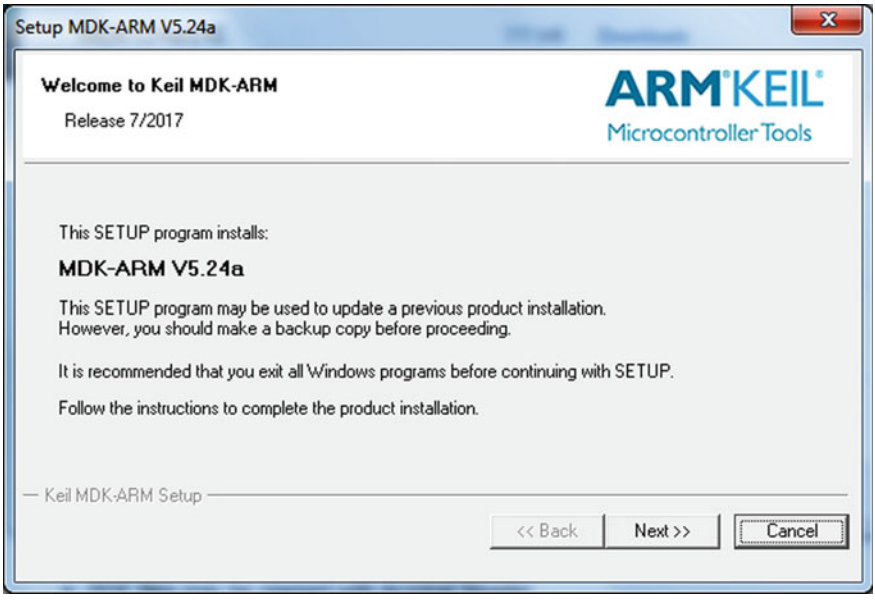


Fig. A.3 The confirmation page

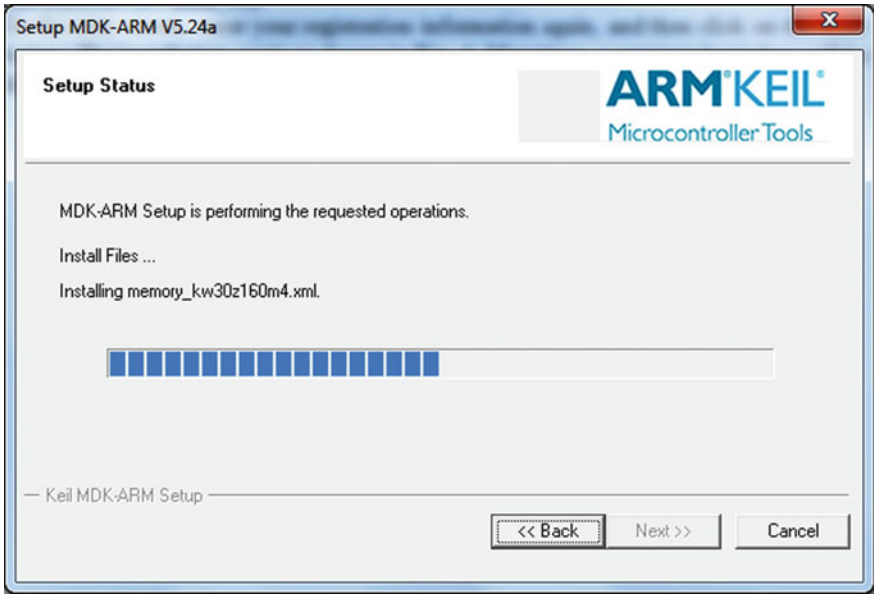


Fig. A.4 The installation process

7. Click on the **Next** button on the next screen if you want to keep the default location to install this software (**C:\Keil_v5**).
8. On the next screen, enter your registration information again, and then click on the **Next** button. The installation starts as shown in Fig. A.4. Meantime, a command window will be displayed to show that the debug driver, the **ULINK Driver**, will be installed (Fig. A.5).
9. A final screen shown in Fig. A.6 will be displayed when this installation is completed. Click on the **Finish** button to close this process.

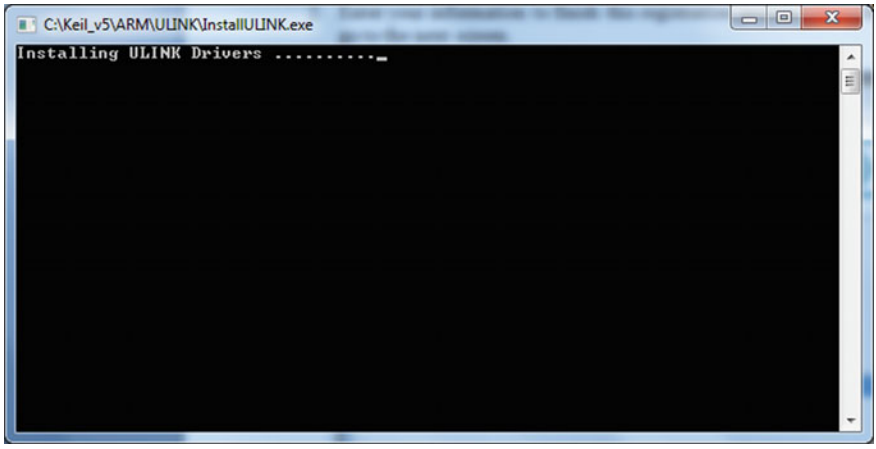


Fig. A.5 The installation process of the ULINK driver

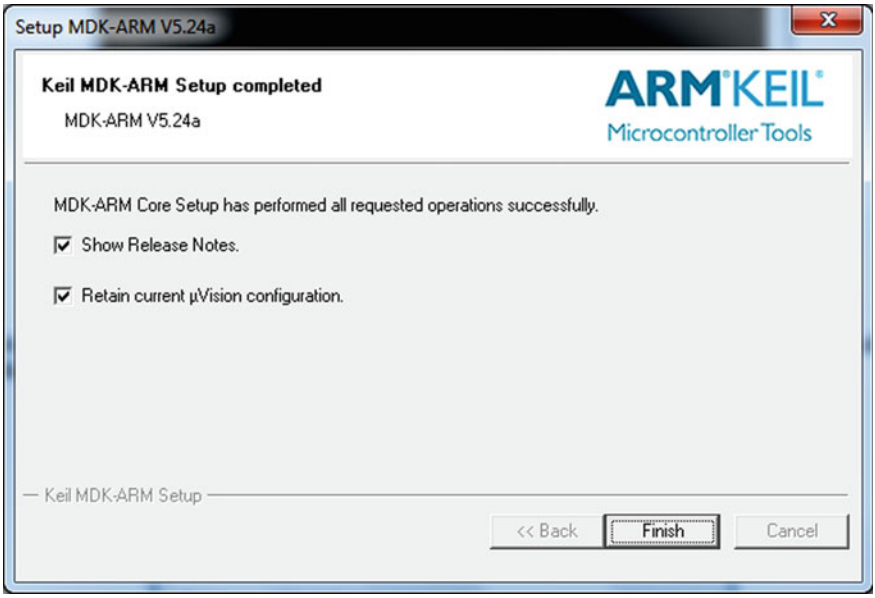


Fig. A.6 The finishing page

10. A release page is displayed to show the details about this updated version of MDK.
11. Also a dialog displaying to install the pack is appeared to help and assist readers to go through all steps for installing required packs and drivers used for your target microcontroller or MCU, as shown in Fig. A.7. Click on the **OK** button to close that message box and continue to go to installing all packs and driver process.
12. In the opened **Pack Installer** window shown in Fig. A.8, all available Developed Firmware Packs (DFPs) are displayed in the left pane under the **Devices** tab. These DFPs are developed by the Keil to assist various MCU vendors to build related developments based on different types of MCU via this IDE. One can select any desired pack to install that pack by clicking on the related **Install** button. The **Pack Installer** is a utility for installing, updating, and removing Software Packs, and can be launched from within µVision or standalone, outside of µVision.
13. Since we will use *Texas Instruments TM4C123G Launchpad—EK-TM4C123GXL Evaluation Kit* as our development tool, we need to install the ***TM4C123x Series Family Pack*** and related examples in this Pack Installer. Expand three items **Texas Instruments**, **Tiva C Series**, and **TM4C123x Series** and click on the **TM4C123GH6PM** under the **Devices** tab in the left pane, as shown in Fig. A.8. Then go to right pane and click on the **Install** button located on the right side of **Keil:TM4C_DFS** under the **Packs** tab, as shown in Fig. A.8.

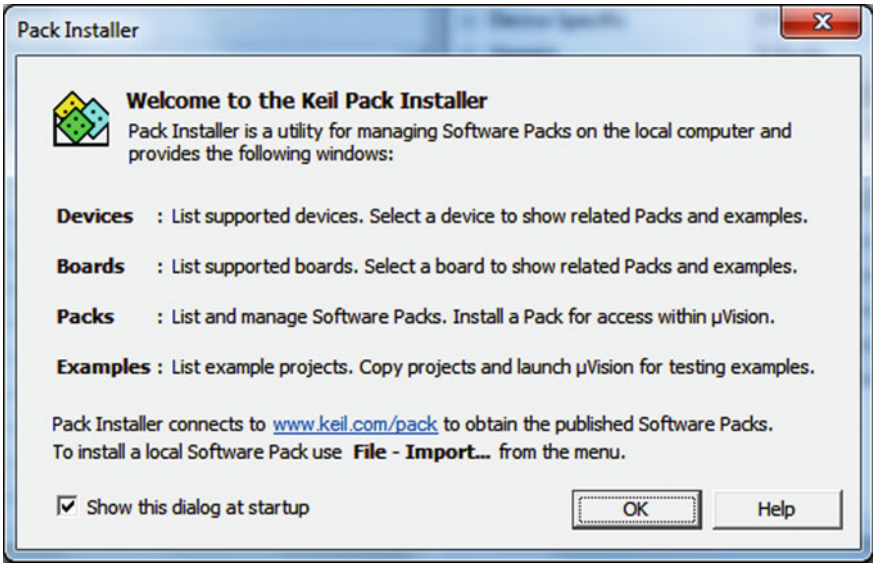


Fig. A.7 The Pack Installer starting page

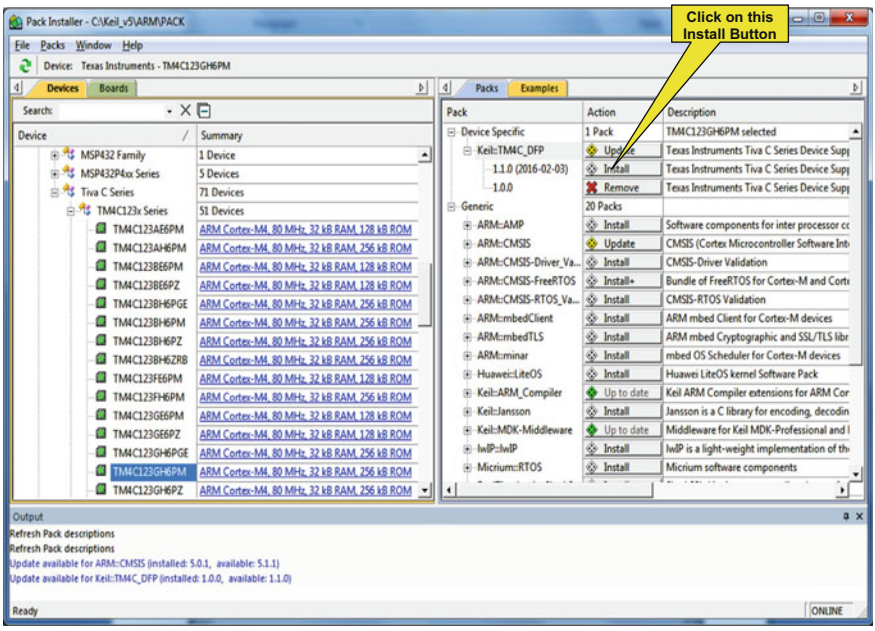


Fig. A.8 The opened Pack Installer window

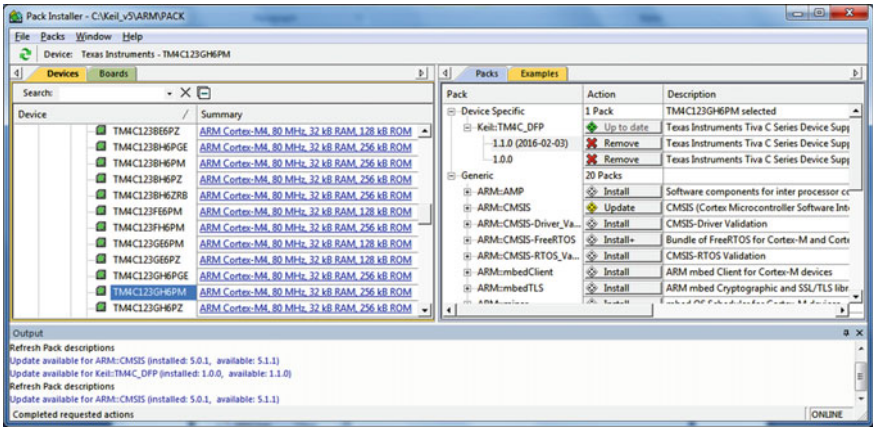


Fig. A.9 The installed pack window

14. The installation process is shown on a task bar located at the lower right corner. Wait for the Install process to be completed.
15. When the **TM4C_DFP** is installed, the **Install** button under the **Action** tab becomes **Up to date**, as shown in Fig. A.9. This indicates that the installed DFP for the selected MCU, **TM4C123GH6PM**, which is an ARM Cortex-M4F based 32-bit microcontroller, is up to date.
16. Three default CMSIS-related support tools, **ARM::CMSIS**, **Keil::ARM_Compiler**, and **Keil::MDK-Middleware**, are also indicated as **Up to date** as shown in Fig. A.10. We need these tools to build our application projects later.

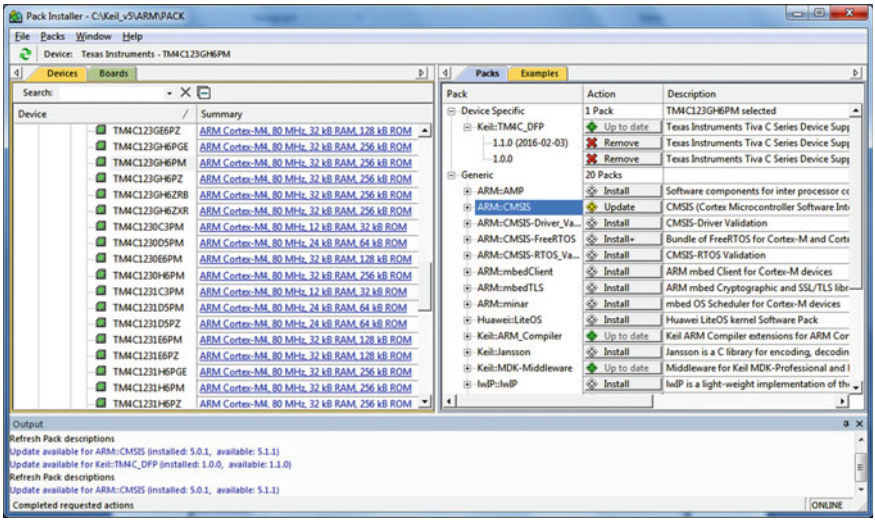


Fig. A.10 The finished installing pack window

Appendix B
Download and Install TivaWare SW-EK-TM4C123GXL Package

The SW-EK-TM4C123GXL package contains the TivaWare™ for C Series release for the Tiva™ C Series TM4C123G Launchpad (EK-TM4C123GXL). This package includes the latest version of the TivaWare for C Series Driver Library, USB Library, and Graphics Library. It also includes several complete examples for the Tiva C Series LaunchPad.

You need to download this package if you have already installed a supported third-party Integrated Development Environment (IDE), such as Keil MDK, on your system.

This EK-TM4C123GXL package contains the SW-EK-TM4C123GXL software as well as installation files for the selected IDE and the Microsoft® Windows® drivers for the debug interface. This package also contains several documents to help you get started with using the Tiva C Series LaunchPad. Each file is a single package with everything you need to program and debug the Tiva C Series LaunchPad using the respective toolset. Depending on your operating systems (Windows 7/8 or 10), this download and installation process is basically identical. Let us do this in the Windows 7 platform as an example.

© Springer Nature Switzerland AG 2019

Y. Bai and Z. S. Roth, *Classical and Modern Controls with Microcontrollers*,
Advances in Industrial Control, https://doi.org/10.1007/978-3-030-01382-0

B.1 Download and Install TivaWare SW-EK-TM4C123GXL Package for Windows 7

1. Go to the URL: http://www.ti.com/tool/SW-EK-TM4C123GXL to begin this process. Click on **Get Software** button on the right-hand side of **SW-EK-TM4C123GXL** to begin this download process (Fig. B.1).
2. A group or a list of software files related to TM4C123GXL is displayed in next page (Fig. B.2). Click the link **SW-EK-TM4C123GXL-2.1.4.178.exe** to begin this downloading process.
3. If you are a new user, please complete registration process by entering the necessary information and click on the **Register** and continue button to go to the next page. Otherwise, you can enter your username and password to login to this system.
4. In the next page, you need to enter related information to complete this US Government Approval process. The point you need to pay attention to is: the purpose of using this software, Military or Civic. Then check the **Civic** radio

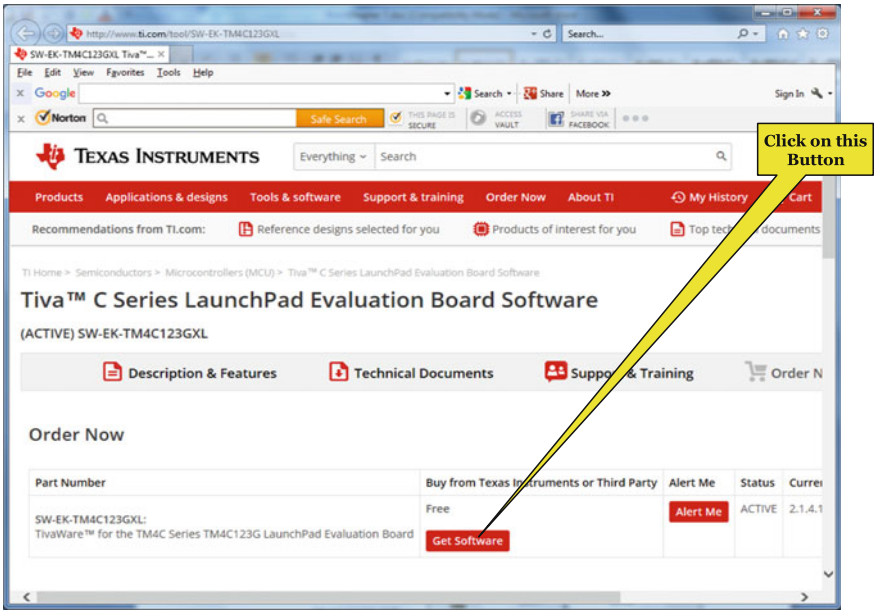


Fig. B.1 The download and install Tiva C Series Evaluation Software window

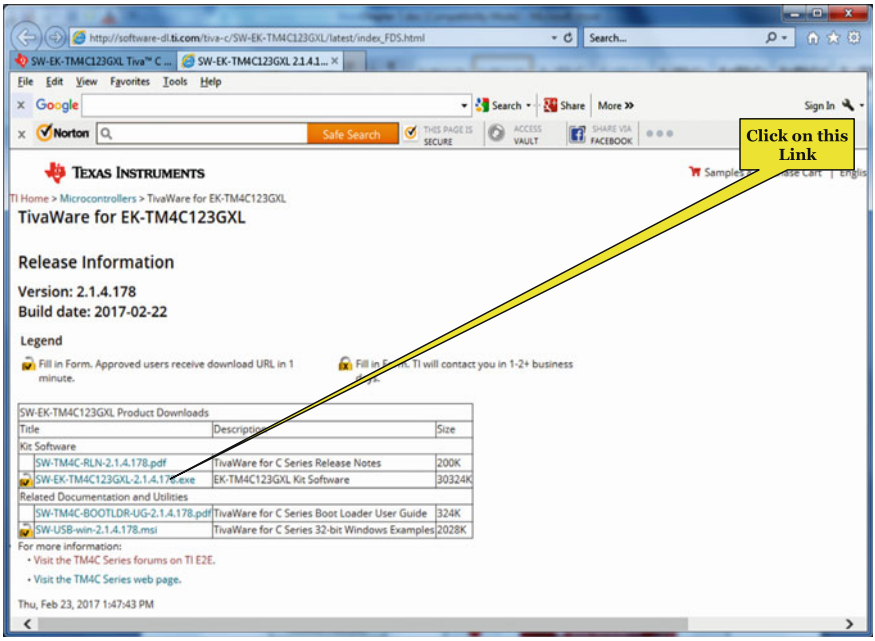


Fig. B.2 The target software package to be downloaded

button for the I CERTIFY ALL THE ABOVE IS TRUE. Then click on the **Submit** button.

5. Click on the **Download** button on the next page to begin the download process. Check your system to make sure that your computer did not block some popup file if this download process cannot start automatically in the next page.
6. A dialog box is displayed at the bottom to confirm that you want to download the software **SW-EK-TM4C123GXL-2.1.4.178.exe**. Click **Run** button to continue.
7. Click **Yes** button to the information dialog to confirm this downloading process.
8. Click on the **Next** button on the next page. Check the Agreement radio button and click on the **Next** button again to go to the next page.
9. Click on the **Next** button in the next page if you do not want to change the default location, which is: C:\ti\TivaWare_C_Series-2.1.4.178, to store this package in your computer, as shown in Fig. B.3.
10. Click on the **Install** button on the next page, as shown in Fig. B.4, to begin this installation process.

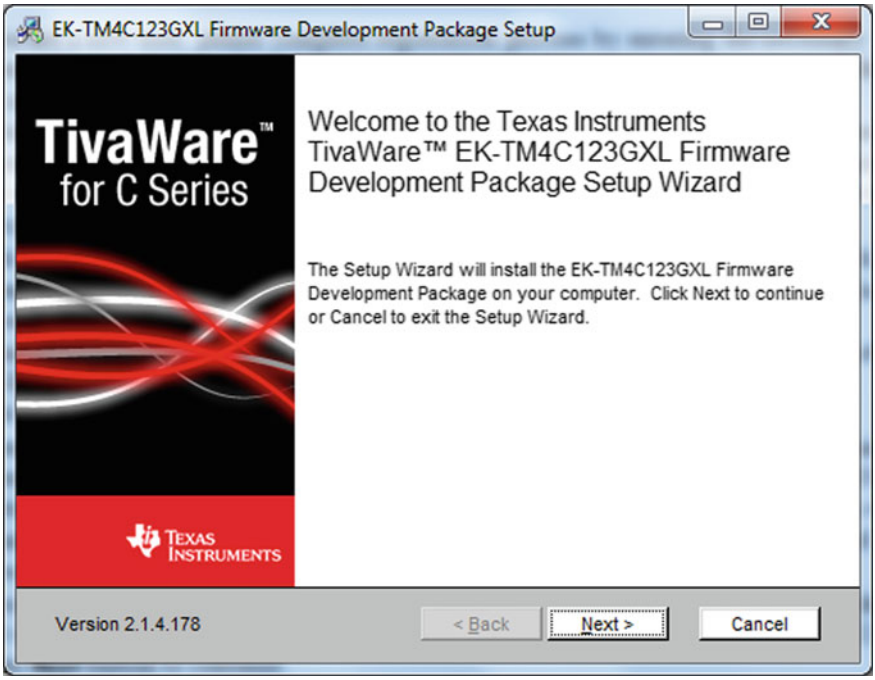


Fig. B.3 The installation page for TivaWare package

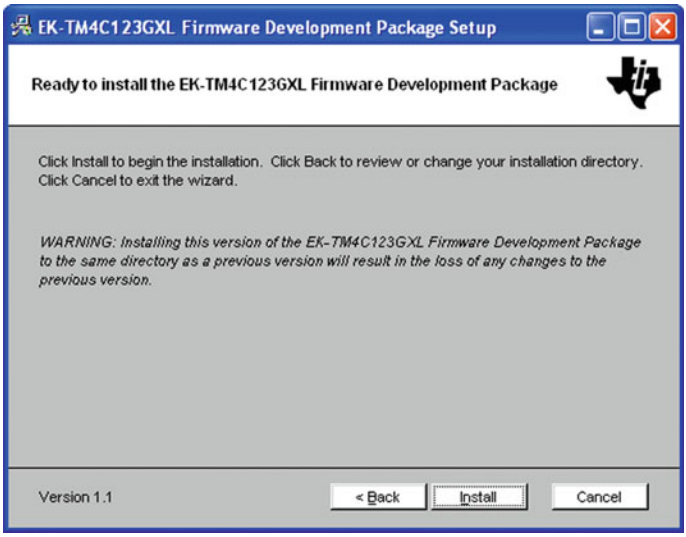


Fig. B.4 The installation starting page

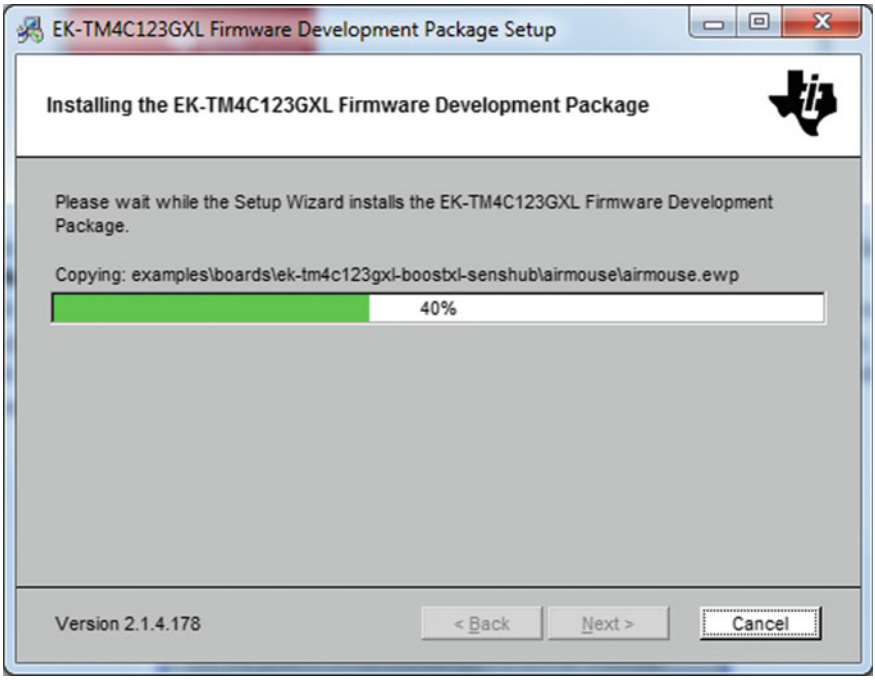


Fig. B.5 The installation process

11. Then the installation process starts, as shown in Fig. B.5.
12. Click on the **Finish** button to complete this installation when it is done.

Now if you go to your **C:\ti** directory in your host computer, you can find that the software package **TivaWare_C_Series-2.1.4.178** has been installed.

Appendix C
Download and Install Stellaris ICDI and Virtual COM Port

Tiva™ C Series evaluation and reference design kits provide an integrated Stellaris In-Circuit Debug Interface (ICDI), which allows programming and debugging of the onboard TM4C123G microcontroller. The Stellaris ICDI can be used with the Stellaris LM Flash Programmer as well as any of the Tiva C Series supported toolchains such as Texas Instruments' Code Composer Studio. Only JTAG is supported. This appendix presents the instructions for installing the appropriate drivers on the host computer.

To debug and download the custom application programs in the microcontroller's Flash memory and use Virtual COM Port connectivity, install the following drivers on the host computer:

- Stellaris Virtual Serial Port
- Stellaris ICDI JTAG/SWD Interface
- Stellaris ICDI DFU

These drivers provide the debugger with access to the JTAG interface and the host PC with access to the Virtual COM Port.

Before you can connect your Tiva TM4C123G based evaluation board to your host PC, you need first to download the Stellaris ICDI Drivers.

© Springer Nature Switzerland AG 2019

Y. Bai and Z. S. Roth, *Classical and Modern Controls with Microcontrollers*,
Advances in Industrial Control, https://doi.org/10.1007/978-3-030-01382-0

C.1 Download Stellaris ICDI Drivers

This download process is to get the necessary drivers for your Tiva evaluation or reference design kit from the Stellaris ICDI Drivers tool folder on the TI website. Then extract the files from the compressed folder to a known location on your Windows-enabled host PC. Perform the following operations to complete this process:

1. Go to the URL: http://www.ti.com/tool/stellaris_icdi_drivers and click on the **Download** button related to the current version, as shown in Fig. C.1.
2. Click on the **Save** button to save these drive files to the default Downloads folder on your host PC.
3. Click **Open Folder** button to begin the unzip process for the downloaded zip driver files.
4. Double click on the downloaded zip file **spmc016a.zip** to unzip it, and click on the **Next** button and another **Next** butt, then select a target folder to save the unzipped driver files. Using the default folder is good enough but you need to remember this location since we may need to move these driver files to our desired location later.
5. Click on the **Unzip** button to begin the unzip process.

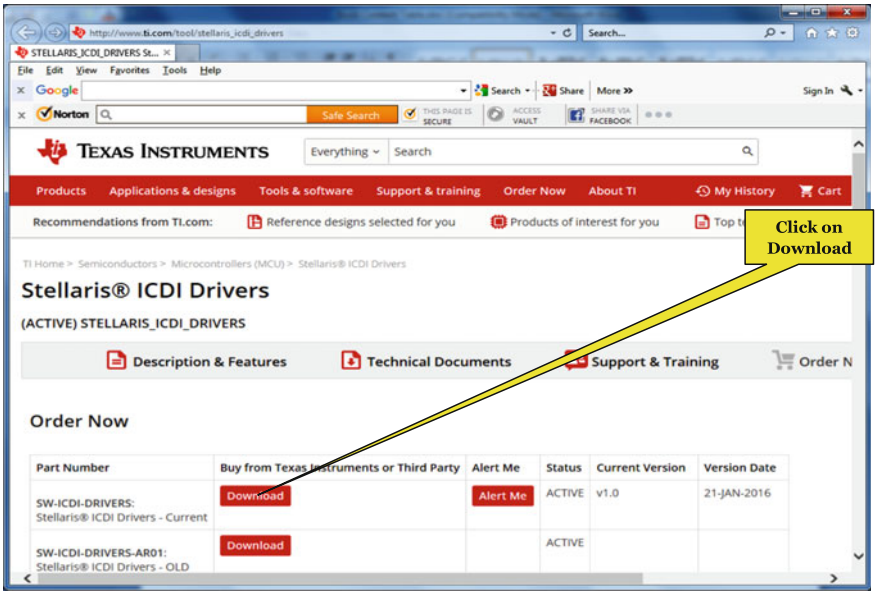


Fig. C.1 The download page on the TI site

6. All Stellaris driver files are located in a folder: *stellaris_icdi_drivers* under your default unzip folder when this unzip process is done. Click on the **Finish** button on the Winzip window to close this unzip process.
7. Now, move this folder with all stellaris drivers to your root drive C:/ in your host computer. Of course, you can save these driver files in any folder you like. But we prefer to save this folder under the root drive to make it easy to be found. Remember that all stellaris drivers now are stored in our target folder *C:/ stellaris_icdi_drivers*.

After the download process is complete, next we need to install these drivers onto your host computer.

C.2 Install Stellaris ICDI Drivers

Depends on the operating system you are using for your host PC, the installation process is different. Let's do this one by one starting from Windows XP, and then Windows 7, Windows 8, and Windows 10.

C.2.1 Install Stellaris ICDI Drivers on Windows XP

Follow these directions to install the drivers on a host PC that is running Windows XP.

1. Connect your Tiva C Series TM4C123G LaunchPad based EVB to your host PC with the USB cable coming with your development kits.
2. When the Tiva board is connected to the host PC for the first time, Windows starts the *Found New Hardware* Wizard and prompts to install the drivers for the Stellaris Virtual Serial Port. Check the *Install from a list or specific location (Advanced)* radio button and then click on the **Next** button (Fig. C.2).
3. Select *Search for the best driver in these locations*, and check the *Include this location in the search* option, as shown in Fig. C.3. Click on the **Browse** button to browse to the known location on your host PC of the driver installation files, in our case, it is *C:/stellaris_icdi_drivers*. Click on the **OK** button, and then click on the **Next** button to continue.
4. A warning may appear during the installation process to caution that the driver is not signed; click on the **Continue Anyway** button to proceed. The wizard displays a *Please wait while the wizard searches…* status window as shown in Fig. C.4. No user action is required at this point.

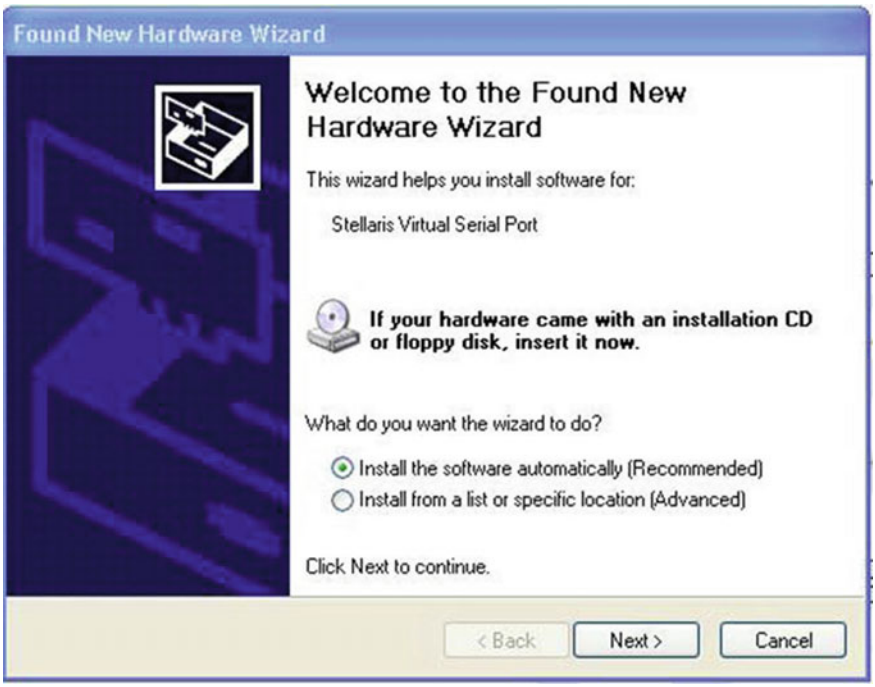


Fig. C.2 Found new hardware wizard

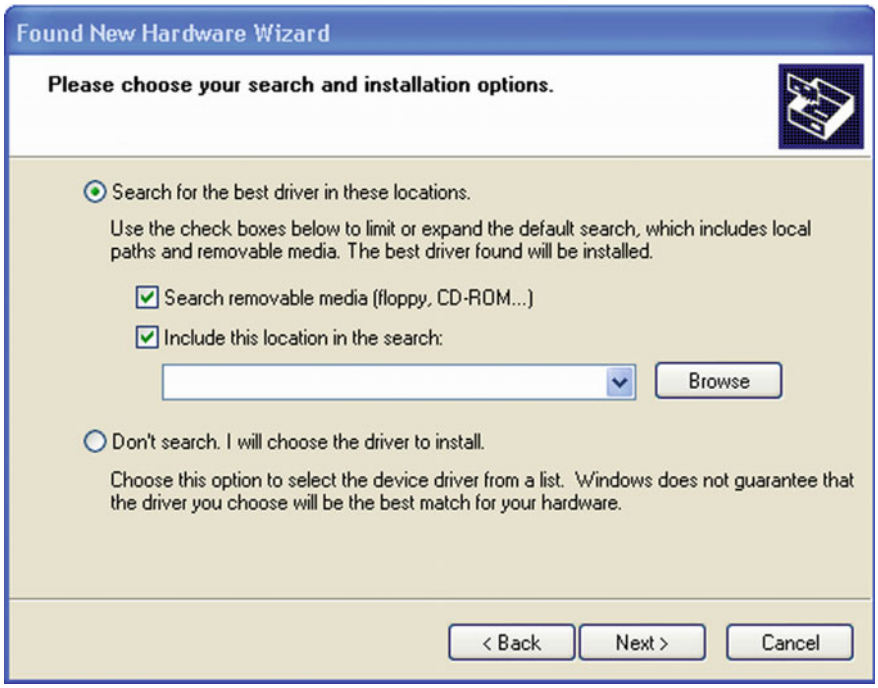


Fig. C.3 Search the best driver wizard

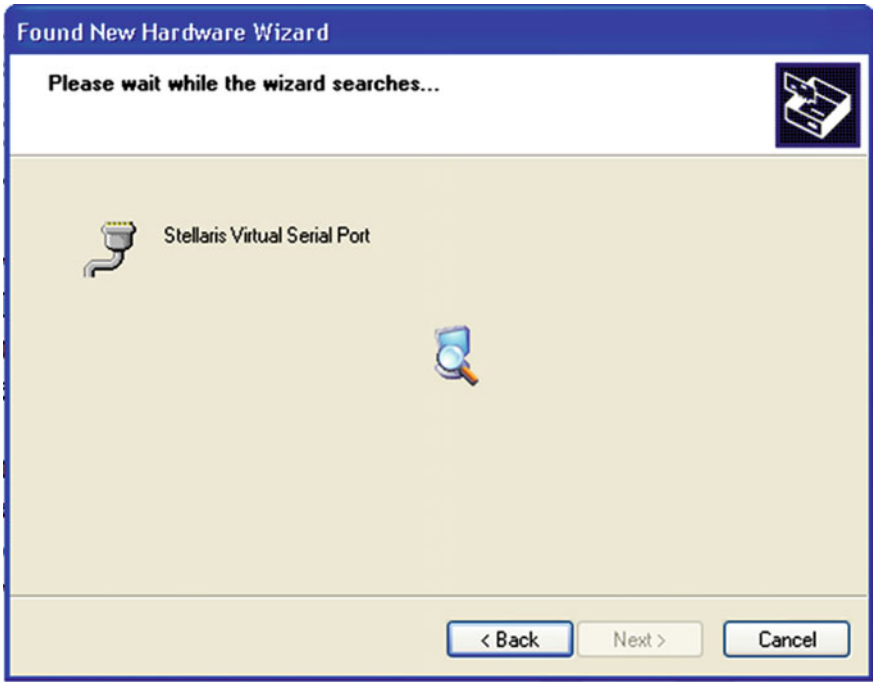


Fig. C.4 Search Virtual Series Port wizard

5. The wizard then displays a *Please wait while the wizard installs the software…* status window as the software is installed.
6. After the installation of the Stellaris Virtual Serial Port drivers completes, click on the **Finish** button shown in Fig. C.5 to close the dialog box and this installation process.

The drivers for the Stellaris Virtual Serial Port have now been installed.

The *Found New Hardware* Wizard appears a second time for the Stellaris ICDI JTAG/SWD Interface, and then one more time for the Stellaris ICDI DFU Device drivers. Follow the same instructions to install the drivers for these two devices.

Confirmation that these device driver installed correctly can be found by launching the Windows Device Manager and right-clicking to select *Scan for Hardware Changes*. This scan updates the Device Manager properties list. Most of the time, the Device Manager refreshes the properties list automatically. The Stellaris Virtual Serial Port, the Stellaris ICDI JTAG/SWD Interface, and the Stellaris ICDI DFU Device should now appear in the list. This action indicates that the drivers have been installed successfully.

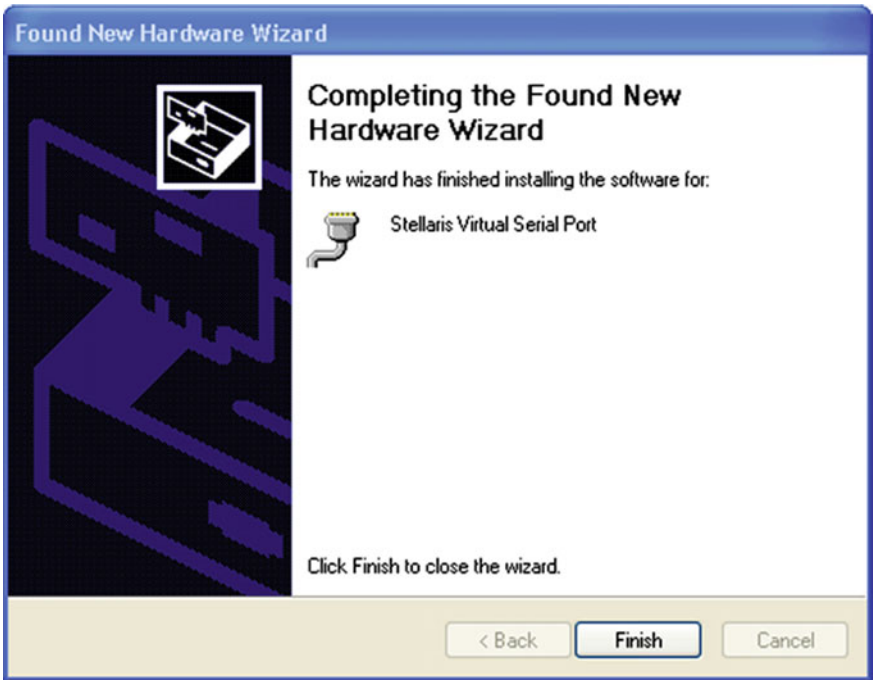


Fig. C.5 The Installation completing wizard

When these drivers are properly installed, Windows automatically detects any new Tiva boards (with a Stellaris-based ICDI) that are connected to the computer, and installs the required drivers.

C.2.1 Install Stellaris ICDI Drivers on Windows 7/8/10

Follow these directions to install the drivers on a host PC that is running Windows 7 or Windows 8.

1. Connect your Tiva C Series TM4C123G LaunchPad based EVB to your host PC with the USB cable coming with your development kits.
2. When the Tiva board is connected for the first time, the Windows 7 or Windows 8 system immediately searches for signed drivers (Fig. C.6). Wait until this process times out. When this searching process is complete, a screen shown in Fig. C.7 appears.
3. Open the Windows Device Manager, which is located at the *Control Panel*|*Hardware and Sound*|*Devices and Printers*. Under the category *Other devices*, you should see three In-Circuit Debug Interface devices with yellow exclamation marks, as shown in Fig. C.8.

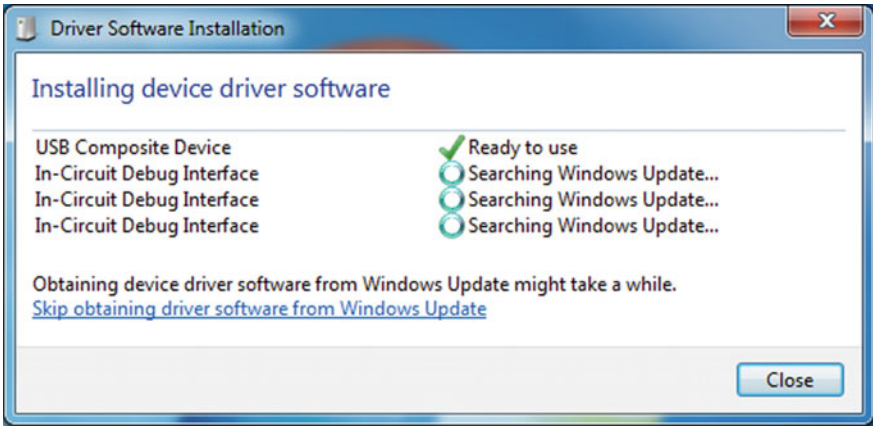


Fig. C.6 The ICDI searching wizard

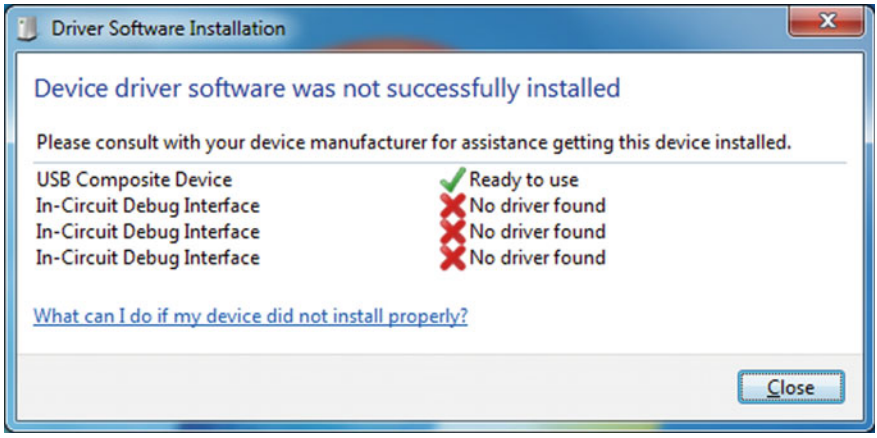


Fig. C.7 The ICDI searching completion wizard

4. Right-click one of these device entries and select *Update Driver Software* (Fig. C.9).
5. Then select the *Browse my computer for driver software*, as shown in Fig. C.10.
6. Click on the **Browse** button to browse to the location we installed Stellaris ICDI Driver, which is *C:\stellaris_icdi_drivers,* as shown in Fig. C.11. Then click on the **OK** and the **Next** buttons.

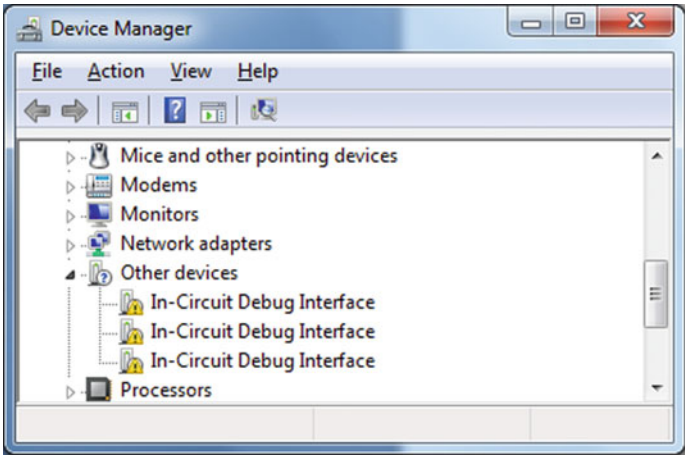


Fig. C.8 The Other Devices wizard

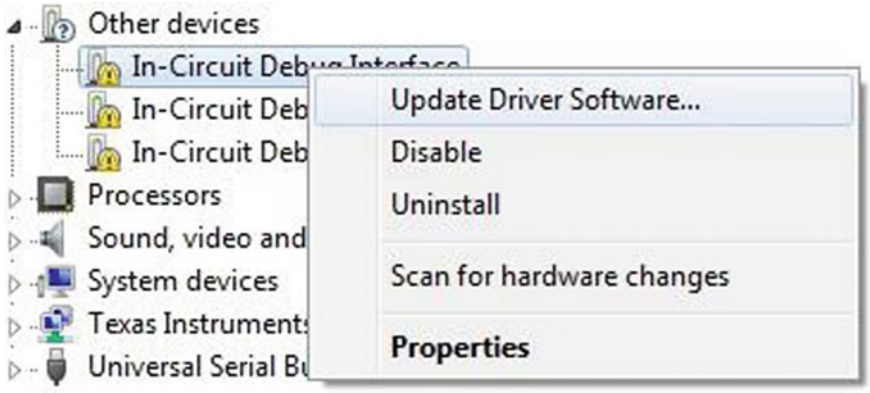


Fig. C.9 The Update Driver Software wizard

7. A warning may appear that says Windows cannot verify the publisher of this driver software. This message appears because the driver is not signed. Click on the *Install this driver software anyway* to proceed.
8. Then the Stellaris ICDI drivers will begin the installation process and all related drivers will be installed in the host computer.
9. When the installation process is complete, a closing wizard appears, as shown in Fig. C.12. Click on the **Close** button to complete this installation process.

As the ICDI driver have been installed, immediately you can find this driver in the folder *Stellaris In-Circuit Debug Interface*, as shown in Fig. C.13.

Now, you can close the Device Manager and the Control Panel.

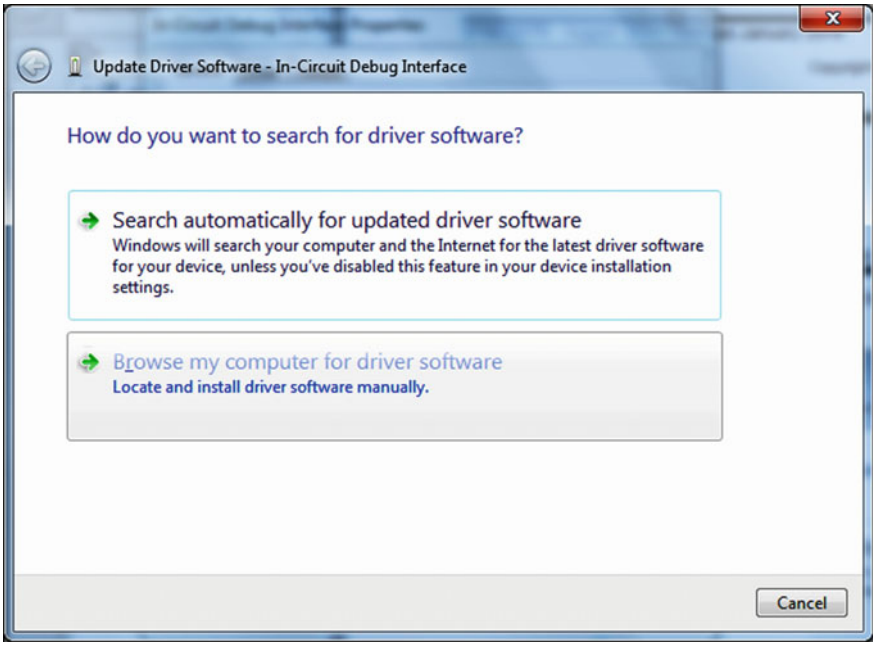


Fig. C.10 Select Browse my computer for driver software

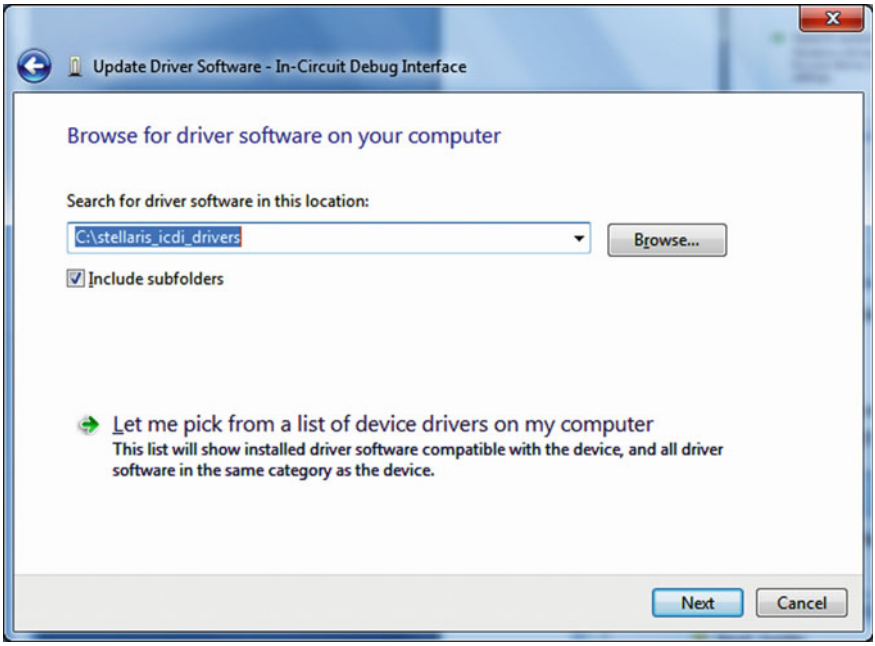


Fig. C.11 The browse wizard

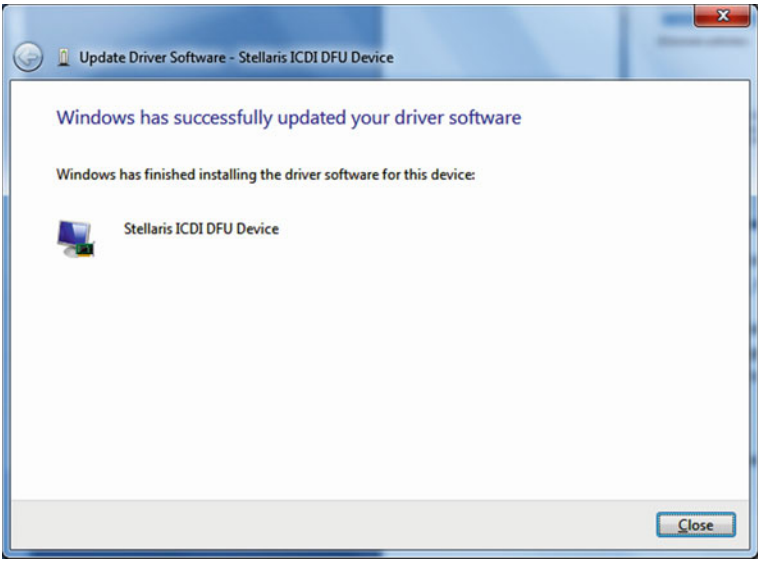


Fig. C.12 The closing wizard

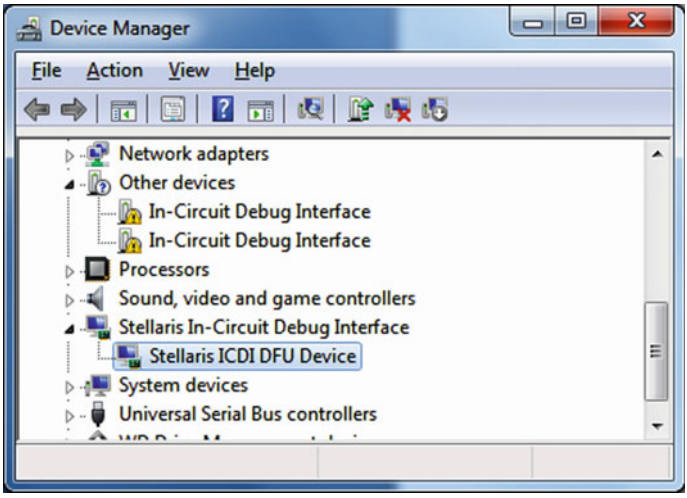


Fig. C.13 The installed ICDI driver

Index

A
Absolute convergence, 18
Absolute position encoder, 142
Acceleration constant, 112
Active-X control, 538
Additional Dependencies, 542
Additional Library Directories, 542
Advanced High-Performance Bus (AHB), 40, 41, 43, 47, 57–59
Advanced Peripheral Bus (APB), 40, 41, 43, 47, 57–59
Aerodynamic friction coefficient, 334
Algebraic Riccati equation, 285, 286
Analog controllers, 30
Analog systems, 30
Analog-To-Digital Converter (ADC), 31, 32, 143
Angle condition, 202, 203, 209, 213, 214, 217, 234
Asymptotes, 204–207, 237–240, 305, 306
Asymptotic curve, 238
Asymptotic stability, 421, 424, 429, 430
Auxiliary polynomial, 117, 118

B
Back electromotive force, 105
Background noise, 503
Bang–bang control, 394, 397, 427, 430
Bell-shaped waveform, 446
Biomedical compartmental modeling, 361
Block diagram, 4, 21, 24, 25, 28, 33–35, 38, 47, 48, 62, 68, 74, 75, 85, 89, 90, 111, 121, 126, 145, 146, 158, 165, 176, 190, 267, 278–281, 288, 296, 298, 299, 305–307, 312, 313, 327, 335, 337, 386, 434, 456–458, 472, 492, 500, 506, 537, 552, 553, 565–567, 569, 575
Bode format, 237, 239, 240, 244
Bode frequency plot, 233, 236, 241
Bode frequency-related functions, 219
Bounded-Input-Bounded-Output (BIBO), 114
Breakaway point, 204, 205
Break frequency, 238

C
Center of Gravity (COG) method, 5, 439, 451, 454, 516, 517, 519, 529
Characteristic equation, 113–117, 202, 205–207, 215, 218, 234, 271, 283, 286, 288, 289, 291, 294, 297, 312, 314
Characteristic root, 19
Classical set, 440–444, 446, 456
Clock/direction mode, 40, 44–46, 59, 62, 68, 78, 145, 147–150, 152, 155, 161, 165–167, 171, 172, 176, 177, 179, 180, 301, 316, 318, 319, 489
Closed-loop control system, 5, 14–16, 21, 23, 32–34, 69, 110, 111, 144, 145, 174, 188, 192, 196, 198, 199, 202, 209, 212, 218, 224, 226, 256, 257, 264, 281, 283, 285, 295, 298–300, 305, 307, 309, 310, 313, 314, 447, 455, 457, 458, 472, 510, 513, 527, 546, 567
Closed-loop system, 5, 14, 126, 127, 192, 196, 215, 219, 222, 223, 227, 234, 246, 251, 264, 270, 271, 274, 290–292, 298, 306–310, 314, 335, 383, 475, 546
CMSIS-CORE, 74, 75, 78
Coarse fuzzy control, 504
Colored noise, 136

© Springer Nature Switzerland AG 2019

Y. Bai and Z. S. Roth, *Classical and Modern Controls with Microcontrollers*,
Advances in Industrial Control, https://doi.org/10.1007/978-3-030-01382-0

Compartmental model, 361
Compensator Editor, 258
Competition model, 343–345, 347, 348
Complementary sensitivity, 113
Complex conjugate pole, 23
Comprehensive parametric studies, 344
Configuration Properties, 489, 541
Continuous control systems, 29
Continuous signals, 30, 32
Contour, 270, 271
Control and Estimation Tools Manager, 225, 226, 231, 256–258, 261
Controllability, 13, 52, 253, 282, 283, 295, 455, 487, 536–538, 575, 576
Controllability matrix, 296, 297
Control law, 281–285, 287, 289–292, 305, 312, 341, 400, 404, 417, 427, 431
Count-down mode, 154–156, 161, 162, 172
Count-up/down mode, 55, 154–157, 161, 162, 172
Crossover frequency, 247–257, 261, 309, 311

D
Damped natural frequency, 106
Damping ratio, 106, 107, 109, 135, 198–201, 212, 217, 220, 221, 225, 226, 233, 236, 253, 257, 306, 307, 309, 311, 314
Data Control Register (GPIODATA), 49
Data Direction Register (GPIODIR), 49, 59
Data Plot View, 137
Data Views, 137
Data View Style, 137
Dead-band generator, 157, 158
Dead-band output, 157
Dead zone, 378, 379, 433
Decayed oscillation, 23
Defuzzifications, 5, 438, 439, 450, 455, 506, 546, 552
Derivative compensation K_D, 196
Diagnostics and prognostics, 135
Digital-To-Analog Converter (DAC), 32
Discrete signal, 31–33
Diverged oscillation, 23
Dominant closed-loop poles, 212, 216, 220
Dominant poles, 209, 211, 218, 221, 307
Duty cycles, 153, 157

E
EduBASE ARM Trainer, 35, 36, 72, 98
Eigenvalues, 276, 278, 285, 286, 288, 305
Eigenvectors, 278
EK-TM4C123GXL-KEIL, 87, 88
Encoder velocity, 146, 150, 303, 509, 578
Euclidean space, 274

F
Feedback control system, 11, 13, 14, 28, 112, 113, 190, 193, 229, 307–314, 361
Feedback gain, 128, 284, 286, 291, 297, 314, 400, 401, 406
Feedback law, 294, 334, 335, 425
Feedback matrix, 297
Feedback path, 14, 21, 111, 126, 173, 264, 307, 510
Final value theorem, 110–112, 188
Fine fuzzy control, 487, 504, 505
Fine fuzzy controller, 504
Finite zeros, 109, 199, 206, 207
Firing intervals, 557, 558
First-order uncertainty fuzzy set, 552
Footprint of Uncertainty (FOU), 550–552
Frequency Response Data (FRD) model, 129–131
Frequency response models, 134
Function Block Parameters, 489
Fuzzifications, 5, 438, 439, 445, 446, 506, 546, 552
Fuzzy control rules, 5, 447–449, 456, 477, 478, 506, 517
Fuzzy data, 438
Fuzzy graph, 448
Fuzzy implication rules, 448, 449
Fuzzy Inference functions, 459, 472, 479
Fuzzy Inference Process, 438, 439, 445, 455, 524, 533
Fuzzy Logic Designer, 465, 469–471
Fuzzy Logic Toolbox Graphical User Interface Tool (FLTGUIT), 459, 461, 464, 465, 472, 478, 481, 483, 506, 507, 509, 510
Fuzzy mapping rules, 448
Fuzzy-PD control algorithm, 457, 486
Fuzzy-PD controller, 457
Fuzzy-PI control algorithm, 458, 486
Fuzzy-PI controller, 458
Fuzzy sets, 5, 439, 440, 443–445, 447, 456, 473, 506, 513, 515, 550, 552, 553, 557, 576, 577
Fuzzy set union, 444

G
Gain Margin (GM), 235, 311
Gaussian noise, 136
Gaussian type-2 curve, 557
Gaussian waveform, 446, 475, 483, 515
General Purpose Input-Output (GPIO), 317
General-Purpose Input–Output (GPIO) ports, 36, 37, 38, 40, 41, 43, 44, 46–50, 52–60, 62, 67, 98, 99, 151, 152, 168–171, 175, 179, 180, 301, 316–320, 522, 533

Global asymptotic stability, 422, 424, 425
Global convergence condition, 400, 401
GPIO ADC Control Register (GPIOADCCTL), 49, 52, 60
GPIO Alternate Function Select Register (GPIOAFSEL), 48, 49, 52, 56, 57, 60, 151, 152, 169, 171
GPIO blocks, 37, 38, 43, 44, 47, 57, 99, 137
GPIO module, 40, 43, 44, 46–48, 98, 99, 152, 171
GPIO module clock, 40, 44

H
Hammerstein–Wiener models, 134
Height Method (HM), 451
High-gain saturating amplifier, 390, 398, 402, 408, 416, 431

I
Iddata object, 136
Implicit zeros, 204
Import data, 137, 140, 184
Import Models, 138
Impulse response, 114, 314
In-Circuit Debug Interface (ICDI), 2, 6, 7, 35, 60–62, 70, 73, 81, 90, 95, 98, 180, 320, 527, 593–595, 597–600, 602
Incremental position encoder, 142
Incremental sinusoidal encoder, 142
Indefinite function, 422
Index pulse, 145, 146, 148, 151
Input matrix, 276
Input–output polynomial models, 134
Integral Compensation K_I, 196
Interface MATLAB function, 539
Internal state variables, 275
Interval type-2 fuzzy sets, 549, 551, 552
Interval type-2 fuzzy sets operations, 551
Interval type-2 rule editor, 561
Inverse Laplace transform, 18, 22, 107, 125, 231
IT2 fuzzy inference system, 5, 559, 563, 577
IT2 Fuzzy Inference System Editor, 559, 560
IT2 Fuzzy Logic Toolbox, 559, 577
IT2 Membership Function Editor, 560

K
Karnik–Mendel (KM) algorithm, 554, 556, 558, 577
Keil®MDK-ARM®5.24a, 73
Keil® MDK Core, 78
Keil® MDK µVersion IDE, 83
Kirchhoff's voltage law, 17

L
Laplace transform method, 18, 20, 22
Laplace transform pairs, 19
Laplace variable, 19, 122
Linear differential equation, 20, 26, 102, 103, 425
Linear encoders, 141
Linear gray-box models, 134
Linearity condition, 26
Linear Quadratic Regulator (LQR) function, 285
Linear Time-Invariant (LTI) systems, 11, 16, 24, 26, 33, 102, 109, 113, 114, 123, 126, 128, 134, 138, 186, 224, 232, 275, 513
Linguistic variables, 438, 446–448, 473, 515
Locally positive definite function, 422, 423
Locally positive semi-definite function, 422
Loci departure angles, 205, 208
Lower Membership Function (LMF), 550, 552, 553
Lower membership grades, 556
Lyapunov Design, 425–427, 431, 432
Lyapunov Functions, 5, 420, 421, 424, 425
Lyapunov's Asymptotic Stability Theorem, 423, 424
Lyapunov's Global Asymptotic Stability Theorem, 423
Lyapunov's stability theorem, 423
Lyapunov Stability Theory, 420
Lyapunov theory, 421, 424, 425

M
Magnitude condition, 202, 203, 209, 215, 217, 218, 234
Magnitude frequency response, 233–235, 239, 242–244, 247, 248, 307, 308, 310
Manage Run-Time Environment, 92, 93
Mathematical model, 101, 102, 104, 188, 198, 274, 325
MATLAB engine library, 5, 538, 541, 542, 546
MATLAB engine method, 538
Maximum likelihood, 134
MDK-ARM® µVision®5 GUI, 84
MDK Core, 73–75, 78, 92
MDK Professional Middleware, 75
Mean of Maximum (MOM) method, 439, 450, 454
Membership Function Editor, 461, 465, 469, 471, 474
Membership Functions (MFs), 5, 438, 439, 445–447, 450, 452–457, 459–463, 466, 468–471, 473–475, 479, 481, 486, 491, 495, 499, 501, 506, 507, 513–516, 519,

520, 527, 529, 530, 536, 537, 549, 551, 553, 557, 560, 577
Minimal model, 361
Minimum time control, 394
Mitsumi 448 PPR Motor, 69, 141, 174–176, 181, 265, 298, 305, 514
Model analysis, 135
Model validation, 135
Model Views, 137, 138
Modified system models, 497, 574, 575
Motion geometry, 4, 323
Motion kinematics, 4, 323
Multiple sliding surfaces, 411
Multi-relay SMC strategy, 417

N
Noise reduction, 495, 496, 499, 501–505, 565, 573
Noise suppression, 502
Non-deterministic truth degree, 550
Nonideal relays, 390
Nonlinear ARX models, 134, 135, 140
Nonlinear control theory, 27
Nonlinear differential equations, 27
Nonlinear gray-box models, 134
Nonlinear time-variant systems, 11
Nonparametric model, 133
Nonstationary system, 20
Normalized coefficients, 348
Normalized interaction coefficients, 346
Nyquist stability criterion, 4, 26, 113, 270, 272, 305

O
Observability, 282, 283, 295–297
Observable matrix, 296
Offline defuzzification, 455
Off-line fuzzy inference system, 536
On-chip peripherals, 37, 40, 42, 75, 88
Online defuzzification, 455
Online estimations, 135
Online fuzzy inference system, 536, 537
Open-loop control, 11–13, 16, 33, 144, 145, 359, 364, 365, 368, 371
Open-loop system, 11–13, 270, 335
Optical rotary encoder, 69, 141, 174–176
Output matrix, 276
Overshoot, 108, 109, 113, 196, 198, 201, 211, 215, 248, 251, 253, 264, 266, 281, 310, 311, 314, 315, 339, 371, 432, 493, 497, 498, 572, 575

P
Pairwise infectious contact rate, 350
Parametric model, 133
Partial fraction expansion, 19, 124, 191, 192
Peak time, 108, 192
Peripheral Driver Library, 7, 47, 61, 85–88, 180, 527
Phase frequency response, 233, 235, 239, 245, 247
Phase lag compensator, 210, 216–218, 221, 247, 248, 251–253, 307
Phase lag controller, 210, 219
Phase lead compensator, 201, 211, 212, 216, 224, 247–251, 256, 292, 306, 307, 309, 310
Phase lead controller, 209, 210
Phase Locked Loop (PLL), 44–46, 62, 171, 176, 179, 317, 318, 324
Phase Margin (PM), 113, 235, 236, 245–247, 249–253, 251, 255–257, 261, 309–311
Phase mode, 145
PID Tuner GUI, 264, 265
Port Control Register (GPIOPCTL), 48, 49, 56, 57, 60, 151, 152, 169, 171
Position counter, 145, 148
Position-error constant, 112
Positive definite function, 422, 423, 427
Power supply noise, 503
Predator equation, 341
Predator–prey model, 341–344
Prediction Error Minimization (PEM), 134
Process models, 134, 135, 185, 186, 418
Propagation time delay, 312, 315, 472, 477, 492, 507
Proportional Compensation K_p, 196
Pseudorandom noises, 133
Pulses Per Revolution (PPR), 69, 141, 143, 174–176, 181, 265, 298, 301–303, 305, 509, 510, 514, 515, 577, 578
Pulse Width Modulation (PWM), 35, 37, 38, 43, 57, 60, 69, 70, 98, 153–159, 161–175, 178–183, 188, 298, 300–303, 315, 316, 318–320, 509, 510, 515, 519, 522, 524, 528, 529, 533, 577, 578
PWM circuits, 154
PWM comparators, 154, 155
PWM counter, 154, 167
PWM Generator A Register, 162–164
PWM generator block, 37, 154, 158, 159, 161, 162, 164–168, 173
PWM Generator B Register, 164

PWM Generator Block Comparator A Register, 161
PWM Generator Block Comparator B Register, 162
PWM Generator Block Control Register, 159
PWM Generator Block Count Register, 161
PWM Generator Block Load Register, 159
PWM Master Control Register, 166–168, 173, 319
PWM output, 37, 43, 154, 156–159, 167, 168, 172–175, 178–182, 298, 300, 303, 315, 515, 522, 524, 528, 529, 533
PWM Output Enable Register, 168, 169
PWM Output Signal Generator, 156
PWM signals, 153–159, 168, 169, 171
PWM Timer Base Synchronous Register, 166, 167

Q
QEI control and status registers, 146, 147
QEI interrupt processing registers, 146, 150, 151
QEI module, 37, 38, 70, 145, 146, 148, 149, 151–153, 176, 179, 180, 524, 527, 533
QEI position registers, 146
QEI velocity registers, 146
Quadrature encoder, 141, 143, 148–150
Quadrature Encoder Interface (QEI), 35, 37, 38, 43, 69, 70, 98, 141, 145–153, 176, 179, 180, 182, 188, 509, 524, 533, 578

R
Rational polynomial function, 202
Real-time fuzzy control output, 536
Relative tolerance, 331, 342
Relay action, 404, 408, 413, 419, 431
Relay switching line, 381, 384
Remove trends, 185
Resonant peak value, 234
Rise time, 108, 109, 196, 198, 201, 211, 216, 234, 264, 281
Root locus, 4, 128, 129, 199, 201–211, 213, 215, 216, 218–226, 228–230, 234, 237, 256, 275, 300, 304–307, 309, 315, 323
Root locus diagram, 202
Root locus plot, 128, 202, 203, 220, 224, 225, 227, 234
Root locus-related functions, 121, 127, 219, 220
Rotary encoders, 141–143, 145
Routh array, 115–117, 205
Routh's stability criterion, 114, 116, 117
Routh table, 115–117
Rule editor, 461, 465, 467, 468, 471, 478, 483, 516
Rule viewer, 461, 465, 468, 469, 479, 480
Run-Mode Clock Configuration Register (RCC), 46, 166, 167, 171, 179, 318–320

S
Sample-holder, 31
S-curve waveform, 446
Second-order, 552
Second-order uncertainty fuzzy set, 552
Self-regulation, 13
Sensitivity, 14, 16, 112
Settling time, 108, 109, 192, 196, 198, 201, 215, 216, 264, 281, 310, 314, 315, 368, 493, 497, 498, 572, 575
Set value, 11–16, 33, 101
Sigmoidal waveform, 446
SI model, 350, 355, 357, 358, 360
Simulator in the Loop, 361, 431
Simulink Browser, 267
SIRD model, 353, 355, 358, 359
SIR model, 350–353, 355
SIS model, 350, 351
SISO design GUI, 223, 225, 256
SISO Design Tool, 223, 224
SISO Tool GUI, 227
Sliding line, 390, 391
Sliding Mode Control (SMC), 29, 390, 398, 402, 409, 432, 434
Sliding motion, 385–390, 392
Sliding regime, 387, 389–391
Sliding surface, 398–400, 408, 409, 411, 412, 414, 415, 417–419, 434
Software Packs, 73–75, 584
Spring-mass-damper, 17, 20
Square-law system, 24–26
Square systems, 418
Stable oscillation, 23
State-dependent control relay, 408
State-dependent levels, 408, 410
State-space model, 130, 131, 277, 285
State-space realization, 131, 275
State-space-related functions, 219
State variable, 275–281, 283, 291, 294–297, 313–315, 325
Statistical methodology, 550
Statistical uncertainty, 550
Steady-state error, 108, 112, 196, 197, 201, 212, 220, 249, 251, 281, 309, 314, 433, 434, 498, 502, 504

Steering angle, 334, 335
Stellaris ICDI, 6, 95, 180, 320, 527, 593–595, 597–600
Superposition principle, 11, 16, 26, 27, 102, 120, 323
Surface viewer, 463, 465, 469, 470
SW-EK-TM4C123GXL, 2, 6, 7, 35, 70, 73, 78, 87, 88, 98, 587–589
Switch point, 555, 556
Symbiosis Model, 343
System identification, 121, 132–135, 137, 140, 186, 188, 194, 273, 341, 361, 365, 491, 497
System identification App, 135, 137, 140
System Identification Tool (GUI), 135, 137, 140, 181, 184
System matrix, 276, 278

T
Tacho-generator, 15
Tachometer, 15, 145, 299, 300, 473, 475
Tangential force, 325
Taylor series expansion, 120, 121
Three-position relay, 378, 379
Time-flip system, 25
Time-invariant system, 24, 25
Time-series analysis, 135
Time-varying coefficients, 20
Time-varying systems, 27, 101
TivaWare™ Driver Libraries, 39
TivaWare™ for C Series CMSIS Support, 89
TivaWare™ for C Series libraries, 86
TM4C123GH6PM MCU, 37–41, 43, 44, 47, 62, 70, 98, 99, 141, 145, 153, 154, 159, 165, 168, 173, 188, 298, 305
TM4C123GXL EVB, 7, 36, 38, 60, 62, 69, 71, 72, 91, 98, 175
To LTI Viewer, 138, 186
To Workspace, 138, 228, 259, 475, 489, 492, 565
Transfer Function Models, 134
Trapezoidal waveform, 446
Triangular waveform, 446
Two-position relay, 378
Type-2 fuzzy set, 5, 550, 552, 553, 576, 577
Type 0 system, 112
Type 1 system, 112
Type 2 system, 112
Type-reducer, 553

U
Uncertain system model, 491, 495, 497, 498
Uncompensated root locus, 209, 211
Undamped natural frequency, 106, 107, 109
Universe of discourse, 440–442, 444, 474, 516
Unnormalized interaction coefficients, 347
Unstable poles, 23, 262
Upper Membership Function (UMF), 550, 552, 553
Upper membership grades, 556

V
Velocity constant, 112, 310, 311, 314

Z
Zero-order holder, 31