

Control, Estimation, and Autonomous Flight

B. Pence
Brigham Young University - Idaho

Rev. October 28, 2024

Table of Contents

1 State-Space Modeling of Dynamic Systems	1
1.1 Review Linear Algebra	2
1.1.1 Basics	2
1.1.2 Dimensions	2
1.1.3 Identity Matrix	3
1.1.4 Rank	3
1.1.5 Determinants	3
1.1.6 Eigenvalues	3
1.1.7 Matrix Transpose	4
1.1.8 Matrix Multiplication	4
1.1.9 Matrix Inverse	6
1.1.10 Matrix Exponential	7
1.2 The State Equation	9
1.2.1 Continuous and Linear-Time-Invariant State Equations	11
1.3 Solution to $\dot{\mathbf{x}} = \mathbf{A}\mathbf{x} + \mathbf{B}\mathbf{u}$	12
1.4 Proof of the Solution to $\dot{\mathbf{x}} = \mathbf{A}\mathbf{x} + \mathbf{B}\mathbf{u}$	14
1.5 The Output Equation $\mathbf{y} = \mathbf{C}\mathbf{x} + \mathbf{D}\mathbf{u}$	18
1.6 Eigenvalues of A in $\dot{\mathbf{x}} = \mathbf{A}\mathbf{x} + \mathbf{B}\mathbf{u}$ and Stability	22
1.6.1 Eigenvalues and Transient Time	24
1.6.2 Eigenvalues of first order ODEs	26
1.6.3 Eigenvalues, time constants, and frequency response of 2nd order ODEs	26
1.7 Expressing Differential Equations in State-Space Form	27
1.7.1 Expressing First Order ODEs in State-Space Form	27
1.7.2 Expressing Multiple First Order ODEs in State-Space Form	28
1.7.3 Expressing Higher-Order ODEs in State-Space Form	29
1.7.4 ODEs with Time-Derivatives of Forcing Functions	30
1.8 Numerical Solutions of LTI State-Space Systems	33

TABLE OF CONTENTS

2 Laplace Transfer Function Modeling	39
2.1 Laplace Transforms	40
2.1.1 Proofs of Selected Laplace Transform Properties	41
2.2 Modeling Dynamic Systems using Laplace Transfer Functions	43
2.2.1 Representing an ODE as a Transfer Function	43
2.2.2 Convert State-Space to Transfer Functions	44
2.2.3 Convert Transfer Functions to State-Space	46
2.3 Coordinate Transformations in State-Space Systems	50
2.3.1 Coordinate Transformation to Controllable Canonical Form	51
2.3.2 Coordinate Transforms to Observable Canonical Form	55
2.3.3 Coordinate Transforms do not Affect Transfer Functions	57
2.4 Poles and Zeros of Laplace Transfer Functions	58
2.4.1 Poles of Transfer Functions	58
2.4.2 Zeros of Transfer Functions	59
2.4.3 Pole-Zero Cancellation and Stability	63
2.5 Zero-Pole-Gain Form of a Transfer Function	65
2.6 Final Value Theorem	66
2.6.1 Final Value for Constant Inputs	67
2.6.2 Steady-State Gain of Stable Transfer Functions	67
2.7 Transfer Functions as Complex Numbers	68
2.7.1 Review of Complex Numbers	68
2.7.2 Conversions between Complex Number Forms	70
2.7.3 Mathematical Operations of Complex Numbers	71
2.7.4 Magnitude and Phase Angle of Complex Fractions	71
2.7.5 Magnitude and Phase of a Transfer Function	72
2.8 Frequency Response Plots and Bode Plots	73
2.8.1 Frequency Response Plots	73
2.8.2 Bode Plots	75
2.8.3 Drawing Bode Magnitude Plots by Hand	78
2.8.4 Drawing Bode Phase Plots by Hand	81
2.8.5 Natural Frequency and Damping Ratios on Bode Plots	82
2.9 Nyquist Plots of Transfer Functions	84
3 Discrete-Time Linear System Modeling	89
3.1 Discrete State Space	89
3.1.1 Difference Equations	90
3.1.2 The Z-Transform and Transfer Functions	92
3.1.3 Zero-Pole-Gain Form of a Z-Domain Transfer Function	94
3.1.4 Converting Between Laplace and Z-Domain Transfer Functions	94
3.1.5 Conversion from Discrete State-Space to a Difference Equation	97
3.1.6 Converting Difference Equations to Discrete State-Space	100
3.1.7 Stability of Discrete-Time Systems	102
3.2 Converting Discrete to Continuous State-Space	104
4 Numerical Solutions of Nonlinear Systems	113

TABLE OF CONTENTS

4.1	The Euler Method for Multivariable $\dot{x} = f(x, u)$	113
4.2	Converting Nonlinear ODEs to $\dot{x} = f(x, u)$	114
4.2.1	Simulation Examples	116
4.3	The Runge-Kutta Method for Multivariable $\dot{x} = f(x, u)$	120
4.3.1	Simulation Examples	121
4.4	Linearizing Nonlinear Systems	127
4.4.1	Taylor Series Expansions	127
4.4.2	Multivariable Taylor Series Expansion	127
4.4.3	Linearization of $\dot{x} = f(x, u)$ around x^* and u^*	127
4.4.4	Linearization around Constant x^* and u^*	130
4.4.5	Linearization around the Values $x^* = x_k$ and $u^* = u_k$	130
5	Block Diagrams of Control Systems	135
5.1	Basic Block Diagrams	135
5.2	Feedforward Path, Feedback Path, and the Feedback Loop	137
5.2.1	Block Diagrams with Multiple Inputs and Outputs	139
5.2.2	Block Diagrams with Multiple Feedback Paths	142
5.3	Control Systems	144
5.3.1	Feedforward Controllers	145
5.3.2	Feedback Controllers	145
5.3.3	The Famous PID Controller	146
5.3.4	Integrator Anti-Windup	149
5.3.5	Modeling Nonlinear ODEs using Integrators	153
5.4	Block Diagrams of Discrete-Time Systems	154
6	Modeling Dynamic Mechatronic Systems	155
6.1	Translational Dynamics and Vibrations	156
6.1.1	Modeling a Mass	157
6.1.2	Modeling a Spring	157
6.1.3	Modeling a Damper	158
6.1.4	d'Alembert's Principle	158
6.1.5	Modeling a Mass-Damper System	158
6.1.6	Modeling a Spring-Damper System	159
6.1.7	Modeling a Mass-Spring-Damper System	160
6.1.8	Frequency Response of a Mass-Spring-Damper System	163
6.1.9	Mass-Spring-Damper Systems with Multiple Masses	166
6.1.10	Mass-Spring-Damper Systems Modeled as a Longitudinal Wave Propagation	169
6.2	Rotational Dynamics and Vibrations	169
6.3	Modeling Dynamic Electrical Circuits	171
6.3.1	Basics of Capacitors	172
6.3.2	Basics of Inductors	174
6.3.3	Inductors store energy in the form of a magnetic field	174
6.3.4	Units of inductance and symbols for an inductor	175
6.4	Modeling Electrical Circuits with $\dot{x}=\mathbf{Ax}+\mathbf{Bu}$	177
6.4.1	Modeling Capacitors	177

TABLE OF CONTENTS

6.4.2	Modeling Inductors	178
6.4.3	Modeling Resistors	178
6.4.4	State-Space Modeling of Electrical Circuits	178
6.4.5	Modeling Electrical Circuits in the Laplace Domain	181
6.4.6	Equivalent Impedance of a Circuit	186
6.4.7	Electrical Circuits with Switches and Diodes	187
6.4.8	Switches	187
6.4.9	Diodes	187
6.4.10	Buck Converter: A Switch and Diode Circuit Example	189
6.5	Summary: Steps for Solving Electrical Circuits	193
6.6	Modeling Batteries using Equivalent Circuit Models	197
6.7	Modeling Electrical Circuits with Op Amps	199
6.7.1	Rules for Modeling Op Amps with Feedback	200
6.8	Modeling DC Motors	207
6.9	Modeling Dynamic Diffusion and Heat Transfer	210
6.9.1	Flux	211
6.9.2	Flux Driving Forces	211
6.9.3	Conservation	212
6.9.4	Conservation of Mass	212
6.9.5	Conservation of Energy	213
6.9.6	Conservation and Flux in Finite Volumes	213
6.9.7	The Heat Equation	215
6.9.8	The Finite Volume Method	216
6.9.9	Boundary Conditions	216
6.9.10	Neumann Boundary Conditions	217
6.9.11	Dirichlet Boundary Conditions	217
6.9.12	Diffusion Equation Solution by $\dot{x} = Ax + Bu$	217
6.9.13	Heat Equation Solution by $\dot{x} = Ax + Bu$	218
6.9.14	Example: Carburization	219
6.9.15	Problem Statement	220
6.9.16	Finite Volume Solution	220
6.9.17	MATLAB Code	222
6.9.18	Simulation Results	224
6.9.19	Analytical Solution	224
7	Modeling Flight Dynamics: North-East-Down	227
7.1	Equations for North-East-Down (NED) Coordinates	228
7.1.1	Airplane Parameters for NED	228
7.1.2	Fixed-Wing Aircraft Equations for NED Coordinate System	230
7.2	Derivation of the Airplane Equations of Motion	233
7.2.1	Quaternion Rotations	233
7.2.2	Properties of Orthonormal Rotation Matrices	234
7.2.3	Angular Velocity and Quaternions	235
7.2.4	Coordinate Frame Translations	235
7.3	Equations of Motion	235

TABLE OF CONTENTS

7.3.1	General 6 DOF Motion of a Rigid Body	235
7.3.2	State-Equations for 6 DOF Rigid Body Motion	237
7.3.3	Equations of Motion for an Aircraft	239
7.4	Airspeed, Angle of Attack, and Side-slip Angle	241
7.5	Wind Model	241
7.6	Input Commands	243
7.7	Modeling Propellers	244
7.7.1	Propeller Thrust	244
7.7.2	Propeller Torque	246
7.7.3	Motor Equations	247
7.7.4	Computational Simplifications for Propeller Speed	248
7.7.5	Summary of Propeller Equations	248
7.8	Gravitational Forces	250
7.9	Aerodynamics of Fixed-Wing Aircraft	250
7.9.1	Lift, Drag, and Side-slip Forces	250
7.9.2	Aerodynamic Torques	251
8	Airplanes, Microcontrollers, and Sensors	255
8.1	Boomerang Warbler	255
8.2	Flight Hardware Bill-of-Materials	259
8.3	Reading Measurements from the IMU Sensor	260
8.3.1	Magnetometer Calibration	265
8.4	Reading Measurements from the BMP280 Sensor	273
8.5	Reading Measurements from the GPS Sensor	274
8.6	Interpreting GPS Data	276
8.6.1	Interpreting Latitude and Longitude	279
8.6.2	Latitude and Longitude to Distance in Meters	280
8.6.3	Converting Distances to Latitude and Longitude	282
8.7	GPS Displacement Algorithm	282
8.7.1	Testing the GPS Displacement Algorithm	283
8.7.2	GPS Velocity Algorithm	284
9	Signal Filtering and Linear State Estimation	285
9.1	Low-Pass, High-Pass, Band-Pass, and Band-Stop Filters	286
9.1.1	Transfer Functions of Low-Pass Filters	287
9.1.2	Digital Implementation of a Low-Pass Filter	289
9.1.3	Electrical Analog Implementation of a Low-Pass Filter	291
9.1.4	Transfer Functions of High-Pass Filters	292
9.1.5	Digital Implementation of a High-Pass Filter	293
9.1.6	Electrical Analog Implementation of High-Pass Filters	293
9.1.7	Band-Pass Filters	294
9.1.8	Band-Stop or Notch Filters	295
9.2	State Estimation using a Kalman Filter	296
9.2.1	The Kalman Filter Algorithm	296
9.3	Luenberger Observer State-Estimation	301

TABLE OF CONTENTS

9.3.1 Observability	301
9.3.2 The Luenberger Observer	303
9.3.3 Calculating the Observer Gain L	304
9.3.4 Comparing the Luenberger Observer with the Kalman Filter	309
9.4 Complementary Filters and Altitude	309
9.4.1 The Complementary Filter	311
9.4.2 Backward Euler Implementation of a Low-Pass Filter	312
9.4.3 Complementary Filter For Altitude Sensor Fusion	312
9.4.4 Altitude Complementary Filter Experimental Results	313
9.5 Estimating Gravity	314
9.5.1 Deriving the Gravity Estimation State-Equations	314
9.5.2 Sensor Measurements for Gravity Estimation	316
9.6 Gravity Estimation using a Kalman Filter	317
9.6.1 Simulation Example	318
9.6.2 Kalman Filtering Results: Actual Flight Data	318
9.7 Gravity Estimation Using a Low Pass Filter	321
9.7.1 Simulation and Experimental Results	322
10 Background for Estimating Orientation	325
10.1 Prediction: Quaternion Prediction with a Gyrometer	326
10.2 Prediction: Gyrometer Prediction of the Rotation Matrix	328
10.3 Prediction: Gyrometer Prediction of Gravity	328
10.4 Prediction: Gyrometer Prediction of Geomagnetic Vector	329
10.5 Measurement: Accelerometer Measurement of Gravity	330
10.6 Measurement: Magnetometer Measurement of Geomagnetic Vector	331
10.7 Update: Quaternion from Gravity and Geomagnetic Vector	332
10.7.1 Quaternion Orientation from a Rotation Matrix	333
11 Quaternion Orientation Estimation	337
11.1 The Quaternion Estimation Algorithm	337
11.1.1 Testing the Quaternion Estimation Algorithm	341
11.2 Background and Derivation of the Kok Schön Algorithm	342
11.3 Estimating Position and Orientation	344
12 Machine Learning and System Identification	347
12.1 System ID in the Time-Domain	348
12.1.1 Singular Value Decomposition	353
12.2 Nonlinear System Identification	363
12.2.1 Linear System ID with Nonlinear Input Functions	363
12.2.2 System ID with Input Neural Networks	365
12.2.3 System ID with Nonlinear Output Functions	367
12.3 System ID in the Frequency Domain	370
12.3.1 Discrete Fourier Transform (DFT)	371
12.3.2 Fast Fourier Transform (FFT)	376
12.3.3 Bode Plots from Experimental Data	377

TABLE OF CONTENTS

12.3.4 Transfer Function Models from Bode Plots	382
12.3.5 Laplace Transfer Functions from Experimental Data	382
12.4 Model Order Reduction and Deduction	382
13 Model Based Control Design	389
13.1 Full-State Feedback Control using Pole Placement	390
13.1.1 Calculate Feedback Control Gains and Feedforward Gain	391
13.1.2 MATLAB Shortcut Commands	398
13.1.3 Effects of Pole Locations on Closed-Loop Response	399
13.1.4 Determine the Controllability of a System	401
13.1.5 Calculate a Desired Closed Loop Characteristic Equation	402
13.2 Full-State Feedback with Integral Control	403
13.3 Full-State Feedback in Discrete-Time Systems using Pole-Placement	403
13.4 Observer-Based Feedback Control	406
13.4.1 Luenberger Observer-based Feedback Control	407
13.4.2 Observer-Based Feedback with Integral Control	409
13.4.3 Stability of Luenberger-Based Feedback Controllers	410
13.5 Control Design using Root Locus	414
13.6 Control Design using Bode Plots	420
13.6.1 Phase and Gain Margins of Unity Feedback Systems	420
13.6.2 Phase Margins on Bode Plots	421
13.6.3 Gain Margins on Bode Plots	421
13.7 Control Design Requirements	424
13.8 Designing Compensators Using Root Locus and Bode Plots	424
13.8.1 Compensators	425
13.8.2 Adding Poles to Compensators	426
13.8.3 Lead and Lag Compensators	428
13.8.4 Digital Implementation of Compensators	431
13.9 Successive Loop Closure Control Design	434
14 Basic Autonomous Flight	439
14.1 Combining GPS with the Quaternion Estimation Algorithm	439
14.2 Roll, Pitch, and Yaw Euler Angles	441
14.2.1 Rotations Using Euler Angles	442
14.3 Waypoint Tracking and Path Planning	443
14.4 Autonomous Control of Roll, Pitch, and Yaw	445
14.4.1 Autonomous Flight Control	445
14.5 Constraints as Guides for Selecting Feedback Control Gains	448
14.6 Modifications for Flying Wing Airplanes	449
15 C++ Crash Course 1	451
15.1 Installing Visual Studio	452
15.2 Create, Compile, and Run a C++ Program	452
15.2.1 Explanation of the Hello World Code	455
15.3 Common Engineering Calculations	456

TABLE OF CONTENTS

15.3.1 More Data Types in C++	457
15.4 Functions that Return One Variable	457
15.4.1 Do It Yourself	458
15.5 Functions that Return Multiple Variables	459
15.5.1 Do It Yourself	460
15.6 Basic Arrays and Matrices	460
15.6.1 Do It Yourself	463
16 C++ Crash Course 2	465
16.1 Creating a Program with Multiple Files	465
16.2 Header Files and Source Files	469
16.2.1 MyMatrixMath.cpp	469
16.2.2 MyMatrixMath.h	470
16.2.3 CppCrashCoursePart2.cpp	470
16.2.4 The vector Library	471
16.3 The MyMatrixMath Library	472
16.3.1 MyMatrixMath.h	472
16.3.2 MyMatrixMath.cpp	473
16.4 Solving $\dot{x} = Ax + Bu$	480
16.4.1 CppCrashCoursePart2.cpp	480
16.4.2 Do It Yourself	483
17 C++ Crash Course 3	485
17.1 Classes in C++	486
17.2 The MyMatrixClass Class	486
17.2.1 MyMatrixClass.h	487
17.2.2 MyMatrixClass.cpp	488
17.2.3 CppCrashCoursePart3.cpp	491
17.2.4 How it Works	492
17.2.5 Do It Yourself	493
17.3 Getting and Setting Private Members	493
17.3.1 MyMatrixClass.h	494
17.3.2 MyMatrixClass.cpp	495
17.4 Solving $\dot{x} = Ax + Bu$	500
17.4.1 CppCrashCoursePart3.cpp	500
17.4.2 Do It Yourself	502
18 Using C++ With MATLAB	503
18.1 Multiplying Matrices with mex Functions	503
18.2 Mex Functions With Multiple Files	509
18.3 Using Debugging Tools	514
19 Creating Custom Arduino Libraries	521
19.1 Programming the Raspberry Pi Pico with the Arduino IDE	521
19.2 Creating Arduino Libraries	523

TABLE OF CONTENTS

20 NED: The MATLAB Flight Simulator	527
20.1 MATLAB Code for NED: The MATLAB Flight Simulator	528
21 C++ and Arduino Code for Autonomous Flight	541
21.1 function_QuaternionOrientationEstimator	541
21.1.1 function_QuaternionOrientationEstimator.h	541
21.1.2 function_QuaternionOrientationEstimator.cpp	543
21.2 function_Quaternion2Euler	550
21.2.1 function_Quaternion2Euler.h	550
21.2.2 function_Quaternion2Euler.cpp	550
21.3 function_GPSwaypointAlgorithm	550
21.3.1 function_GPSwaypointAlgorithm.h	550
21.3.2 function_GPSwaypointAlgorithm.cpp	551
21.4 function_FeedbackControlOfDelta_tear	552
21.4.1 function_FeedbackControlOfDelta_tear.h	552
21.4.2 function_FeedbackControlOfDelta_tear.cpp	552
21.5 Arduino Code for the Autopilot	554

Chapter 1

State-Space Modeling of Dynamic Systems

Contents

1.1	Review Linear Algebra	2
1.1.1	Basics	2
1.1.2	Dimensions	2
1.1.3	Identity Matrix	3
1.1.4	Rank	3
1.1.5	Determinants	3
1.1.6	Eigenvalues	3
1.1.7	Matrix Transpose	4
1.1.8	Matrix Multiplication	4
1.1.9	Matrix Inverse	6
1.1.10	Matrix Exponential	7
1.2	The State Equation	9
1.2.1	Continuous and Linear-Time-Invariant State Equations	11
1.3	Solution to $\dot{\mathbf{x}} = \mathbf{A}\mathbf{x} + \mathbf{B}\mathbf{u}$	12
1.4	Proof of the Solution to $\dot{\mathbf{x}} = \mathbf{A}\mathbf{x} + \mathbf{B}\mathbf{u}$	14
1.5	The Output Equation $\mathbf{y} = \mathbf{C}\mathbf{x} + \mathbf{D}\mathbf{u}$	18
1.6	Eigenvalues of \mathbf{A} in $\dot{\mathbf{x}} = \mathbf{A}\mathbf{x} + \mathbf{B}\mathbf{u}$ and Stability	22
1.6.1	Eigenvalues and Transient Time	24
1.6.2	Eigenvalues of first order ODEs	26
1.6.3	Eigenvalues, time constants, and frequency response of 2nd order ODEs	26
1.7	Expressing Differential Equations in State-Space Form	27
1.7.1	Expressing First Order ODEs in State-Space Form	27
1.7.2	Expressing Multiple First Order ODEs in State-Space Form	28
1.7.3	Expressing Higher-Order ODEs in State-Space Form	29
1.7.4	ODEs with Time-Derivatives of Forcing Functions	30
1.8	Numerical Solutions of LTI State-Space Systems	33

Dynamic systems are everywhere! If something moves or changes, it is probably a dynamic system. Boats, trains, planes, drones, automobiles, fluid flow, heat transfer, thermodynamics, batteries, biomedical pumps, engines, and electrical circuits are just a few examples. Engineers are expected to understand dynamic systems. This book is dedicated to modeling, testing, controlling, simulating, and measuring dynamic mechatronic systems. It particularly focuses on fixed-wing autonomous airplanes and drones. There are many ways to model dynamic systems. Modeling techniques could include differential equations, Laplace transfer functions, state-space, difference equations, control block diagrams, Z-transfer functions, etc. This chapter will introduce state-space, which is one of the foundations for modeling, testing, controlling, and measuring dynamic systems.

State-space is a mathematical representation that uses linear algebra and differential or difference equations to model the behavior of dynamic systems. The first section of this chapter provides a brief review of linear algebra. People with a good understanding of matrices, eigenvalues, matrix multiplication, determinants, and matrix inversion may skip a few subsections. However, they may still want to review the subsection titled “Matrix Exponential”. The matrix exponential is defined by a power series. Many undergraduate engineering students are unfamiliar with it.

1.1 Review Linear Algebra

1.1.1 Basics

Linear algebra is used to combine linear systems of equations into matrix form. For example, the following equations:

$$\begin{aligned} 2x + 3y &= 5 \\ x - 4z &= 2 \\ 4x + 8y + 2z &= 0 \end{aligned}$$

are arranged into matrix form to be:

$$\begin{bmatrix} 2 & 3 & 0 \\ 1 & 0 & -4 \\ 4 & 8 & 2 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} 5 \\ 2 \\ 0 \end{bmatrix}$$

1.1.2 Dimensions

The sizes of matrices are denoted first by their number of rows and then their number of columns. For example, the following matrices are listed below with their dimensions:

3×3	2×3	3×2	1×3 row vector	3×1 Column Vector
$\begin{bmatrix} 2 & 3 & 0 \\ 1 & 0 & -4 \\ 4 & 8 & 2 \end{bmatrix}$	$\begin{bmatrix} 2 & 3 & 0 \\ 1 & 0 & -4 \end{bmatrix}$	$\begin{bmatrix} 2 & 3 \\ 1 & 0 \\ 4 & 8 \end{bmatrix}$	$\begin{bmatrix} 2 & 3 & 0 \end{bmatrix}$	$\begin{bmatrix} x \\ y \\ z \end{bmatrix}$

A matrix is a square matrix when the number of rows equals the number of columns

1.1 Review Linear Algebra

1.1.3 Identity Matrix

An identity matrix is a square matrix with ones on the diagonal and zeros elsewhere. The identity matrix is usually given the symbol ' I '. The following are examples of identity matrices:

$$\begin{array}{c} \text{2x2} \\ \left[\begin{array}{cc} 1 & 0 \\ 0 & 1 \end{array} \right] \end{array} \quad \begin{array}{c} \text{3x3} \\ \left[\begin{array}{ccc} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{array} \right] \end{array}$$

Any matrix, multiplied by the identity matrix will result in the same matrix. It is essentially the same thing as multiplying any variable by the number one, the variable is unchanged.

1.1.4 Rank

The rank of a matrix is the number of linearly independent vectors in the matrix. A linearly independent vector is one that cannot be duplicated by a linear combination of other vectors in the matrix. If a matrix is full rank then all of the vectors in the matrix are linearly independent. A square matrix is full rank if its determinant is nonzero.

1.1.5 Determinants

The determinant of a matrix can only be determined if the matrix is square. The determinant of a 2x2 matrix is:

$$\det \begin{bmatrix} a & b \\ c & d \end{bmatrix} = a \cdot d - c \cdot b \quad (1.1)$$

The determinant of a 3x3 matrix is:

$$\det \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix} = a \cdot \det \begin{bmatrix} e & f \\ h & i \end{bmatrix} - b \cdot \det \begin{bmatrix} d & f \\ g & i \end{bmatrix} + c \cdot \det \begin{bmatrix} d & e \\ g & h \end{bmatrix} \quad (1.2)$$

1.1.6 Eigenvalues

The eigenvalues of a matrix are important for determining stability of a dynamic system. They also are related to the system's response-time, overshoot, oscillations, and mathematical stiffness. The eigenvalues of a matrix A can be determined by solving the following equation:

$$\det(sI - A) = 0$$

where I is the identity matrix of the same size as A .

Eigenvalues of a 2x2 matrix

Example 1.1.1. Eigenvalues of a 2x2 matrix

Determine the eigenvalues of the following matrix:

$$\begin{bmatrix} 1 & 3 \\ 0 & 2 \end{bmatrix}$$

The eigenvalues are found by solving the equation:

$$\det\left(s \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} - \begin{bmatrix} 1 & 3 \\ 0 & 2 \end{bmatrix}\right) = 0$$

$$\det\left(\begin{bmatrix} s-1 & -3 \\ 0 & s-2 \end{bmatrix}\right) = 0$$

$$(s-1)(s-2) - (0)(-3) = 0$$

$$(s-1)(s-2) = 0$$

to get:

$$s_1 = 1$$

$$s_2 = 2$$

1.1.7 Matrix Transpose

A matrix transpose is basically a rotation of a matrix and can be applied to a matrix of any shape. The transpose is indicated by a superscript T: A^T . For example, the following matrices are shown with their transposes:

2x3 transposed into a 3x2:

$$\begin{bmatrix} a & b & c \\ d & e & f \end{bmatrix}^T = \begin{bmatrix} a & d \\ b & e \\ c & f \end{bmatrix}$$

2x2 transposed into a 2x2:

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix}^T = \begin{bmatrix} a & c \\ b & d \end{bmatrix}$$

1.1.8 Matrix Multiplication

When matrices are multiplied, their inner dimensions need to match. For example, a 1x2 matrix can be multiplied with a 2x3 because the inner dimensions match. The size of the resultant matrix will be the outer dimensions. For example, a 1x2 multiplied by a 2x3 will result in a 1x3. Matrix multiplication is

1.1 Review Linear Algebra

done by combining each row in the first matrix with each column in the second matrix. For example, the following is a 2x3 matrix multiplied by a 3x2 matrix.

$$\begin{bmatrix} 1 & 2 & 0 \\ 3 & 0 & 4 \end{bmatrix} \begin{bmatrix} 5 & 0 \\ 0 & 6 \\ 7 & 8 \end{bmatrix}$$

Because the inner dimensions of the two matrices (2x3)(3x2) are equal to each other (3), they can be multiplied. The result will be a matrix in size equal to the outer dimensions, or 2x2:

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} = \begin{bmatrix} 1 & 2 & 0 \\ 3 & 0 & 4 \end{bmatrix} \begin{bmatrix} 5 & 0 \\ 0 & 6 \\ 7 & 8 \end{bmatrix}$$

The value of a is calculated by multiplying the first row of the A matrix with the first column of the B matrix:

$$a = \begin{bmatrix} 1 & 2 & 0 \end{bmatrix} \begin{bmatrix} 5 \\ 0 \\ 7 \end{bmatrix} = 1 \cdot 5 + 2 \cdot 0 + 0 \cdot 7 = 5$$

The value of b is calculated by multiplying the first row of the A matrix with the second column of the B matrix:

$$b = \begin{bmatrix} 1 & 2 & 0 \end{bmatrix} \begin{bmatrix} 0 \\ 6 \\ 8 \end{bmatrix} = 1 \cdot 0 + 2 \cdot 6 + 0 \cdot 8 = 12$$

The value of c is calculated by multiplying the second row of the A matrix with the first column of the B matrix:

$$c = \begin{bmatrix} 3 & 0 & 4 \end{bmatrix} \begin{bmatrix} 5 \\ 0 \\ 7 \end{bmatrix} = 3 \cdot 5 + 0 \cdot 0 + 4 \cdot 7 = 43$$

The value of d is calculated by multiplying the second row of the A matrix with the second column of the B matrix:

$$d = \begin{bmatrix} 3 & 0 & 4 \end{bmatrix} \begin{bmatrix} 0 \\ 6 \\ 8 \end{bmatrix} = 3 \cdot 0 + 0 \cdot 6 + 4 \cdot 8 = 32$$

The resulting matrix is

$$\begin{bmatrix} 1 & 2 & 0 \\ 3 & 0 & 4 \end{bmatrix} \begin{bmatrix} 5 & 0 \\ 0 & 6 \\ 7 & 8 \end{bmatrix} = \begin{bmatrix} 5 & 12 \\ 43 & 32 \end{bmatrix}$$

1.1.9 Matrix Inverse

An inverse of a matrix is one such that the following is true:

$$A^{-1} A = I \quad (1.3)$$

$$AA^{-1} = I \quad (1.4)$$

where A is a square matrix, A^{-1} is its inverse, and I is the identity matrix. A matrix inverse is similar to dividing by a matrix. For example, when solving the following equation for x :

$$a \cdot x = b$$

the solution is found by diving by a , or using the inverse of a :

$$x = \frac{b}{a} = a^{-1}b$$

The same is true for matrices, because we do not divide by a matrix we instead pre-multiply by the inverse:

$$A \cdot X = B \quad (1.5)$$

$$A^{-1} \cdot A \cdot X = A^{-1} \cdot B \quad (1.6)$$

$$I \cdot X = A^{-1} \cdot B \quad (1.7)$$

and because any matrix multiplied by the identity matrix is the same, then we get

$$X = A^{-1} \cdot B$$

Matrix Inversion to Solve A Linear Algebra Equation

Example 1.1.2. Matrix Inverse Use matrix inversion to solve the following linear equations:

$$2x + 3y = 5$$

$$x - 4z = 2$$

$$4x + 8y + 2z = 0$$

which are arranged into matrix form to be:

$$\begin{bmatrix} 2 & 3 & 0 \\ 1 & 0 & -4 \\ 4 & 8 & 2 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} 5 \\ 2 \\ 0 \end{bmatrix}$$

Solution: These matrices have the same form as:

$$A \cdot X = B$$

thus the resulting values in the column vector X can be found from:

1.1 Review Linear Algebra

$$X = A^{-1}B = \begin{bmatrix} 2 & 3 & 0 \\ 1 & 0 & -4 \\ 4 & 8 & 2 \end{bmatrix}^{-1} \begin{bmatrix} 5 \\ 2 \\ 0 \end{bmatrix}$$
$$= \begin{bmatrix} 14.8 \\ -8.2 \\ 3.2 \end{bmatrix}$$

The last equation was found using the following command in MATLAB:

```
X = [2,3,0;1,0,-4;4,8,2] \ [5;2;0]
```

In MATLAB, the backslash (\) operator takes the inverse of the matrix preceding it and multiplies it by the matrix after it.

The inverse of a 2x2 matrix is:

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix}^{-1} = \frac{1}{\det(A)} \begin{bmatrix} d & -b \\ -c & a \end{bmatrix} \quad (1.8)$$

$$= \frac{1}{a \cdot d - c \cdot b} \begin{bmatrix} d & -b \\ -c & a \end{bmatrix} \quad (1.9)$$

A matrix only has an inverse if it is square in dimension and is not singular. When matrices are not invertible because they have more rows than columns, the Moore-Penrose pseudo-inverse (A^+) may be able to approximate the inverse:

$$A^+ = (A^T A)^{-1} A^T$$

This approximation is optimal in the least-squares sense.

1.1.10 Matrix Exponential

So far this chapter has reviewed how to multiply matrices together, which can also be used to calculate the power of a matrix ($A^3 = A \cdot A \cdot A$). It has also reviewed how to divide by a matrix using the matrix inverse. Now we will review how to calculate the matrix exponential: e^A . The matrix exponential is used to solve a system of differential equations, as will be shown in Section 1.3.

The matrix exponential is *not* an elemental operation. If A is a matrix, the MATLAB command `exp(A)` will *not* calculate the matrix exponential. It will calculate the exponential of each element of the matrix. The correct MATLAB command is `expm(A)`. For example, if the A matrix is:

$$A = \begin{bmatrix} a & b \\ c & d \end{bmatrix}$$

then the matrix exponential is not:

$$e^A = \begin{bmatrix} e^a & e^b \\ e^c & e^d \end{bmatrix}$$

Figure 1.1 The matrix exponential is not the exponential of each element.

The matrix exponential is instead defined by the infinite sum:

$$e^A = \sum_{k=0}^{\infty} \frac{1}{k!} (A)^k \quad (1.10)$$

but the infinite series usually converges quickly after only a few iterations of k .

Matrix Exponential

Example 1.1.3. Matrix Exponential

Approximate the matrix exponential of the matrix

$$\begin{bmatrix} 1 & 2 \\ 0 & 0.5 \end{bmatrix}$$

by calculating the first five terms of the power series.

Solution: The matrix exponential is:

$$e^A = \frac{1}{0!} A^0 + \frac{1}{1!} A^1 + \frac{1}{2!} A^2 + \frac{1}{3!} A^3 + \frac{1}{4!} A^4 + \dots$$

where:

$$A^0 = I = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

$$A^1 = \begin{bmatrix} 1 & 2 \\ 0 & 0.5 \end{bmatrix}$$

$$A^2 = \begin{bmatrix} 1 & 2 \\ 0 & 0.5 \end{bmatrix} \begin{bmatrix} 1 & 2 \\ 0 & 0.5 \end{bmatrix} = \begin{bmatrix} 1 & 3 \\ 0 & 0.25 \end{bmatrix}$$

$$A^3 = A^2 A = \begin{bmatrix} 1 & 3 \\ 0 & 0.25 \end{bmatrix} \begin{bmatrix} 1 & 2 \\ 0 & 0.5 \end{bmatrix} = \begin{bmatrix} 1 & 3.5 \\ 0 & 0.125 \end{bmatrix}$$

$$A^4 = A^3 A = \begin{bmatrix} 1 & 3.5 \\ 0 & 0.125 \end{bmatrix} \begin{bmatrix} 1 & 2 \\ 0 & 0.5 \end{bmatrix} = \begin{bmatrix} 1 & 3.75 \\ 0 & 0.0625 \end{bmatrix}$$

and the matrix exponential is approximately:

$$e^A \approx \frac{1}{1} \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} + \frac{1}{1} \begin{bmatrix} 1 & 2 \\ 0 & 0.5 \end{bmatrix} + \frac{1}{2} \begin{bmatrix} 1 & 3 \\ 0 & 0.25 \end{bmatrix} + \frac{1}{6} \begin{bmatrix} 1 & 3.5 \\ 0 & 0.125 \end{bmatrix} + \frac{1}{24} \begin{bmatrix} 1 & 3.75 \\ 0 & 0.0625 \end{bmatrix}$$

1.2 The State Equation

$$e^A \approx \begin{bmatrix} 1 + 1 + \frac{1}{2} + \frac{1}{6} + \frac{1}{24} & 0 + 2 + \frac{3}{2} + \frac{3.5}{6} + \frac{3.75}{24} \\ 0 & 1 + 0.5 + \frac{0.25}{2} + \frac{0.125}{6} + \frac{0.0625}{24} \end{bmatrix}$$

$$e^A \approx \begin{bmatrix} 2.7083 & 4.2396 \\ 0 & 1.6484 \end{bmatrix}$$

which is our approximation after using only 5 terms. The exact solution using all of the terms is:

$$e^A = \begin{bmatrix} 2.7183 & 4.2782 \\ 0 & 1.6487 \end{bmatrix}$$

1.2 The State Equation

The **state** x of a system is a snapshot of the system at a given time. The state can consist of multiple variables that describe the system at one point in time. Consider a car for example. At a given time t , it has a velocity v and a position p . The velocity v and position p are state variables. Combined in vector form, they are the state x of the car.

$$x = \begin{bmatrix} v \\ p \end{bmatrix} \quad (1.11)$$

State variables change with time t , so we could write the state as a function of time $x(t)$; but to simplify notation, we will not write the independent variable t . The path that the state variables take through time is called the **state trajectory**.

The **state derivative** \dot{x} is another important term. It is simply the derivative of the state with respect to the independent variable (usually time):

$$\dot{x} = \frac{dx}{dt} \quad (1.12)$$

Continuing our car example, the state derivative would be the acceleration \dot{v} and the velocity $v = \dot{p}$:

$$\dot{x} = \begin{bmatrix} \dot{v} \\ \dot{p} \end{bmatrix} \quad (1.13)$$

State-space equations usually have an **input** u that varies with time t . The input could be a forcing function, or it could just be a measured value. In the car example, the input could be the drivetrain forces that propel the car forward, or it could be the acceleration of the car measured using an accelerometer. It could be wind forces acting on the car, or it could be the changing road grade as the car travels up and down hills. The input could be a vector or array of multiple inputs. For the car example, the input u could be a vector of the drivetrain forces F , the wind velocity w , and the slope of the road θ :

$$u = \begin{bmatrix} F \\ w \\ \theta \end{bmatrix} \quad (1.14)$$

Modeling dynamic systems is the process of determining the relationship between the state x , its derivative \dot{x} , and its input u . Many engineering courses are focused on modeling. As an engineering student, you likely have taken Statics and Dynamics courses and maybe even Fluids and Thermodynamics. You may have taken Physics and Differential Equations. Principles from these courses can help you model dynamic systems.

Kinematic modeling provides one way to relate the state x to its derivative \dot{x} and input u . Velocity v is the first derivative of position p with respect to time $v = \dot{p}$. Acceleration a is the first derivative of velocity $a = \ddot{v}$ and the second derivative of position $a = \ddot{p}$. In the car example, if the acceleration a of the car is measured using an accelerometer, the input u is the acceleration $u = a$. The state x is a vector of the car's position p and velocity v . The state derivative is the derivative of position, *i.e.*, the velocity, and the derivative of the velocity, *i.e.*, the acceleration. In state-equation form, the kinematic model is

$$\begin{bmatrix} \dot{p} \\ \dot{v} \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} p \\ v \end{bmatrix} + \begin{bmatrix} 0 \\ 1 \end{bmatrix} a \quad (1.15)$$

On close inspection of Equation (1.15), can you find the kinematic relationships $v = \dot{p}$ and $a = \ddot{v}$? With $A = \begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix}$ and $B = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$, this equation is in the **linear state-equation form**:

$$\dot{x} = Ax + Bu \quad (1.16)$$

The matrix A is called the **state matrix** and B is called the **input matrix**. If A and B are constant and do not change over time, Eq. (1.16) is **Linear Time-Invariant** (LTI). The LTI state-equation form is one of the most fundamental topics of this book. We will study this equation in detail later in this chapter and throughout this book.

Dynamic modeling provides another way to relate the state x to its derivative \dot{x} and input u . Dynamic models of mechanical systems describe how forces influence the motion of the system. Dynamic models of electrical systems describe how voltage and current sources influence the flow of electricity through circuit components. Dynamic models of thermal systems relate heat sources to changes in temperatures.

Consider the car example. Newton's laws of motion, especially $\sum F = ma$, provide a dynamic model of the car. Consider the free-body-diagram in Figure 1.2. Forces on the car include the drivetrain forces F , air drag cV^2 , rolling resistance mbv , and gravity $mg \sin\theta$. The notation for car parameters are mass m (kg), gravity g (9.8 kg/m²), drag coefficient c (kg/m), velocity v (m/s), longitudinal displacement position p (m), rolling resistance coefficient b (s⁻¹), and drivetrain force F (N).

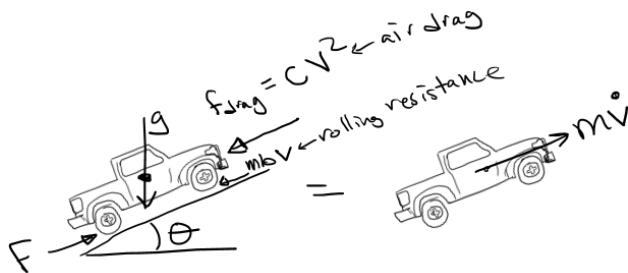


Figure 1.2 Free-body-diagram of a truck traveling up a hill on a calm day

The summation of forces is equal to mass times acceleration:

1.2 The State Equation

$$F - cv^2 - mbv - mg \sin \theta = m\dot{v} \quad (1.17)$$

The dynamic model in state-equation form is

$$\begin{bmatrix} \dot{p} \\ \dot{v} \end{bmatrix} = \begin{bmatrix} v \\ \frac{1}{m}(F - cv^2 - mbv - mg \sin \theta) \end{bmatrix} \quad (1.18)$$

Can you find Eq. (1.17) in Eq. (1.18)? Did you notice that the dynamic model also used the kinematic model $v = \dot{p}$?

The dynamic model Eq. (1.18) cannot be written in the linear state-equation form of Eq. (1.16). The function $\frac{1}{m}(F - cv^2 - mbv - mg \sin \theta)$ is nonlinear with respect to the state $x = [p \ v]$ due to the v^2 term. The input u is a vector that includes the drivetrain force F and road angle θ : $u = [F \ \theta]$, and the dynamic model is in **nonlinear state-equation form**:

$$\dot{x} = f(x, u) \quad (1.19)$$

Just as the state is a vector of state-variables, the function $f(x, u)$ is a vector of functions. Nonlinear state equations are also an important focus of this book. We will study how to solve these equations in Chapter 4. We will use nonlinear state equations throughout this book.

Concept Check!

If you ignore the term cv^2 , can you write Eq. (1.18) in the linear state-equation form $\dot{x} = Ax + Bu$? See if you can find this solution:

$$\begin{bmatrix} \dot{p} \\ \dot{v} \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 0 & -b \end{bmatrix} \begin{bmatrix} p \\ v \end{bmatrix} + \begin{bmatrix} 0 & 0 \\ \frac{1}{m} & -g \end{bmatrix} \begin{bmatrix} F \\ \sin \theta \end{bmatrix} \quad (1.20)$$

Notice that the input u does not need to be linear.

1.2.1 Continuous and Linear-Time-Invariant State Equations

The equation of a line is $y = m \cdot x + b$, where m is the slope, x is the independent variable, and b is the y-intercept. Can you recognize the equation of a line in the linear state equation?

$$\dot{x} = Ax + Bu \quad (1.21)$$

To relate the state equation to the equation for a line, set $y = \dot{x}$, $m = A$, $x = x$, and $b = Bu$. Any state equation that can be written in the form of Eq. (1.21) is a **linear state equation**.

A state-equation is **time-invariant** if the state matrix A and input matrix B are constant (do not change over time). The abbreviation for Linear-Time-Invariant is **LTI**.

Eq. (1.21) is a continuous state equation. Continuous state equations are identified by the state-derivative \dot{x} . A **continuous** state equation is one in which the state can be defined at every instance in time; *i.e.*, time is continuous. The opposite of continuous-time is **discrete-time**. In discrete-time, the state is only defined at discrete intervals or time-samples. Discrete state equations do not have time-derivatives. They will be discussed in Chapter 3. Because computers and microcontrollers take samples at discrete

intervals, they are more common in mechatronic applications than continuous state equations. As a result, most computational applications of continuous state equations require converting the continuous state equation to a discrete form. There are many techniques to do so. Most are discussed in the Discrete State Space chapter. One technique, however, is based on the exact solution to Eq. (1.21). It is presented next.

1.3 Solution to $\dot{x} = Ax + Bu$

The exact solution to the LTI state equation $\dot{x} = Ax + Bu$ from time t_k to t_{k+1} for an input u that is constant during that duration is

$$x_{k+1} = A_d x_k + B_d u_k \quad (1.22)$$

where x_{k+1} is the value of x at time t_{k+1} , A_d is the **state-transition matrix**, B_d is the **input transition matrix**, x_k is the known initial value of x at time t_k , and u_k is the (constant) value of u from time t_k to t_{k+1} . Eq. (1.22) is in discrete state-equation form because the state x is defined at discrete time samples t_k and t_{k+1} . The matrices A_d and B_d are calculated by solving the **solution matrix** F_d , whose submatrices include A_d and B_d .

$$F_d = \begin{bmatrix} A_d & B_d \\ \mathbf{0} & I \end{bmatrix} = e^{\begin{bmatrix} A\Delta t & B\Delta t \\ \mathbf{0} & \mathbf{0} \end{bmatrix}} \quad (1.23)$$

where each $\mathbf{0}$ is a vector or matrix of zeros. Each $\mathbf{0}$ submatrix must be appropriately sized to make the matrix square. The identity submatrix I (a square matrix with ones on the diagonal and zeros for all other elements) must also be appropriately sized to make the matrix square.

In the special case in which the A matrix is invertible, A_d and B_d could also be calculated by

$$A_d = e^{A\Delta t} \quad (1.24)$$

$$B_d = A^{-1}(e^{A\Delta t} - I)B \quad \text{if } \det(A) \neq 0 \quad (1.25)$$

If the A matrix is not invertible, the B_d matrix must be calculated using Eq. (1.23).

The **matrix exponential** is defined by the infinite power series:

$$e^A = \sum_{k=0}^{\infty} \frac{1}{k!} (A)^k \quad (1.26)$$

Be careful, e^A is not an element-wise exponential! For example, the MATLAB command “`expm`” calculates the matrix exponential, but the command “`exp`” calculates the element-wise exponential. Using the “`exp`” command would result in incorrect solutions.

Solution to the State Equation

Example 1.3.1. Constant acceleration

A car starts at position 0 m with a velocity of 2 m/s at time $t_1 = 0$ s. It experiences a constant acceleration of 3 m/s² over a 6 s time period. How far has the car traveled when $t_2 = 6$ s?

1.3 Solution to $\dot{x} = Ax + Bu$

Solution: We can use the state equation Eq. (1.15) and its solution Eq. (1.22) to determine the distance traveled. The state x_1 of the car at the initial time t_1 is

$$\begin{aligned} x_1 &= \begin{bmatrix} p_1 \\ v_1 \end{bmatrix} \\ &= \begin{bmatrix} 0 \\ 2 \end{bmatrix} \end{aligned}$$

because the initial position is 0 m, and the initial velocity is 2 m/s. The acceleration $a = 3 \text{ m/s}^2$ is constant throughout the duration from $t_1 = 0 \text{ s}$ to $t_2 = 6 \text{ s}$. The time-step is $\Delta t = t_2 - t_1 = 6 \text{ s}$. The kinematic state equation Eq. (1.15) for the car is

$$\begin{bmatrix} \dot{p} \\ \dot{v} \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} p \\ v \end{bmatrix} + \begin{bmatrix} 0 \\ 1 \end{bmatrix} a$$

which means that the state matrix is

$$A = \begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix}$$

The input matrix is

$$B = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$$

The state matrix A has a column of zeros and is therefore not full-rank. It is not invertible. Consequently, we cannot use Eq. (1.25), but must instead use Eq. (1.23) to solve for the state-transition matrix A_d and input transition matrix B_d . Because the timestep is $\Delta t = 6 \text{ s}$, the solution matrix F_d , the state-transition matrix A_d , and the input transition matrix B_d are found using the following MATLAB code:

```
A = [0,1;0,0]
B = [0;1]
dt = 6
Fd = expm([A*dt, B*dt; zeros(1,3)])
Ad = Fd(1:2,1:2)
Bd = Fd(1:2,3)
```

Which results in

$$F_d = \begin{bmatrix} 1 & 6 & 18 \\ 0 & 1 & 6 \\ 0 & 0 & 1 \end{bmatrix}$$

$$A_d = \begin{bmatrix} 1 & 6 \\ 0 & 1 \end{bmatrix}$$

$$B_d = \begin{bmatrix} 18 \\ 6 \end{bmatrix}$$

Can you see how A_d and B_d are submatrices of F_d ? The solution Eq. (1.22) can now be used to find the state x_2 of the car at time $t_2 = 6$ s:

$$\begin{aligned} x_2 &= A_d x_1 + B_d a \\ &= \begin{bmatrix} 1 & 6 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} 0 \\ 2 \end{bmatrix} + \begin{bmatrix} 18 \\ 6 \end{bmatrix}^3 \\ &= \begin{bmatrix} 66 \\ 20 \end{bmatrix} \end{aligned}$$

From this solution, we know that the car has traveled 66 m (and has a velocity of 20 m/s) at time $t_2 = 6$ s.

The solution to the state-space equation is only exact for an input u that is constant, or at least stepwise constant; but we would like to apply this solution to any input. Fortunately, we can find a very good approximate solution that works for any case. The main idea is that any input u can be represented by a series of steps. If we take small enough steps, then u is approximately constant between the time steps t_k and t_{k+1} . Then the solution approach is to apply Eq. (1.22) iteratively to solve the equation. This approximation approach will be discussed with examples in the discrete state space chapter.

1.4 Proof of the Solution to $\dot{x} = Ax + Bu$

The most inquisitive engineering students will demand a proof that the solution to $\dot{x} = Ax + Bu$ is found using the discrete A_d and B_d matrices as defined by Eq. (1.23). This section will provide the proof for the general matrix case in which x is a vector and A and B are matrices. First, however, it will solve the simpler equation $\dot{x} = Ax$ where the state x , its derivative \dot{x} , and A are all scalars rather than matrices.

Proof that $x_{k+1} = e^{A\Delta t}x_k$ is the solution to $\dot{x} = Ax$ for scalar variables

Note that we can write $\dot{x} = Ax$ as

1.4 Proof of the Solution to $\dot{x} = Ax + Bu$

$$\frac{dx}{dt} = Ax$$

Separating the x and t variables gives us

$$\frac{dx}{x} = Adt$$

Now we can take the integral of both sides.

$$\int_{x_k}^{x_{k+1}} \frac{1}{x} dx = \int_{t_k}^{t_{k+1}} Adt$$

The solution is

$$\ln \frac{x_{k+1}}{x_k} = A\Delta t$$

Now we take the exponential of both sides to get

$$\frac{x_{k+1}}{x_k} = e^{A\Delta t}$$

We multiply both sides by x_k to get

$$x_{k+1} = e^{A\Delta t} x_k$$

The discrete A_d parameter is $A_d = e^{A\Delta t}$ and the proof is complete.

Now we will prove the solution to $\dot{x} = Ax + Bu$ for the general case. The state x and its derivative \dot{x} are vectors, the input vector u is constant from t_k to t_{k+1} , and A and B are constant matrices. First note that because u is constant, its time-derivative \dot{u} is the zeros vector $\mathbf{0}$. We can rewrite $\dot{x} = Ax + Bu$ as

$$\begin{bmatrix} \dot{x} \\ \dot{u} \end{bmatrix} = \begin{bmatrix} A & B \\ \mathbf{0} & \mathbf{0} \end{bmatrix} \begin{bmatrix} x \\ u \end{bmatrix} \quad (1.27)$$

Now we have a new state equation

$$\dot{q} = Fq \quad (1.28)$$

where the new state q is defined as

$$q = \begin{bmatrix} x \\ u \end{bmatrix} \quad (1.29)$$

and the new extended state matrix F is

$$F = \begin{bmatrix} A & B \\ \mathbf{0} & \mathbf{0} \end{bmatrix} \quad (1.30)$$

This new extended state-equation $\dot{q} = Fq$ significantly simplifies the proof. Specifically, it will show that $q_{k+1} = F_d q_k$ which is equivalent to proving that $x_{k+1} = A_d x_k + B_d u_k$ and $u_{k+1} = u_k$ by the definitions of q_k and F_d .

Proof that $q_{k+1} = F_d q_k$ is the general solution to $\dot{q} = Fq$

Consider a trial function $q = e^{Ft} v$, where F is defined in Eq. (1.30), t is the scalar independent variable for time, and v is an unknown but constant column vector that is the same size as q . Because F is a matrix, e^{Ft} is a matrix exponential. The derivative of the trial function $e^{Ft} v$ with respect to t is

$$\begin{aligned}\frac{de^{Ft} v}{dt} &= Fe^{Ft} v \\ &= \dot{q} \\ &= Fq\end{aligned}$$

The second $Fe^{Ft} v = \dot{q}$ and third $Fe^{Ft} v = Fq$ equalities show that the trial function $q = e^{Ft} v$ satisfies the equation $\dot{q} = Fq$. At time t_k the trial function is

$$q_k = e^{Ft_k} v$$

We can solve this for the unknown vector v :

$$v = e^{-Ft_k} q_k$$

Ft_k is a square matrix, and the inverse e^{-Ft_k} is guaranteed to exist because $e^{-Ft_k} e^{Ft_k} = e^{(-F+F)t_k} = e^0 = I$, where I is the identity matrix. At time t_{k+1} the trial function is

$$q_{k+1} = e^{Ft_{k+1}} v$$

Plugging in the solution for v , $v = e^{-Ft_k} q_k$, we get

$$\begin{aligned}q_{k+1} &= e^{Ft_{k+1}} e^{-Ft_k} q_k \\ &= e^{F(t_{k+1}-t_k)} q_k \\ &= e^{F\Delta t} q_k \\ &= F_d q_k\end{aligned}$$

By definition (see Eq. (1.23)), the solution matrix F_d is the matrix exponential of the product of F (see Eq. (1.30)) and the time-step Δt :

$$F_d = e^{F\Delta t}$$

This has proved that $q_{k+1} = F_d q_k$, which also proves that F_d is the solution matrix, and $x_{k+1} = A_d x_k + B_d u_k$ is the solution to $\dot{x} = Ax + Bu$ when u is constant from t_k to t_{k+1} .

There is another way to prove that $x_{k+1} = e^{A\Delta t} x_k$ is the general solution to $\dot{x} = Ax$. As will be explained

1.4 Proof of the Solution to $\dot{x} = Ax + Bu$

later, we only need to prove the solution for $\dot{x} = Ax$, and we will be able to show that it is also valid for $\dot{x} = Ax + Bu$ if u is constant. The proof requires some understanding of the Laplace transform \mathcal{L} and its inverse \mathcal{L}^{-1} . The Laplace transform is typically a subject of a math course in ordinary differential equations. This proof assumes you already have that background.

Proof that $x_{k+1} = e^{A\Delta t} x_k$ is the general solution to $\dot{x} = Ax$

The Laplace transform of $\dot{x} = Ax$ is

$$sX - x_k = AX$$

where $X = \mathcal{L}\{x\}$ and s is the scalar Laplace independent variable; x_k is the known initial condition at time t_k . We can rewrite this equation as

$$(sI - A)X = x_k$$

or

$$X = (sI - A)^{-1} x_k$$

Consider the term $(sI - A)^{-1}$ which can also be written as $\frac{1}{s}(I - \frac{A}{s})^{-1}$. The Maclaurin series expansion (Taylor series expansion around $C = 0$) of $(I - C)^{-1}$ is

$$(I - C)^{-1} = I + C + C^2 + C^3 + \dots$$

Therefore,

$$X = \left(\frac{I}{s} + \frac{A}{s^2} + \frac{A^2}{s^3} + \frac{A^3}{s^4} + \dots \right) x_k \quad (1.31)$$

The inverse Laplace transform of $\frac{1}{s^n}$ evaluated on the time interval from t_k to t_{k+1} is

$$\mathcal{L}^{-1} \left\{ \frac{1}{s^n} \right\} = \frac{\Delta t^{n-1}}{n!}$$

and since A and x_k are constants, we can take the inverse Laplace transform of both sides of Eq. (1.31) to get

$$x_{k+1} = \left(I + A\Delta t + \frac{(A\Delta t)^2}{2!} + \frac{(A\Delta t)^3}{3!} + \dots \right) x_k$$

The series $\left(I + A\Delta t + \frac{(A\Delta t)^2}{2!} + \frac{(A\Delta t)^3}{3!} + \dots \right)$ is the expansion of the matrix exponential (see Eq. (1.26)). Therefore, we can write the equation as

$$x_{k+1} = e^{A\Delta t} x_k$$

which completes the proof.

We have proven that $x_{k+1} = e^{A\Delta t} x_k$ is the solution to $\dot{x} = Ax$. We have also shown that we can extend the state of $\dot{x} = Ax + Bu$ to $q = [x \ u]^T$. Doing so, we can write $\dot{x} = Ax + Bu$ as $\dot{q} = Fq$. Therefore, the

solution to $\dot{q} = Fq$ is $q_{k+1} = e^{F\Delta t} q_k$. Since $F = \begin{bmatrix} A & B \\ \mathbf{0} & \mathbf{0} \end{bmatrix}$, the solution to $\dot{x} = Ax + Bu$ is

$$\begin{bmatrix} x_{k+1} \\ u_{k+1}^- \end{bmatrix} = e^{\begin{bmatrix} A & B \\ \mathbf{0} & \mathbf{0} \end{bmatrix} \Delta t} \begin{bmatrix} x_k \\ u_k \end{bmatrix} \quad (1.32)$$

Where u_{k+1}^- is the value of the input just before it steps to the new value u_{k+1} at time t_{k+1} , i.e., $u_{k+1}^- = u_k$. The state transition in Eq. (1.32) can also be written as

$$x_{k+1} = A_d x_k + B_d u_k$$

where the discrete A_d and B_d matrices are calculated by solving

$$\begin{bmatrix} A_d & B_d \\ \mathbf{0} & I \end{bmatrix} = e^{\begin{bmatrix} A\Delta t & B\Delta t \\ \mathbf{0} & \mathbf{0} \end{bmatrix}}$$

We have proven the solution to $\dot{x} = Ax + Bu$ is given by Eq. (1.22) with A_d and B_d calculated by Eq. (1.23).

1.5 The Output Equation $y=Cx+Du$

State-space is the combination of a state equation and an output equation. The output equation is a mathematical relationship between the **output** y , the state x , and the input u . Similar to the state equation, the **output equation** can either be linear

$$y = Cx + Du \quad (1.33)$$

or nonlinear

$$y = h(x, u) \quad (1.34)$$

In these equations, y is the output, which can be either a scalar or a vector. C is called the **output matrix**. D is the **feedthrough matrix** or **direct transition matrix** because if it is nonzero, the input u can directly influence the output y . In Eq. (1.34), y is the output; h is a nonlinear function of the state x and the input u . Note that the output equation is not dynamic, i.e., there are no derivatives in the output equation.

To be complete, the state-space form includes both a state-equation and an output equation. The complete **linear state-space form** is

$$y = Cx + Du \quad (1.35)$$

$$\dot{x} = Ax + Bu \quad (1.36)$$

The complete **nonlinear state-space form** is

1.5 The Output Equation $y=Cx+Du$

$$y = h(x, u) \quad (1.37)$$

$$\dot{x} = f(x, u) \quad (1.38)$$

State-space can be a mixture of both linear and nonlinear equations. For example, it could have a linear state-equation but nonlinear output equation:

$$y = h(x, u) \quad (1.39)$$

$$\dot{x} = Ax + Bu \quad (1.40)$$

or vice-versa.

The following examples demonstrate that finding the output equation requires determining the mathematical relationship between the output, the state, and the input.

Output Equation

Example 1.5.1. Determine the output equation

The state-equation for the longitudinal motion of a car is given by the kinematic model Eq. (1.15):

$$\begin{bmatrix} \dot{p} \\ \dot{v} \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} p \\ v \end{bmatrix} + \begin{bmatrix} 0 \\ 1 \end{bmatrix} a$$

The output is the air-drag. Use Figure 1.2 to determine the output equation.

Solution: To determine the output equation, we need to find the mathematical relationship between the output y , the state x , and the input u . The output y is the air-drag force on the car. The state x is the position p and velocity v of the car. The input u is the car's acceleration a . As shown in Figure 1.2, the air-drag force is given by the equation $c v^2$. Since v is one of the states, and it is squared, the output equation is the nonlinear equation

$$y = c v^2$$

The next example numerically simulates a state-space system using MATLAB.

Output Equation and Trajectory

Example 1.5.2. Output trajectory

The state-equation for the dynamic model of the longitudinal motion of a car is given by Eq. (1.18):

$$\begin{bmatrix} \dot{p} \\ \dot{v} \end{bmatrix} = \begin{bmatrix} v \\ \frac{1}{m}(F - cv^2 - mbv - mg \sin \theta) \end{bmatrix}$$

The output is the car's velocity. Determine the output equation. Then use MATLAB and the linear approximation given in Eq. (1.20) to plot the velocity of the car over 10 second given the following parameters:

1. Time step $\Delta t = 0.01$ s
2. Time vector $t \in [0, 10]$ s
3. Initial position $p(0) = 0$ m
4. Initial velocity $v(0) = 18$ m/s
5. Mass $m = 1000$ kg
6. Gravity $g = 9.8$ m/s²
7. Rolling resistance $b = 0.01$ 1/s
8. Road grade angle $\theta = \frac{\pi}{30}(t - 5)$ rad
9. Force $F = 900 - 50t$ N

Solution: To determine the output equation, we need to find the mathematical relationship between the output y , the state x , and the input u . The output y is the velocity v of the car. We can tell by the state-derivative $\dot{x} = \begin{bmatrix} \dot{p} \\ \dot{v} \end{bmatrix}$ that the state x is the position p and velocity v of the car:

$$x = \begin{bmatrix} p \\ v \end{bmatrix}$$

Since v is the second variable in the state vector, the output equation is the linear equation

$$y = \begin{bmatrix} 0 & 1 \end{bmatrix} x$$

Can you see the relationship $y = v$ is this equation? In this linear output equation, the output matrix C is $\begin{bmatrix} 0 & 1 \end{bmatrix}$, and the feedthrough matrix D is 0. Next we must use MATLAB to plot the velocity v of the car over 10 seconds. The state and output equations are

$$\begin{aligned} \begin{bmatrix} \dot{p} \\ \dot{v} \end{bmatrix} &= \begin{bmatrix} 0 & 1 \\ 0 & -b \end{bmatrix} \begin{bmatrix} p \\ v \end{bmatrix} + \begin{bmatrix} 0 & 0 \\ \frac{1}{m} & -g \end{bmatrix} \begin{bmatrix} F \\ \sin\theta \end{bmatrix} \\ y &= \begin{bmatrix} 0 & 1 \end{bmatrix} \begin{bmatrix} p \\ v \end{bmatrix} + \begin{bmatrix} 0 & 0 \end{bmatrix} \begin{bmatrix} F \\ \sin\theta \end{bmatrix} \end{aligned}$$

The solution is provided in the MATLAB code below. The MATLAB code results in Figure 1.3 which plots the velocity of the car over the 10 s simulation time.

MATLAB code for Example 1.5.2

1.5 The Output Equation $y=Cx+Du$

```
close all
clear
dt = 0.01; %(s) Time step
t = 0:dt:10; %(s) Time vector
x = [0;18]; %([m;m/s]) Initial position and velocity
m = 1000; %(kg) Mass
g = 9.8; %(m/s^2) Gravity
b = 0.01; %(1/s) Rolling resistance
theta = pi/30*(t-5); %(rad) Road grade angle
F = 900-50*t; %(N) Force
N = length(t); %Number of time-steps
y = zeros(1,N); %(m/s) allocate memory to store the output
A = [0,1;0,-b]; %State matrix for the car's linear dynamic model
B = [0,0;1/m,-g]; %Input matrix for the car's linear dynamic model
Fd = expm([A*dt,B*dt;zeros(2,4)]); %solution matrix
Ad = Fd(1:2, 1:2); %State-transition matrix
Bd = Fd(1:2,3:4); %Input-transition matrix
C = [0,1]; %Output matrix
D = [0,0]; %Feedthrough matrix
u = [F;sin(theta)]; %Input
for ii = 1:N
    y(ii) = C * x + D * u(:,ii); %output equation
    x = Ad * x + Bd * u(:,ii); %state-equation solution
end
figure
plot(t,y) %Plot the velocity versus time trajectory
xlabel('Time (s)') %Label the x-axis
ylabel('Velocity (m/s)') %Label the y-axis
title('Velocity of the Car') %Give the graph a title
grid on %Turn on the grid lines
```

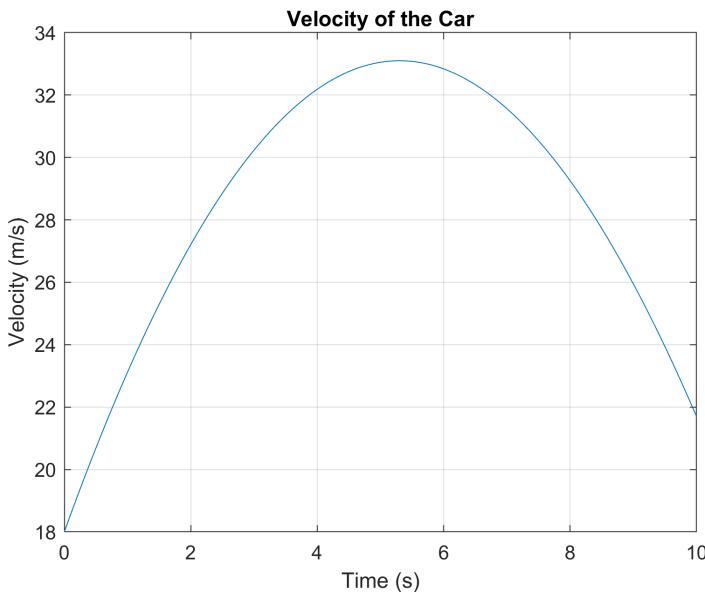


Figure 1.3 Plot of the car's velocity

1.6 Eigenvalues of A in $\dot{x} = Ax + Bu$ and Stability

The **poles** of a state-space system are the eigenvalues of the state matrix A . Each **eigenvalue** λ of a matrix A in $\dot{x} = Ax + Bu$ is a root or zero of the **characteristic equation**:

$$\det(\lambda I - A) = 0 \quad (1.41)$$

where \det is the determinant operator. An $n \times n$ square matrix has n eigenvalues. The following example shows how to calculate the poles of a two dimensional state-space equation.

Poles of a State Equation with Two States

Example 1.6.1. Eigenvalues of a 2x2 matrix

Find the poles of the state-space equation

$$\dot{x} = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} x + \begin{bmatrix} b_1 \\ b_2 \end{bmatrix} u$$

Solution: The poles of the state-space equation are the eigenvalues of the A matrix. In this case, the A matrix is

$$A = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix}$$

1.6 Eigenvalues of A in $\dot{x} = Ax + Bu$ and Stability

A 2×2 matrix has two eigenvalues λ_1 and λ_2 . To find them, we need to find the complex roots of the characteristic equation $\det(\lambda I - A) = 0$.

$$\begin{aligned}\det\left(\begin{bmatrix}\lambda & 0 \\ 0 & \lambda\end{bmatrix} - \begin{bmatrix}a_{11} & a_{12} \\ a_{21} & a_{22}\end{bmatrix}\right) &= \det\left(\begin{bmatrix}\lambda - a_{11} & -a_{12} \\ -a_{21} & \lambda - a_{22}\end{bmatrix}\right) \\ &= (\lambda - a_{11})(\lambda - a_{22}) - a_{21}a_{12} \\ &= \lambda^2 - (a_{11} + a_{22})\lambda + a_{11}a_{22} - a_{21}a_{12} \\ &= 0\end{aligned}$$

We use the quadratic formula to solve for the two roots:

$$\lambda_1 = \frac{(a_{11} + a_{22}) + \sqrt{(a_{11} + a_{22})^2 - 4(a_{11}a_{22} - a_{21}a_{12})}}{2} \quad (1.42)$$

$$\lambda_2 = \frac{(a_{11} + a_{22}) - \sqrt{(a_{11} + a_{22})^2 - 4(a_{11}a_{22} - a_{21}a_{12})}}{2} \quad (1.43)$$

The eigenvalues λ_1 and λ_2 are the poles of the system, and we have accomplished the objective of this example problem.

The eigenvalues of the A matrix determine the stability of a state-space equation. Eigenvalues are complex numbers with both real and imaginary parts. Although the imaginary parts play a role in the system response, only the real parts of the eigenvalues determine the stability. If the real part of an eigenvalue is negative, the eigenvalue is stable. If the real part is zero, the eigenvalue is marginally stable. If the real part is positive, the eigenvalue is unstable. The stability of the system is determined by the least stable eigenvalue. For example, if any eigenvalue is unstable, the whole system is **unstable**. If the largest real part of any eigenvalue is zero, the whole system is **marginally stable**. All eigenvalues must have negative real parts for the system to be **stable**. In dynamic systems modeled by state-space equations, **poles** are another name for the eigenvalues of the A matrix.

Poles, or the eigenvalues, govern how the dynamic system will respond to nonzero initial conditions. Because poles can be complex numbers, we plot pole locations on the complex plane. Figure 1.4 shows how the pole location affects how the dynamic system will respond to a step input. The following can be observed from this figure:

- When the pole is located in the right-hand plane, or when the real part of the pole is positive, then the response will grow in amplitude exponentially. These are called unstable pole locations.
- When the pole is on the vertical, or imaginary, axis then the real value of the pole is zero and the response does not increase or decrease in amplitude. These are called marginally stable pole locations.
- When the pole is in the left-hand plane, or when the real part of the pole is negative, then the transient response will decay or decrease in amplitude exponentially. These are called stable pole locations.

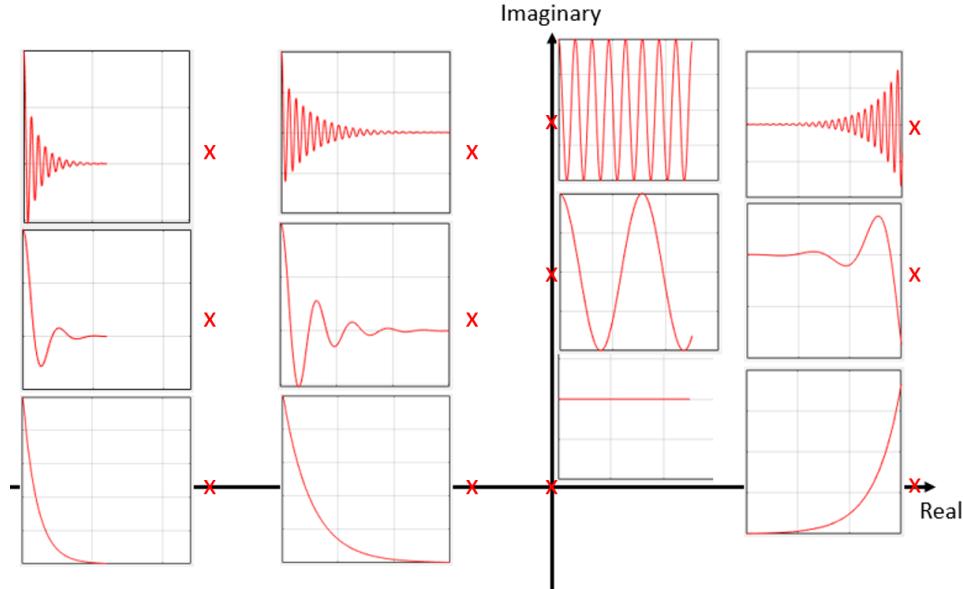


Figure 1.4 Transient response based on pole locations in the complex plane. (For systems with non-minimum phase zeros, the stability criteria apply, but the transient responses may be different from the ones in these graphs.)

- When the pole is on the horizontal, or real, axis then the imaginary value of the pole is zero and the response does not oscillate.
- The farther vertically the pole is from the real axis, the greater the frequency of oscillation.
- The farther horizontally the pole is from the imaginary axis, the smaller the time constant or the faster the time to reach steady state.

1.6.1 Eigenvalues and Transient Time

The eigenvalues of A in $\dot{x} = Ax + Bu$ reveal the transient time of *stable* systems. **Transient time** is the amount of time it takes for transients (effects of initial conditions) to wear off. The eigenvalue that is nearest to the imaginary axis is referred to as the **slowest eigenvalue** λ_{slowest} . The real part of the slowest eigenvalue determines the transient time of stable systems.

Transient time is characterized by the **time constant** τ . The time constant τ of a stable system is the negative reciprocal of the real part of the slowest eigenvalue:

$$\tau = \frac{-1}{\text{Real}(\lambda_{\text{slowest}})} \quad (1.44)$$

As a rule of thumb, the transient time is approximately four time constants:

$$\text{Transient Time} \approx 4\tau \quad (1.45)$$

In the most mathematically rigorous definition, steady-state is when the state x is constant and will not change in the future. In that case, the state-derivative \dot{x} is also zero indefinitely. However, dynamic systems and controls textbooks often use a less rigorous definition: **steady-state** is when the effects of

1.6 Eigenvalues of A in $\dot{x} = Ax + Bu$ and Stability

initial conditions have died away sufficiently that their effect can be neglected. We will adopt this less rigorous definition. Doing so allows us to have steady-state conditions for time-varying inputs such as sinusoidal forcing functions.

Knowledge of the time constant τ can be used to determine how long the transients of a stable system will take to die away. It takes three time-constants for a first-order system (a system with only one state variable) to change from an initial value to 95% of the steady-state value. If a first-order system is given a constant input, like in the examples shown above, then the time it will take the output to go a given percentage from its input to its steady-state value is shown in Table 1.1. This behavior is also shown in the graph of Figure 1.5.

Table 1.1 Time constants and percent attenuation of the transient response

1τ	63.2%
2τ	86.5%
3τ	95.0%
4τ	98.2%
5τ	99.3%
6τ	99.8%

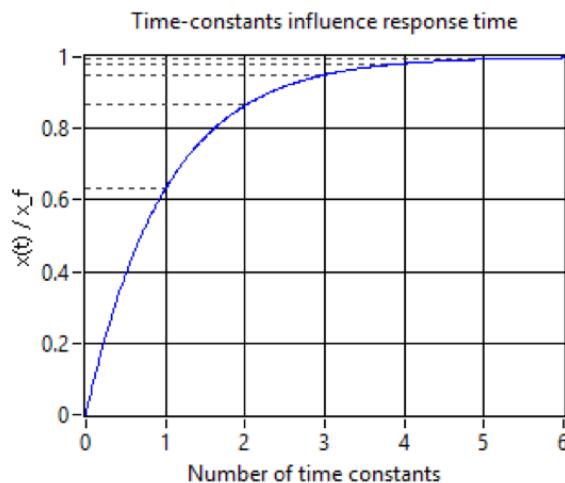


Figure 1.5 Time-constants influence response time of a stable ODE

Linear systems can be written as combinations of first and second-order systems. The following two subsections find the eigenvalues and response of first and second order systems respectively.

1.6.2 Eigenvalues of first order ODEs

A first order Ordinary Differential Equation (ODE) can be written as follows

$$\dot{x} = -\frac{1}{\tau}x + bu$$

where τ is the time-constant if it is positive, and b and u are real-valued scalars. If u and b are constant, the state x will eventually reach a final value x_f of

$$x_f = \tau bu$$

after the effect of initial conditions have worn off. The eigenvalue λ of a first order ODE is the negative reciprocal of τ :

$$\lambda = -\frac{1}{\tau}$$

1.6.3 Eigenvalues, time constants, and frequency response of 2nd order ODEs

A second order linear-time-invariant ODE can be written as

$$\ddot{x} + 2\zeta\omega_n\dot{x} + \omega_n^2x = f(t) \quad (1.46)$$

where ζ and ω_n are constants, and the forcing function $f(t)$ does not depend on the variable x , only time t . If it is stable, ζ is the damping ratio and ω_n is the natural frequency.

We define the state variable x_1 to be x and its derivative \dot{x} as x_2 , i.e., $\dot{x}_1 = x_2$. With these definitions, we can rewrite Eq. (1.46) with the following set of equations:

$$\begin{aligned}\dot{x}_1 &= x_2 \\ \dot{x}_2 &= -2\zeta\omega_n x_2 - \omega_n^2 x_1 + f(t)\end{aligned}$$

Which, in state-equation form, becomes

$$\begin{bmatrix} \dot{x}_1 \\ \dot{x}_2 \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ -\omega_n^2 & -2\zeta\omega_n \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} + \begin{bmatrix} 0 \\ 1 \end{bmatrix} f(t)$$

The roots of the characteristic polynomials are the eigenvalues (λ_1 and λ_2) of the ODE. Using Example 1.6.1, the eigenvalues are

$$\lambda_{1,2} = -\zeta\omega_n \pm j\omega_d$$

where $j = \sqrt{-1}$ is the imaginary number, and

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

ω_d is the damped natural frequency only when the ODE is stable, i.e., if $\omega_n > 0$ and $\zeta > 0$. Also, if the ODE is stable and $0 < \zeta < 1$, the time constant τ is the negative reciprocal of the real part of the eigenvalues:

$$\tau = \frac{1}{\zeta\omega_n}$$

1.7 Expressing Differential Equations in State-Space Form

If the ODE is stable and $\zeta > 1$, the time constant τ is the negative reciprocal of the less negative eigenvalue

$$\tau = \frac{1}{\zeta \omega_n - \omega_n \sqrt{\zeta^2 - 1}}$$

The time constant is only defined when the ODE is stable.

1.7 Expressing Differential Equations in State-Space Form

This section provides examples of how to express Ordinary Differential Equations (ODEs) in linear state-space form.

1.7.1 Expressing First Order ODEs in State-Space Form

The following example shows how to write first order ODEs in state-space form. The example uses a resistor-capacitor (RC) circuit. Dynamic modeling RC circuits is taught in Section 6.3 and Section 6.4.

Modeling an RC Circuit

Example 1.7.1. Modeling an RC Circuit

Suppose we have a series resistor-capacitor (RC) circuit. Find the state-space model for the circuit.

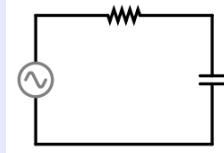


Figure 1.6 Series RC Circuit

Solution: We use Kirchoff's law to sum the voltage drops around the circuit. The current of a capacitor is $I = C\dot{V}_C$. The dynamic model of the circuit is

$$V_R + V_C = V_{in}$$
$$RC\dot{V}_C + V_C = V_{in}$$

We can arrange this equation into state-space form using V_C as the state variable:

$$\dot{V}_C = -\frac{1}{RC}V_C + \frac{1}{RC}V_{in}$$

thus:

The state-transition matrix is the scalar: $A = -\frac{1}{RC}$

The input matrix is also a scalar: $B = \frac{1}{RC}$

The input variable is: $u = V_{in}$

If the capacitor voltage was the desired output, then the output equation is simply:

$$y = V_C$$

thus:

Output Variable: $y = V_C$

Output Matrix: $C = 1$

Direct Transition Matrix: $D = 0$

1.7.2 Expressing Multiple First Order ODEs in State-Space Form

We can combine multiple first order ordinary differential equations into state-space form.

Two First-Order ODEs in State-Space Form

Example 1.7.2. Modeling two first order ODEs into state-space form

Express the following coupled first order ODEs in state-space form:

$$\begin{aligned} 3\dot{x} + 4(x - y) &= f(t) \\ 2\dot{y} - x &= 0 \end{aligned}$$

Solution: We express the ODEs in state-space form by solving each equation for the derivative term.

$$\begin{aligned} \dot{x} &= -\frac{4}{3}x + \frac{4}{3}y + \frac{1}{3}f(t) \\ \dot{y} &= \frac{1}{2}x \end{aligned}$$

and then arranging them in matrix form:

$$\begin{bmatrix} \dot{x} \\ \dot{y} \end{bmatrix} = \begin{bmatrix} -\frac{4}{3} & \frac{4}{3} \\ \frac{1}{2} & 0 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} \frac{1}{3} \\ 0 \end{bmatrix} f(t)$$

thus:

State Vector is $\begin{bmatrix} x \\ y \end{bmatrix}$

State-transition matrix: $A = \begin{bmatrix} -\frac{4}{3} & \frac{4}{3} \\ \frac{1}{2} & 0 \end{bmatrix}$

Input matrix is: $B = \begin{bmatrix} \frac{1}{3} \\ 0 \end{bmatrix}$

Input Variable: $u = f(t)$

If we were to output the value of y , then the output equation would be $y = y$ or:

$$y = [0 \quad 1] \begin{bmatrix} x \\ y \end{bmatrix}$$

1.7 Expressing Differential Equations in State-Space Form

thus:

Output Variable is y

$$\text{Output Matrix: } C = \begin{bmatrix} 0 & 1 \end{bmatrix}$$

$$\text{Direct Transition Matrix: } D = \begin{bmatrix} 0 \end{bmatrix}$$

1.7.3 Expressing Higher-Order ODEs in State-Space Form

In order to express an ODE that is higher than first-order into state-space form, we have to convert the higher order equation into systems of first-order ODEs first, because the state-space form only has single derivatives in the state equation. If we had an n -order ODE, then we will need to convert it into n number of first order ODEs.

The steps involved to convert an n^{th} -order ODE into n first order ODEs is to do the following:

1. Solve for the highest derivative
2. Replace each derivative on the right-hand side of the equation with a new variable
3. Replace the highest derivative on the left-hand side with a first order derivative using one of the new variables
4. Combine the equations into state-space form

Modeling a 2nd Order ODE in State-Space Form

Example 1.7.3. Model A Second Order ODE in State-Space Form

Express the given second-order ODE of a spring-mass-damper system:

$$m\ddot{x} + b\dot{x} + kx = f(t)$$

into state-space

Solution: Applying the above steps to the second order spring-mass-damper equation:

1. Solve for the highest derivative:

$$\ddot{x} = \frac{1}{m}(-b\dot{x} - kx + f(t))$$

2. Replace \dot{x} on the right-hand side with $\dot{x} = v$ to get:

$$\ddot{x} = \frac{1}{m}(-bv - kx + f(t))$$

3. Replace the \ddot{x} on the left-hand side \dot{v} since $v = \dot{x}$

$$\dot{v} = \frac{1}{m}(-bv - kx + f(t))$$

4. Combine the equations:

$$\begin{aligned}\dot{x} &= v \\ \dot{v} &= \frac{1}{m}(-bv - kx + f(t))\end{aligned}$$

into state-space form to get:

$$\begin{bmatrix} \dot{x} \\ \dot{v} \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ -\frac{k}{m} & -\frac{b}{m} \end{bmatrix} \begin{bmatrix} x \\ v \end{bmatrix} + \begin{bmatrix} 0 \\ \frac{1}{m} \end{bmatrix} f(t)$$

and if we were interested in outputting the position then the output equation would be

$$y = \begin{bmatrix} 1 & 0 \end{bmatrix} \begin{bmatrix} x \\ v \end{bmatrix}$$

1.7.4 ODEs with Time-Derivatives of Forcing Functions

The inputs to the ODEs in the previous sections had forcing functions $u(t)$ without any time derivatives ($\dot{u}(t)$, $\ddot{u}(t)$, $\ddot{u}(t)$, etc. In general, an ODE could have the form

$$x^{(n)} + a_{n-1}x^{(n-1)} + \cdots + a_2\ddot{x} + a_1\dot{x} + a_0x = b_m u^{(m)} + b_{m-1}u^{(m-1)} + \cdots + b_1\dot{u} + b_0u \quad (1.47)$$

where \dot{u} , \ddot{u} , ..., $u^{(m)}$ are the time-derivatives of the forcing function u , and \dot{x} , \ddot{x} , ..., $x^{(n)}$ are the time-derivatives of the dependent variable x .

In general, the approach to numerically solve such a differential equation using state-space is to first find the Laplace transfer function $\frac{X}{U}$ and then use the technique presented in Section 2.2.3 to convert it back to state-space form; however, that approach ignores the initial conditions.

A differential equation with n number of time-derivatives of the dependent variable will need $n - 1$ initial conditions: $x(0) = x_0$, ..., $x^{(n-1)}(0) = x_0^{(n-1)}$. An input with m number of time-derivatives will need $m - 1$ initial conditions: $u(0) = u_0$, ..., $u^{(m-1)}(0) = u_0^{(m-1)}$. To include these initial conditions in the solution would prove to be a more accurate result, but there isn't a place for input derivatives in the general state-space form:

$$\begin{aligned}\dot{x} &= Ax + Bu \\ y &= Cx + Du\end{aligned}$$

One simple method to include input derivatives is to add another state matrix (E) to the state equation

$$\dot{x} = Ax + Bu + Eu$$

and then define a new state vector q to be

$$q = x - Eu$$

1.7 Expressing Differential Equations in State-Space Form

Taking the derivative of the new state vector and simplifying gives

$$\begin{aligned}\dot{q} &= \dot{x} - E\dot{u} \\ &= Ax + Bu + E\dot{u} - E\dot{u} \\ &= A(q + Eu) + Bu \\ &= Aq + (B + AE)u \\ &= Aq + B_q u\end{aligned}$$

The initial conditions in x and u are then included into the new state vector q using:

$$q_0 = x_0 - Eu_0$$

and the solution to the differential equation is determined by solving the equations:

$$\begin{aligned}\dot{q} &= Aq + B_q u \\ x &= q + Eu \\ y &= Cx + Du\end{aligned}$$

where $B_q = B + AE$. Note that the solution method can only be applied when the highest order of derivatives on the input is less than the highest order of derivative on the dependent variable ($m \leq n$).

If we substitute $x = q + Eu$ into the output equation $y = Cx + Du$, the state-space and output equations can be written as

$$\dot{q} = Aq + B_q u \quad (1.48)$$

$$y = Cq + D_q u \quad (1.49)$$

where $D_q = D + CE$. The initial condition is

$$q_0 = x_0 - Eu_0 \quad (1.50)$$

The following example demonstrates this process for a first order ODE with an input derivative.

First Order ODE with an Input Derivative

Example 1.7.4. First Order ODE with an Input Derivative

Develop the discretized state-space equations to solve the ODE

$$\tau \dot{x} + x = b_1 \dot{u} + b_0 u$$

for $x(t)$ using the initial conditions

$$x(0) = x_0$$

$$u(0) = u_0$$

Solution: To solve the ODE, we first arrange it into the expanded state space form:

$$\begin{aligned}\dot{x} &= Ax + Bu + E\dot{u} \\ &= -\frac{1}{\tau}x + \frac{b_0}{\tau}u + \frac{b_1}{\tau}\dot{u}\end{aligned}$$

thus

$$\begin{aligned}A &= -\frac{1}{\tau} \\ B &= \frac{b_0}{\tau} \\ E &= \frac{b_1}{\tau}\end{aligned}$$

and

$$\begin{aligned}B_q &= B + AE = \frac{b_0}{\tau} - \frac{b_1}{\tau^2} \\ C &= 1 \\ D &= 0\end{aligned}$$

and the state-space form becomes:

$$\begin{aligned}\dot{q} &= -\frac{1}{\tau}q + \left(\frac{b_0}{\tau} - \frac{b_1}{\tau^2}\right)u \\ x &= q + \frac{b_1}{\tau}u \\ y &= x\end{aligned}$$

The discretized matrices are:

$$\begin{aligned}A_d &= e^{A\Delta t} \\ &= e^{-\Delta t/\tau} \\ B_{qd} &= A^{-1}(e^{A\Delta t} - 1)(B + AE) \\ &= (-\tau)(e^{-\Delta t/\tau} - 1)\left(\frac{b_0}{\tau} - \frac{b_1}{\tau^2}\right) \\ &= (1 - e^{-\Delta t/\tau})\left(b_0 - \frac{b_1}{\tau}\right)\end{aligned}$$

The solution $x(t)$ is then solved numerically by initializing the state vector:

$$q_0 = x_0 - \frac{b_1}{\tau}u_0$$

and iterating through time using the discretized matrices:

$$\begin{aligned}y_k &= x_k \\ x_k &= q_k + \frac{b_1}{\tau}u_k \\ q_{k+1} &= e^{-\Delta t/\tau}q_k + (1 - e^{-\Delta t/\tau})\left(b_0 - \frac{b_1}{\tau}\right)u_k\end{aligned}$$

1.8 Numerical Solutions of LTI State-Space Systems

1.8 Numerical Solutions of LTI State-Space Systems

Table 1.2 State- and Input-Transition Matrices for Numerical Methods Ordered from Most to Least Accurate

Method	State-Transition Matrix A_d	Input-Transition Matrix B_d
Piecewise Exact Method	$A_d = \text{expm}\{A\Delta t\} = \sum_{k=0}^{\infty} \frac{1}{k!} (A\Delta t)^k$	$\begin{bmatrix} A_d & B_d \\ \mathbf{0} & I \end{bmatrix} = \text{expm}\left\{\begin{bmatrix} A\Delta t & B\Delta t \\ \mathbf{0} & \mathbf{0} \end{bmatrix}\right\}$
4th Order Runge-Kutta	$A_d = \sum_{k=0}^4 \frac{1}{k!} (A\Delta t)^k$	$B_d = \left(\sum_{k=0}^3 \frac{1}{(k+1)!} (A\Delta t)^k\right) B\Delta t$
Tustin's Method	$A_d = \left(I - \frac{\Delta t}{2} A\right)^{-1} \left(I + \frac{\Delta t}{2} A\right)$	$B_d = \left(I - \frac{\Delta t}{2} A\right)^{-1} B\Delta t$
Backward Euler	$A_d = (I - A\Delta t)^{-1}$	$B_d = (I - A\Delta t)^{-1} B\Delta t$
Forward Euler	$A_d = I + A\Delta t$	$B_d = B\Delta t$

This section presents numerical solutions to linear-time-invariant state-space systems with time-varying inputs $u(t)$. There are many possible solutions, such as forward and backward Euler methods, Runge-Kutta methods, Tustin's method, etc. The most accurate method, however, is based on the exact solution presented in Section 1.3. It is exact for constant or piecewise constant inputs. The difference between these methods is entirely captured by the state-transition matrix A_d and input-transition matrix B_d . These differences are listed in Table 1.2.

To use these numerical methods, continuous-time-varying inputs can be sampled at discrete time-intervals and treated as piecewise constant. If the sampling period Δt is small enough, these methods provide accurate approximations to the exact solution. As $\Delta t \rightarrow 0$ the solution theoretically becomes exact. Practically, however, exactness is impossible because of computational limitations – computers cannot represent floating point numbers with exactness.

All the methods of Table 1.2 are discrete-time solutions. However, of all the numerical methods of Table 1.2, only the Piecewise Exact Method has the same stability characteristics in discrete-time and in continuous-time. In other words, only the piecewise exact method guarantees that continuous-time stable eigenvalues will map to discrete-time stable eigenvalues, continuous-time marginally stable eigenvalues map to discrete-time marginally stable eigenvalues, and continuous-time unstable eigenvalues map to discrete-time unstable eigenvalues. Discrete-time stability will be discussed in Section 3.1.7.

The following example illustrates how a time-varying input can be sampled into discrete stepwise

constant values. It then solves a state-space system in MATLAB using each of the methods listed in Table 1.2.

Comparing Numerical Solutions to State-Space Systems

Example 1.8.1. Comparing Numerical Solutions to State-Space Systems

Use each numerical method listed in Table 1.2 to solve the following state-space system:

$$y = \begin{bmatrix} 1 & 0 \end{bmatrix} x + 0u$$

$$\dot{x} = \begin{bmatrix} 0 & 0 \\ -0.5 & -0.2 \end{bmatrix} x + \begin{bmatrix} 0 \\ 1 \end{bmatrix} u$$

Numerically solve the state-space equations over a period of $t = 6$ s. Use a sampling period of $\Delta t = 0.3$ s. Compare the piecewise constant input to the true time-varying input:

$$u = \frac{1}{2}t$$

Use the initial condition

$$x(0) = \begin{bmatrix} 0 \\ 2 \end{bmatrix}$$

Solution: The MATLAB code below numerically approximates the solution to the state-equations using each of the methods in Table 1.2. It also creates the graph of Figure 1.7 which compares the true time-varying input u with its piecewise constant approximation.

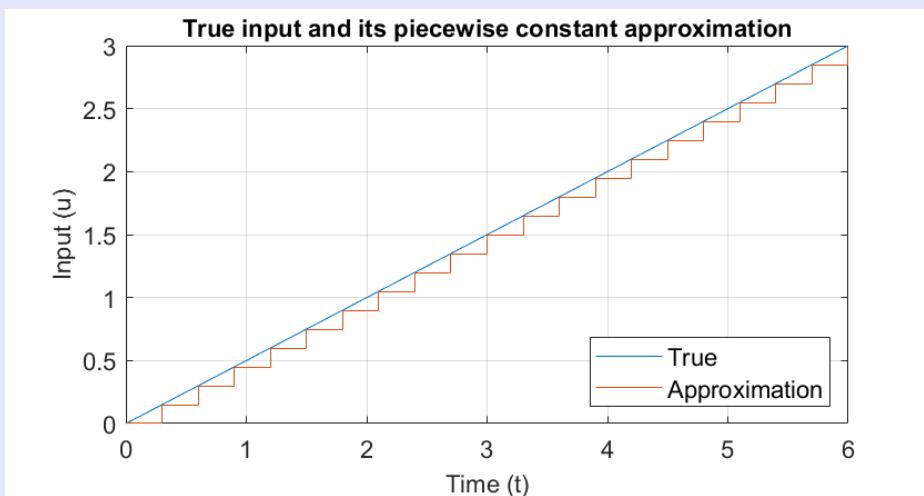


Figure 1.7 The true time-varying input is converted to a piecewise constant input

```
close all %Close all open figures
```

1.8 Numerical Solutions of LTI State-Space Systems

```
clear all %Clear all variables and functions
clc %Clear the command window

% The state-space system
A = [0,1;-0.5,-0.2];
B = [0;1];
C = [1,0]; %The position is the first state x(1) = [1,0]*x
D = 0;

% Discrete timestep
dt = 0.3; %(s)

% Time vector 0 to 6 s
t = 0:dt:6; %(s)

% The stepwise constant input acceleration (m/s^2)
u = 1/2*t; %(m/s^2) vector of stepwise constant input acceleration values

%Discrete-time state-transition and input-transition matrices
%Exact method
Fd = expm([A*dt,B*dt;zeros(1,3)]);
AdExact = Fd(1:2,1:2);
BdExact = Fd(1:2,3);
%Runge-Kutta method
I = eye(2);
AdRK = I+A*dt+(A*dt)^2/2+(A*dt)^3/factorial(3)+(A*dt)^4/factorial(4);
BdRK = dt*(I+A*dt/2+(A*dt)^2/factorial(3)+(A*dt)^3/factorial(4))*B;
%Tustin's method
AdTustin = (I-dt/2*A)\(I+dt/2*A);
BdTustin = (I-dt/2*A)\B*dt;
%Backward Euler
AdBE = (I-A*dt)^(-1);
BdBE = (I-A*dt)^(-1)*B*dt;
%Forward Euler
AdFE = I+A*dt;
BdFE = B*dt;

%Set the initial condition for each method
xExact = [0;2]; %0 m initial position and 2 m/s initial velocity
xRK = xExact;
xTustin = xExact;
xBE = xExact;
xFE = xExact;
xTrue = xExact;

%Allocate memory to store the output (position) trajectory for each method
N = length(t); %Number of timesteps
yExact = zeros(1,N);
yRK = zeros(1,N);
yTustin = zeros(1,N);
xBE = zeros(1,N);
xFE = zeros(1,N);
```

```
%Run the simulation
for ii = 1:N
    %Solve the output equation and store the result
    %The output is the first state, i.e., the distance (m) traveled
    yExact(ii) = C*xExact+D*u(ii);
    yRK(ii) = C*xRK+D*u(ii);
    yTustin(ii) = C*xTustin+D*u(ii);
    yBE(ii) = C*xBE+D*u(ii);
    yFE(ii) = C*xFE+D*u(ii);
    %Solve the state equation using each method
    xExact = AdExact*xExact+BdExact*u(ii);
    xRK = AdRK*xRK+BdRK*u(ii);
    xTustin = AdTustin*xTustin+BdTustin*u(ii);
    xBE = AdBE*xBE+BdBEBE*u(ii);
    xFE = AdFE*xFE+BdFE*u(ii);
end

%Now take much smaller timesteps and run the simulation again to get
% a nearly perfect solution
% Discrete timestep
dT = 0.0001; %(s)

% Time vector 0 to 6 s
tT = 0:dT:6; %(s)

%Exact method with a much smaller time-step
FdT = expm([A*dT,B*dT;zeros(1,3)]);
AdT = FdT(1:2,1:2);
BdT = FdT(1:2,3);

% The stepwise constant input acceleration (m/s^2)
uT = 1/2*tT; %(m/s^2) vector of stepwise constant input acceleration values
NT = length(tT);
yTrue = zeros(1,NT);
for ii = 1:NT
    yTrue(ii) = C*xTrue+D*uT(ii);
    xTrue = AdT*xTrue+BdT*uT(ii);
end

%Create a graph comparing the performance of each numerical method
figure %create a figure
%Plot the results of each numerical method on the figure
plot(t,yExact,'.-',t,yRK,'^-',t,yTustin,'-s',t,yBE,'-o',t,yFE,'-d')
hold on %Get a handle to the figure to plot additional signals on it
plot(tT,yTrue,'LineWidth',3) %Plot the best solution with a fat line
xlabel('Time (t)') %Label the x-axis
ylabel('Output Trajectory (y)') %Label the y-axis
grid on %Turn on the grid
title('Comparison of Numerical Methods') %Write a title
legend('Piecewise Exact','Runge-Kutta','Tustin','Backward Euler', ...
'Forward Euler','True','Location','southeast') %Create a legend

%Create a graph comparing the true input u with the stepwise constant input
```

1.8 Numerical Solutions of LTI State-Space Systems

```

figure %Create a new figure
plot(t,u) %Graph the true input
hold on %Get a handle to the figure to plot additional signals on it
stairs(t,u) %Graph the piecewise constant input
grid on %Turn on the grid
xlabel('Time (t)') %Label the x-axis
ylabel('Input (u)') %Label the y-axis
title('True input and its piecewise constant approximation') %Title it
legend('True','Approximation','Location','southeast') %Create the legend

```

The MATLAB code produces the graph of Figure 1.8. Approximating the time-varying input $u = \frac{1}{2}t$ as piecewise constant (see Figure 1.7) causes even the Exact Method to deviate from the true solution. However, the Exact Method, Runge-Kutta, and Tustin methods produce nearly the same result as each other. Comparing the entire range, the Backward and Forward Euler methods are less accurate.

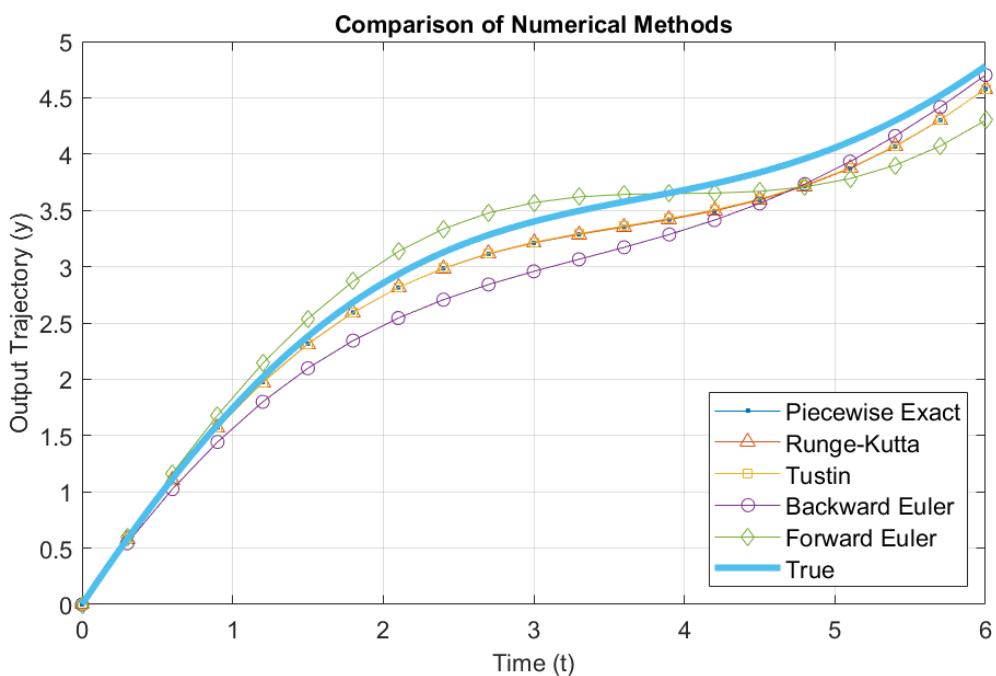


Figure 1.8 Comparison of numerical solutions to the true solution. The inaccuracy in the piecewise exact solution results from sampling the input at discrete time intervals. All other methods are approximations.

Chapter 2

Laplace Transfer Function Modeling

Contents

2.1	Laplace Transforms	40
2.1.1	Proofs of Selected Laplace Transform Properties	41
2.2	Modeling Dynamic Systems using Laplace Transfer Functions	43
2.2.1	Representing an ODE as a Transfer Function	43
2.2.2	Convert State-Space to Transfer Functions	44
2.2.3	Convert Transfer Functions to State-Space	46
2.3	Coordinate Transformations in State-Space Systems	50
2.3.1	Coordinate Transformation to Controllable Canonical Form	51
2.3.2	Coordinate Transforms to Observable Canonical Form	55
2.3.3	Coordinate Transforms do not Affect Transfer Functions	57
2.4	Poles and Zeros of Laplace Transfer Functions	58
2.4.1	Poles of Transfer Functions	58
2.4.2	Zeros of Transfer Functions	59
2.4.3	Pole-Zero Cancellation and Stability	63
2.5	Zero-Pole-Gain Form of a Transfer Function	65
2.6	Final Value Theorem	66
2.6.1	Final Value for Constant Inputs	67
2.6.2	Steady-State Gain of Stable Transfer Functions	67
2.7	Transfer Functions as Complex Numbers	68
2.7.1	Review of Complex Numbers	68
2.7.2	Conversions between Complex Number Forms	70
2.7.3	Mathematical Operations of Complex Numbers	71
2.7.4	Magnitude and Phase Angle of Complex Fractions	71
2.7.5	Magnitude and Phase of a Transfer Function	72
2.8	Frequency Response Plots and Bode Plots	73
2.8.1	Frequency Response Plots	73
2.8.2	Bode Plots	75

2.8.3	Drawing Bode Magnitude Plots by Hand	78
2.8.4	Drawing Bode Phase Plots by Hand	81
2.8.5	Natural Frequency and Damping Ratios on Bode Plots	82
2.9	Nyquist Plots of Transfer Functions	84

Laplace transfer functions can model dynamic systems. Transfer functions are especially useful in control block diagrams and in analyzing the frequency domain behavior of dynamic system. Therefore, the concepts of this chapter are valuable to system identification, feedback control, and designing filters for signal processing.

2.1 Laplace Transforms

This book assumes that the reader has a background in differential equations, including Laplace transforms. It will, however, provide a review of Laplace transforms and their properties that are relevant to this book. Perhaps you have seen a table or list of properties of Laplace transforms online or in a differential equations textbook. A truncated list of relevant properties is provided in Table 2.1.

Table 2.1 Selected properties of Laplace transforms

Property	Time-domain $x(t) = \mathcal{L}^{-1}\{X(s)\}$	Laplace-domain $X(s) = \mathcal{L}\{x(t)\}$
Linearity	$ax + bx$	$aX + bX$
Time-derivative	\dot{x}	$sX - x(0)$
Second derivative	\ddot{x}	$s^2X - sx(0) - \dot{x}(0)$
n^{th} derivative	$x^{(n)}$	$s^nX - \sum_{k=1}^n s^{n-k}x^{k-1}(0)$
Time-integral	$\int x dt$	$\frac{1}{s}X$
Second integral	$\int \int x d^2t$	$\frac{1}{s^2}X$
n^{th} integral	$\int \dots \int x d^n t$	$\frac{1}{s^n}X$

A time-domain signal, $x(t)$, is a function of the independent time variable t . Similarly, a Laplace-domain signal, $X(s)$, is a function of the independent Laplace variable s . The independent variable s can be considered a complex frequency, having units of, for example, rad/s. This property is used in Section 2.8 to create Bode plots and frequency response plots. Mathematically, the time-domain signal $x(t)$ is converted to a Laplace-domain signal $X(s)$ using the Laplace transform:

$$X(s) = \int_0^\infty x(t)e^{-st} dt \quad (2.1)$$

To simplify notation, this book follows the pattern of most textbooks. It drops the independent variables t and s and replaces the integral with \mathcal{L} . With this simplified notation, Eq. (2.1) is written as $X = \mathcal{L}\{x\}$. The inverse Laplace transform converts a Laplace-domain signal back to the time-domain. The simplified notation for the inverse Laplace transform is $x = \mathcal{L}^{-1}\{X\}$.

This book uses three properties of Laplace transforms frequently. The first property is linearity, *i.e.*, $\mathcal{L}\{ax + bx\} = aX + bX$ if a and b are scalar constants. The second property is the general derivative:

2.1 Laplace Transforms

$\mathcal{L}\{x^{(n)}\} = s^n X$ where initial conditions are assumed to be zero. The third property is the general time-domain integral: $\mathcal{L}\{\int \cdots \int x d^n t\} = \frac{1}{s^n} X$.

This book solves differential equations numerically and will not typically apply the inverse Laplace transform.

2.1.1 Proofs of Selected Laplace Transform Properties

This section provides proofs for some of the selected Laplace transform properties. The proof of the linearity property is a direct result of the linearity property of integration, and hence needs no additional proof.

The proof of the time-derivative property is derived below.

Proof of the Time-Derivative Property

Example 2.1.1. Proof that $\mathcal{L}\{\dot{x}\} = sX - x(0)$

Integration by parts, $\int u dv = uv - \int v du$, is used to prove the Laplace transform of the time-derivative $\mathcal{L}\{\dot{x}\} = sX - x(0)$. Let

$$u = e^{-st}$$

and

$$\begin{aligned} dv &= \dot{x} dt \\ &= \frac{dx}{dt} dt \\ &= dx \end{aligned}$$

Then

$$\begin{aligned} v &= \int_{x(0)}^{x(t)} dx \\ &= x(t) - x(0) \end{aligned}$$

and

$$du = -se^{-st} dt$$

Therefore,

$$\begin{aligned}
 \int_0^\infty \dot{x} e^{-st} dt &= uv - \int v du \\
 &= (e^{-st}(x(t) - x(0)))|_{t=0}^\infty - \int_0^\infty (x(t) - x(0))(-se^{-st}) dt \\
 &= \int_0^\infty (x(t) - x(0))(se^{-st}) dt \\
 &= \int_0^\infty x(t)(se^{-st}) dt - \int_0^\infty x(0)(se^{-st}) dt \\
 &= \int_0^\infty (sx(t))e^{-st} dt - sx(0) \int_0^\infty e^{-st} dt \\
 &= sX - x(0)
 \end{aligned}$$

The second equality above is due to substituting the terms from integration by parts. The third equality is because $\lim_{t \rightarrow \infty} e^{-st} = 0$ and $t = 0$ implies that $x(t) - x(0) = 0$. The last equality used integration by substitution: $\int_0^\infty e^{-st} dt = -\frac{1}{s} \int_0^\infty e^u du = \lim_{u \rightarrow -\infty} -\frac{1}{s}(e^u - e^0) = \frac{1}{s}$.

The proof of the second derivative is provided next.

Proof of the Second Derivative Property

Example 2.1.2. Proof that $\mathcal{L}\{\ddot{x}\} = s^2 X - sx(0) - \dot{x}(0)$

The proof begins by applying integration by parts, $\int u dv = uv - \int v du$, to the Laplace transform of the second derivative \ddot{x} . As before, the variable u is $u = e^{-st}$. The variable dv is $dv = \ddot{x} dt$. The derivative of u is $du = -se^{-st} dt$, and the integral of dv is $v = \dot{x}(t) - \dot{x}(0)$. Then, integration by parts is

$$\begin{aligned}
 \int_0^\infty \ddot{x} e^{-st} dt &= (\dot{x}(t) - \dot{x}(0))e^{-st}|_{t=0}^\infty - \int_0^\infty (\dot{x}(t) - \dot{x}(0))(-se^{-st}) dt \\
 &= \int_0^\infty s(\dot{x}(t) - \dot{x}(0))(e^{-st}) dt \\
 &= \int_0^\infty s\dot{x}(t)e^{-st} dt - s\dot{x}(0) \int_0^\infty e^{-st} dt \\
 &= s \int_0^\infty \dot{x}(t)e^{-st} dt - \dot{x}(0) \\
 &= s(sX - x(0)) - \dot{x}(0)
 \end{aligned}$$

The first equality is due to substituting the terms from integration by parts. The second equality is because $\lim_{t \rightarrow \infty} e^{-st} = 0$ and $t = 0$ implies that $\dot{x}(t) - \dot{x}(0) = 0$. The fourth equality used integration by substitution: $\int_0^\infty e^{-st} dt = -\frac{1}{s} \int_0^\infty e^u du = \lim_{u \rightarrow -\infty} -\frac{1}{s}(e^u - e^0) = \frac{1}{s}$. The last equality is a result of the time-derivative property of the Laplace transform, a proof of which was derived above.

By repeating a similar process, mathematical induction proves the n^{th} derivative property of the Laplace transform. The proofs of the integration properties require understanding of convolution integrals,

2.2 Modeling Dynamic Systems using Laplace Transfer Functions

which are beyond the scope of this textbook.

2.2 Modeling Dynamic Systems using Laplace Transfer Functions

All Laplace transfer functions in this textbook ignore initial conditions. There are a few justifications for this approximation. First, for stable systems, Section 1.6.1 showed that the effects of initial conditions wear off after about four time-constants (4τ). Second, for unstable systems that are stabilized using feedback control, this book necessarily assumes that the unstable system started from rest at equilibrium. Therefore, the initial conditions are all zero, and again can be neglected.

In any case, transfer functions ignore initial conditions. As a result, the derivative property becomes $\mathcal{L}\{\dot{x}\} = sX$, and the n^{th} derivative property is $\mathcal{L}\{x^{(n)}\} = s^n X$ for transfer functions.

2.2.1 Representing an ODE as a Transfer Function

A transfer function is the ratio of the output of an ordinary differential equation (ODE) to its input. The output is usually either the dependent variable $x(t)$ of the ODE or some function of the dependent variable $y = g(x(t))$. The input is the forcing function $u(t)$.

An n^{th} order ODE of x with the input forcing function u and its time-derivatives ($\dot{u}, \ddot{u}, \dots, u^{(n)}$) can be written as follows:

$$x^{(n)} + a_{n-1}x^{(n-1)} + \dots + a_2\ddot{x} + a_1\dot{x} + a_0x = b_nu^{(n)} + b_{n-1}u^{(n-1)} + \dots + b_1\dot{u} + b_0u$$

The initial conditions are

$$\begin{aligned} x(0) &= x_0 \\ &\vdots \\ x^{(n-1)}(0) &= x_0^{(n-1)} \end{aligned}$$

Transfer functions from the input forcing function u to the output x of the ODE ignore initial conditions. Taking the Laplace transform of both sides of the transfer function but ignoring initial conditions gives:

$$s^n X + a_{n-1}s^{n-1}X + \dots + a_0X = b_n s^n U + \dots + b_0U \quad (2.2)$$

Factoring X out of the left side and U from the right side of the equation:

$$X(s^n + a_{n-1}s^{n-1} + \dots + a_1s + a_0) = U(b_n s^n + \dots + b_1s + b_0) \quad (2.3)$$

The transfer function from the input forcing function u to the output x is the ratio of X/U :

$$\frac{X}{U} = \frac{b_n s^n + b_{n-1}s^{n-1} + \dots + b_1s + b_0}{s^n + a_{n-1}s^{n-1} + \dots + a_1s + a_0} \quad (2.4)$$

Notice that in this equation, the highest order term s^n in the denominator has a coefficient of 1. This is the preferred form of the transfer function for reasons that are explained in Section 2.2.3. Essentially, it makes the problem easier to solve using observable or controllable canonical forms. Another important observation is that initial conditions do not appear in the transfer function.

Transfer Function of a 2nd Order ODE

Example 2.2.1. Transfer Function of a 2nd Order ODE

Determine the transfer function of the following simple second-order ODE:

$$\ddot{x} + a_1 \dot{x} + a_0 x = b_2 \ddot{u} + b_1 \dot{u} + b_0 u$$

Solution: Following the same process of Eq. (2.2) through Eq. (2.4) results in the transfer function:

$$\frac{X(s)}{U(s)} = \frac{b_2 s^2 + b_1 s + b_0}{s^2 + a_1 s + a_0}$$

The same result can be reached by inspection of Eq. (2.4) by setting $a_2 = 1$ and all higher-order coefficients (b_3, b_4, \dots and a_3, a_4, \dots) to zero.

The following example calculates a transfer function for a mass-spring-damper system.

Transfer Function of a Mass-Spring-Damper System

Example 2.2.2. Transfer Function of a Mass-Spring-Damper System

If the ODE for a mass-spring-damper system is

$$m\ddot{x} + b\dot{x} + kx = f(t)$$

what is its transfer function?

Solution: By inspection of Eq. (2.4), we set $a_2 = m$, $a_1 = b$, $a_0 = k$, and $b_0 = 1$. All other coefficients are zero. The transfer function is the ratio of the output $X(s)$ to the input forcing function $F(s)$:

$$TF = \frac{X(s)}{F(s)} = \frac{1}{ms^2 + bs + k}$$

2.2.2 Convert State-Space to Transfer Functions

In the previous section, we saw that an ODE can be represented as a transfer function. Chapter 1 showed that ODEs can also be represented in state-space form. The unique transfer function from input u to output y of a linear time-invariant state-space system with state-equation $\dot{x} = Ax + Bu$ and output equation $y = Cx + Du$ is

$$\frac{Y}{U} = C(sI - A)^{-1}B + D \quad (2.5)$$

where I is the appropriately sized identity matrix, Y is the Laplace transform of the output y , and U is the Laplace transform of the input u . The proof is provided below.

2.2 Modeling Dynamic Systems using Laplace Transfer Functions

Proof of Eq. (2.5)

Example 2.2.3. Proof that $\frac{Y}{U} = C(sI - A)^{-1}B + D$

The proof that $\frac{Y}{U} = C(sI - A)^{-1}B + D$ is the unique Laplace transfer function form of the state-space system $\dot{x} = Ax + Bu$ and $y = Cx + Du$ is as follows. The derivation of Eq. (2.5) uses the Laplace transform of the state equation $\dot{x} = Ax + Bu$ to get

$$sX = AX + BU$$

which is solved for X to get

$$X = (sI - A)^{-1}BU$$

If we take the Laplace transform of the output equation $y = Cx + Du$, we get

$$Y = CX + DU$$

Plugging in X from Eq. (2.2.3) results in

$$Y = C(sI - A)^{-1}BU + DU$$

By dividing both sides by U , we get

$$\frac{Y}{U} = C(sI - A)^{-1}B + D$$

which is Eq. (2.5).

The following example converts a 2nd order state-space system to a transfer function.

Convert State-Space to a Transfer Function

Example 2.2.4. Convert a 2nd Order State-Space System to a Transfer Function

Find the transfer function of the state-space system

$$\begin{aligned}\dot{x} &= \begin{bmatrix} 0 & 1 \\ -1 & -2 \end{bmatrix}x + \begin{bmatrix} 0 & -1 \\ 1 & 0 \end{bmatrix}\begin{bmatrix} u_1 \\ u_2 \end{bmatrix} \\ \begin{bmatrix} y_1 \\ y_2 \end{bmatrix} &= \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}x + \begin{bmatrix} 1 & -1 \\ 0 & 0 \end{bmatrix}\begin{bmatrix} u_1 \\ u_2 \end{bmatrix}\end{aligned}$$

Solution: Because the state-space system has 2 inputs and two outputs, the transfer function from u to y has the matrix form:

$$\frac{Y}{U} = \begin{bmatrix} \frac{Y_1}{U_1} & \frac{Y_1}{U_2} \\ \frac{Y_2}{U_1} & \frac{Y_2}{U_2} \end{bmatrix}$$

where Y_1 is the Laplace transform of y_1 , U_1 is the Laplace transform of u_1 , etc. Using the conversion

equation, Eq. (2.5), the transfer function is:

$$\begin{aligned}
 \frac{Y}{U} &= C(sI - A)^{-1}B + D \\
 &= \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \left(\begin{bmatrix} s & 0 \\ 0 & s \end{bmatrix} - \begin{bmatrix} 0 & 1 \\ -1 & -2 \end{bmatrix} \right)^{-1} \begin{bmatrix} 0 & -1 \\ 1 & 0 \end{bmatrix} + \begin{bmatrix} 1 & -1 \\ 0 & 0 \end{bmatrix} \\
 &= \begin{bmatrix} \frac{s^2+2s+2}{s^2+2s+1} & \frac{-(s^2+3s+3)}{s^2+2s+1} \\ \frac{s}{s^2+2s+1} & \frac{1}{s^2+2s+1} \end{bmatrix}
 \end{aligned}$$

2.2.3 Convert Transfer Functions to State-Space

Not only can we convert state-space systems to transfer functions, as shown in Section 2.2.2, we can also convert transfer functions to state-space form. It is possible to convert a transfer function into an ODE by taking the inverse Laplace transform. We could then use principles from Section 1.7 to convert the ODE into state-space form.

This section provides a method to convert directly from a transfer function to state-space in a single step. To do so, it uses canonical forms, which are defined below. The two most common canonical forms are the **Controllable Canonical Form** and the **Observable Canonical Form**. The difference between these two forms and their uses are not important to mention at this point; for now we will simply reference the two methods.

Using canonical realizations, converting a transfer function to state-space form is accomplished by first arranging the transfer function to look like

$$\frac{Y(s)}{U(s)} = \frac{b_n s^n + b_{n-1} s^{n-1} + \dots + b_1 s + b_0}{s^n + a_{n-1} s^{n-1} + \dots + a_1 s + a_0} \quad (2.6)$$

where the highest power of s in the denominator has a coefficient of 1. Then the coefficients are placed into matrices to form the canonical state-space realizations. The Controllable Canonical Form is defined below.

Controllable Canonical Form

$$\begin{aligned}
 \dot{q} &= Aq + Bu & A = \begin{bmatrix} 0 & 1 & 0 & \dots & 0 \\ 0 & 0 & 1 & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \dots & 1 \\ -a_0 & -a_1 & -a_2 & \dots & -a_{n-1} \end{bmatrix} & B = \begin{bmatrix} 0 \\ 0 \\ \vdots \\ 0 \\ 1 \end{bmatrix} \\
 y &= Cq + Du & C = \begin{bmatrix} b_0 - a_0 b_n & b_1 - a_1 b_n & \dots & b_{n-1} - a_{n-1} b_n \end{bmatrix} & D = b_n
 \end{aligned}$$

Derivation of the Controllable Canonical Form

2.2 Modeling Dynamic Systems using Laplace Transfer Functions

To derive the controllable canonical form, we begin with the transfer function of Eq. (2.6) written as

$$Y = \frac{b_n s^n + b_{n-1} s^{n-1} + \dots + b_1 s + b_0}{s^n + a_{n-1} s^{n-1} + \dots + a_1 s + a_0} U$$

where the highest power of s in the denominator has a coefficient of 1. We can write this as

$$Y = (b_n s^n + b_{n-1} s^{n-1} + \dots + b_1 s + b_0) X \quad (2.7)$$

where X is defined as

$$X = \frac{1}{s^n + a_{n-1} s^{n-1} + \dots + a_1 s + a_0} U \quad (2.8)$$

Cross multiplying and then taking the inverse Laplace transform results in the following ODE.

$$x^{(n)} = -a_{n-1} x^{(n-1)} - \dots - a_1 \dot{x} - a_0 x + u \quad (2.9)$$

We choose state variables to be $x, \dot{x}, \dots, x^{(n-1)}$.

$$q = \begin{bmatrix} x \\ \dot{x} \\ \ddot{x} \\ \vdots \\ x^{(n-1)} \end{bmatrix} \quad (2.10)$$

As a result, the state derivative \dot{q} is

$$\dot{q} = \begin{bmatrix} \dot{x} \\ \ddot{x} \\ \ddot{x} \\ \vdots \\ x^{(n)} \end{bmatrix} \quad (2.11)$$

Solving for the state derivative in terms of the state variables (and using Eq. (2.9)) gives the controllable canonical state-equation:

$$\dot{q} = \begin{bmatrix} 0 & 1 & 0 & \dots & 0 \\ 0 & 0 & 1 & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \dots & 1 \\ -a_0 & -a_1 & -a_2 & \dots & -a_{n-1} \end{bmatrix} q + \begin{bmatrix} 0 \\ 0 \\ \vdots \\ 0 \\ 1 \end{bmatrix} u \quad (2.12)$$

To derive the output equation, we apply the inverse Laplace transform to Eq. (2.7).

$$y = b_n x^{(n)} + b_{n-1} x^{(n-1)} + \dots + b_1 \dot{x} + b_0 x \quad (2.13)$$

Using Eq. (2.9) for $x^{(n)}$, Eq. (2.13) can be written in terms of the state variables $x^{(n-1)}, \dots, \dot{x}, x$.

$$y = (b_{n-1} - b_n a_{n-1})x^{(n-1)} + \dots + (b_1 - b_n a_1)\dot{x} + (b_0 - b_n a_0)x + b_n u$$

This can be written in the controllable canonical form for the output equation:

$$y = [b_0 - a_0 b_n \quad b_1 - a_1 b_n \quad \dots \quad b_{n-1} - a_{n-1} b_n] q + b_n u$$

The derivation of the controllable canonical form is complete.

Another canonical state-space realization is the Observable Canonical Form. It is defined below.

Observable Canonical Form

$$\dot{q} = Aq + Bu \quad A = \begin{bmatrix} -a_{n-1} & 1 & 0 & \dots & 0 \\ -a_{n-2} & 0 & 1 & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ -a_1 & 0 & \vdots & \dots & 1 \\ -a_0 & 0 & 0 & \dots & 0 \end{bmatrix} \quad B = \begin{bmatrix} b_{n-1} - a_{n-1} b_n \\ b_{n-2} - a_{n-2} b_n \\ \vdots \\ b_1 - a_1 b_n \\ b_0 - a_0 b_n \end{bmatrix}$$

$$y = Cq + Du \quad C = [1 \quad 0 \quad \dots \quad 0 \quad 0] \quad D = b_n$$

Examples Converting Transfer Functions to State-Space

This section provides a couple of examples demonstrating how to convert transfer functions to state-space forms.

Convert a Transfer Function to State-Space

Example 2.2.5. Convert a transfer function to state-space form using canonical forms

Convert the following transfer function:

$$\frac{Y(s)}{U(s)} = \frac{5s + 4}{s^2 + 3s + 2}$$

to state space using controllable and observable canonical forms.

Solution: The coefficients of the numerator, based upon Eq 2.6, are:

$$b_2 = 0$$

$$b_1 = 5$$

$$b_0 = 4$$

2.2 Modeling Dynamic Systems using Laplace Transfer Functions

The coefficients of the denominator are:

$$a_1 = 3$$

$$a_0 = 2$$

The controllable canonical state-space form is:

$$\begin{bmatrix} \dot{x}_1 \\ \dot{x}_2 \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ -2 & -3 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} + \begin{bmatrix} 0 \\ 1 \end{bmatrix} u$$

$$y = \begin{bmatrix} 4 & 5 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} + \begin{bmatrix} 0 \end{bmatrix} u$$

and the observable canonical state-space form is:

$$\begin{bmatrix} \dot{x}_1 \\ \dot{x}_2 \end{bmatrix} = \begin{bmatrix} -3 & 1 \\ -2 & 0 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} + \begin{bmatrix} 5 \\ 4 \end{bmatrix} u$$

$$y = \begin{bmatrix} 1 & 0 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} + \begin{bmatrix} 0 \end{bmatrix} u$$

In the previous example, the transfer function was already in a form amenable to converting to state-space. This is because the highest-order coefficient in the denominator was one. The next example shows how to convert a transfer function to state-space form when the highest-order coefficient in the transfer function denominator is not equal to one. To do so, algebra is first required to set the highest-order denominator coefficient to one.

Canonical Forms

Example 2.2.6. Convert transfer functions to state-space using canonical forms

Convert the following transfer function:

$$\frac{V_L}{V_{in}} = \frac{RCLs^2 + Ls}{RCLs^2 + Ls + R}$$

to state space using both controllable and observable canonical forms.

Solution: We can convert the transfer function into state-space form by first dividing the numerator and denominator by RCL to get the leading s in the denominator all by itself like the form in Eq 2.6:

$$\frac{V_L}{V_{in}} = \frac{s^2 + \frac{1}{RC}s + 0}{s^2 + \frac{1}{RC}s + \frac{1}{CL}}$$

So the coefficients would be:

$$\begin{aligned} b_2 &= 1 & b_1 &= \frac{1}{RC} & b_0 &= 0 \\ a_1 &= \frac{1}{RC} & a_0 &= \frac{1}{CL} \end{aligned}$$

The controllable canonical state-space form is

$$\begin{aligned} \begin{bmatrix} \dot{x}_1 \\ \dot{x}_2 \end{bmatrix} &= \begin{bmatrix} 0 & 1 \\ -\frac{1}{CL} & -\frac{1}{RC} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} + \begin{bmatrix} 0 \\ 1 \end{bmatrix} u \\ y &= \begin{bmatrix} -\frac{1}{CL} & 0 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} + \begin{bmatrix} 1 \end{bmatrix} u \end{aligned}$$

and the observable canonical state-space form is:

$$\begin{aligned} \begin{bmatrix} \dot{x}_1 \\ \dot{x}_2 \end{bmatrix} &= \begin{bmatrix} -\frac{1}{RC} & 1 \\ -\frac{1}{CL} & 0 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} + \begin{bmatrix} 0 \\ -\frac{1}{CL} \end{bmatrix} u \\ y &= \begin{bmatrix} 1 & 0 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} + \begin{bmatrix} 1 \end{bmatrix} u \end{aligned}$$

2.3 Coordinate Transformations in State-Space Systems

The above section showed that a transfer function can have different state-space forms. For example, the same transfer function could be converted to the controllable canonical form or the observable canonical form. In fact, state-space systems can take on many different realizations by means of coordinate transformations. In a coordinate transformation, the state x is mapped to a different coordinate system z by a linear transformation:

$$x = Tz \quad (2.14)$$

where T is a constant, invertible $n \times n$ matrix, and z is a mapping of the time-varying states x in a new coordinate system. Then the whole state-space system can be converted to the new coordinate system. The state-equation is transformed as follows:

$$\begin{aligned} \dot{x} &= T\dot{z}, \quad \text{by linearity of the derivative} \\ &= Ax + Bu \\ &= ATz + Bu, \quad \text{since } x = Tz \end{aligned}$$

2.3 Coordinate Transformations in State-Space Systems

therefore,

$$T\dot{z} = ATz + Bu \quad (2.15)$$

$$\dot{z} = T^{-1}ATz + T^{-1}Bu \quad (2.16)$$

$$= A_T z + B_T u, \quad \text{where } A_T = T^{-1}AT, \text{ and } B_T = T^{-1}B \quad (2.17)$$

The output equation becomes:

$$y = Cx + Du \quad (2.18)$$

$$= CTz + Du \quad (2.19)$$

$$= C_T z + Du, \quad \text{where } C_T = CT \quad (2.20)$$

2.3.1 Coordinate Transformation to Controllable Canonical Form

Any controllable state-space system can be transformed to controllable canonical form by a coordinate transformation. Consider a controllable state-space system:

$$\dot{x} = Ax + Bu$$

$$y = Cx + Du$$

whose controllable canonical realization is

$$\dot{z} = A_c z + B_c u$$

$$y = C_c z + Du$$

where $A_c = T^{-1}AT$, $B_c = T^{-1}B$, and $C_c = CT$ are in the controllable canonical form:

$$A_c = \begin{bmatrix} 0 & 1 & 0 & \dots & 0 \\ 0 & 0 & 1 & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \dots & 1 \\ -a_0 & -a_1 & -a_2 & \dots & -a_{n-1} \end{bmatrix} \quad B_c = \begin{bmatrix} 0 \\ 0 \\ \vdots \\ 0 \\ 1 \end{bmatrix}$$

$$C_c = \begin{bmatrix} b_0 - a_0 b_n & b_1 - a_1 b_n & \dots & b_{n-1} - a_{n-1} b_n \end{bmatrix} \quad D = b_n$$

where the coefficients a_i , $i = 0, \dots, n-1$ and b_i , $i = 0, \dots, n$ are the transfer function coefficients resulting from Eqs. (2.5) and (2.6). Then the linear transformation matrix T is

$$T = \mathcal{C}(A, B) [\mathcal{C}(A_c, B_c)]^{-1} \quad (2.21)$$

The controllability matrix $\mathcal{C}(A, B)$ is defined as

$$\mathcal{C}(A, B) = \begin{bmatrix} B & | & AB & | & A^2B & | & \dots & | & A^{n-1}B \end{bmatrix} \quad (2.22)$$

The notation in Eq. (2.22) attempts to indicate that B is the first column of the controllability matrix $\mathcal{C}(A, B)$, AB is the second column, A^2B is the third column, and so forth. A state-space system is **controllable** if the inverse of its controllability matrix exists.

Proof that Eq. (2.21) is the Transformation Matrix to Controllable Canonical Form

Example 2.3.1. Proof of Eq. (2.21)

Prove that Eq. (2.21),

$$T = \mathcal{C}(A, B) [\mathcal{C}(A_c, B_c)]^{-1}$$

is the linear transformation matrix to controllable canonical form where the controllability matrix is given by Eq. (2.22),

$$\mathcal{C}(A, B) = \begin{bmatrix} B & | & AB & | & A^2B & | & \cdots & | & A^{n-1}B \end{bmatrix}$$

Solution: This derivation will provide proof for an arbitrary, controllable, third-order state-space system, $\dot{x} = Ax + Bu$ and $y = Cx + Du$. Mathematical induction using the same reasoning provides proof for higher-order systems. We begin with the third-order controllable canonical state matrix A_c :

$$A_c = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ -a_0 & -a_1 & -a_2 \end{bmatrix}$$

and the controllable canonical input matrix B_c :

$$B_c = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}$$

From Eq. (2.17), the following relationships exist:

$$A_c T^{-1} = T^{-1} A \quad (2.23)$$

$$B_c = T^{-1} B \quad (2.24)$$

The T matrix and its inverse are 3×3 matrices. Let the inverse matrix be represented by

$$T^{-1} = \begin{bmatrix} t_1 \\ t_2 \\ t_3 \end{bmatrix}$$

where t_i , $i = 1, 2, 3$ are 1×3 row vectors that make up the T^{-1} matrix. With this definition, Eq. (2.23) becomes

2.3 Coordinate Transformations in State-Space Systems

$$\begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ -a_0 & -a_1 & -a_2 \end{bmatrix} \begin{bmatrix} t_1 \\ t_2 \\ t_3 \end{bmatrix} = \begin{bmatrix} t_1 \\ t_2 \\ t_3 \end{bmatrix} A \quad (2.25)$$

$$\begin{bmatrix} t_2 \\ t_3 \\ -a_0 t_1 - a_1 t_2 - a_2 t_3 \end{bmatrix} = \begin{bmatrix} t_1 A \\ t_2 A \\ t_3 A \end{bmatrix} \quad (2.26)$$

From this, we see that

$$t_2 = t_1 A \quad (2.27)$$

$$t_3 = t_2 A \quad (2.28)$$

$$= t_1 A^2 \quad (2.29)$$

From Eq. (2.24) we get the following relationship:

$$\begin{aligned} B_c &= T^{-1} B \\ \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} &= \begin{bmatrix} t_1 \\ t_2 \\ t_3 \end{bmatrix} B \\ &= \begin{bmatrix} t_1 B \\ t_2 B \\ t_3 B \end{bmatrix} \\ &= \begin{bmatrix} t_1 B \\ t_1 AB \\ t_1 A^2 B \end{bmatrix}, \quad \text{from Eqs. (2.27) and (2.29)} \end{aligned}$$

Since $0 = t_1 B$, $0 = t_1 AB$, and $1 = t_1 A^2 B$, we can write the last equation in a row vector:

$$\begin{bmatrix} 0 & 0 & 1 \end{bmatrix} = t_1 \begin{bmatrix} B & | & AB & | & A^2 B \end{bmatrix} \quad (2.30)$$

The controllability matrix is $\mathcal{C}(A, B) = [B \quad | \quad AB \quad | \quad A^2 B]$. Therefore, solving Eq. (2.30) for t_1 gives

$$t_1 = \begin{bmatrix} 0 & 0 & 1 \end{bmatrix} [\mathcal{C}(A, B)]^{-1} \quad (2.31)$$

Next, we will solve for t_2 in a similar manner. Notice that $t_2 B = 0$ and $t_3 B = 1$. Since $t_3 = t_2 A$, then $t_2 AB = 1$. Next, we right-multiply the last row of Eq. (2.26) by B to get

$$-a_0 t_1 B - a_1 t_2 B - a_2 t_3 B = t_3 AB \quad (2.32)$$

But, since $t_1B = 0$, $t_2B = 0$, $t_3B = 1$, and $t_3 = t_2A$, Eq. (2.32) becomes

$$t_2A^2B = -a_2 \quad (2.33)$$

We can combine this equation with the earlier solutions $t_2B = 0$ and $t_3B = 1$, to get

$$\begin{bmatrix} t_2B & t_2AB & t_2A^2B \end{bmatrix} = \begin{bmatrix} 0 & 1 & -a_2 \end{bmatrix} \quad (2.34)$$

$$t_2 \begin{bmatrix} B & | & AB & | & A^2B \end{bmatrix} = \begin{bmatrix} 0 & 1 & -a_2 \end{bmatrix} \quad (2.35)$$

$$t_2 = \begin{bmatrix} 0 & 1 & -a_2 \end{bmatrix} [\mathcal{C}(A, B)]^{-1} \quad (2.36)$$

To solve for t_3 , we right-multiply the last row of Eq. (2.26) by AB to get

$$-a_0t_1AB - a_1t_2AB - a_2t_3AB = t_3A^2B \quad (2.37)$$

Using the relationships $t_1A = t_2$ and $t_2A = t_3$, Eq. (2.37) becomes

$$-a_0t_2B - a_1t_3B - a_2t_3AB = t_3A^2B \quad (2.38)$$

Now we can use the equations $t_2B = 0$, $t_3B = 1$, and $t_3AB = t_2A^2B = -a_2$, and Eq. (2.38) becomes

$$-a_1 + a_2^2 = t_3A^2B \quad (2.39)$$

Combining the equations $t_3B = 1$, $t_3AB = -a_2$ with $t_3A^2B = -a_1 + a_2^2$ from Eq. (2.39), we get

$$t_3 = \begin{bmatrix} 1 & -a_2 & -a_1 + a_2^2 \end{bmatrix} [\mathcal{C}(A, B)]^{-1} \quad (2.40)$$

Combining Equations (2.31), (2.36), and (2.40), the inverse matrix is

$$T^{-1} = \begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & -a_2 \\ 1 & -a_2 & -a_1 + a_2^2 \end{bmatrix} [\mathcal{C}(A, B)]^{-1} \quad (2.41)$$

$$= \begin{bmatrix} B_c & | & A_cB_c & | & A_c^2B_c \end{bmatrix} [\mathcal{C}(A, B)]^{-1} \quad (2.42)$$

$$= \mathcal{C}(A_c, B_c) [\mathcal{C}(A, B)]^{-1} \quad (2.43)$$

Finally, we can take the inverse of Eq. (2.43) to get the transformation matrix T

$$T = \mathcal{C}(A, B) [\mathcal{C}(A_c, B_c)]^{-1}$$

which was the goal of this derivation. The proof is complete.

2.3 Coordinate Transformations in State-Space Systems

2.3.2 Coordinate Transforms to Observable Canonical Form

The transformation matrix to convert a state-space system to its observable canonical form is

$$T = [\mathcal{O}(A, C)]^{-1} \mathcal{O}(A_o, C_o) \quad (2.44)$$

where $\mathcal{O}(A, C)$ is the observability matrix:

$$\mathcal{O}(A, C) = \begin{bmatrix} C \\ CA \\ CA^2 \\ \vdots \\ CA^{n-1} \end{bmatrix} \quad (2.45)$$

A state-space system is **observable** if the inverse of its observability matrix $\mathcal{O}(A, C)$ exists. The proof is not provided here, but its derivation is similar to Example 2.3.1. The matrices A_o , B_o , and C_o are defined by the observable canonical form:

$$A_o = \begin{bmatrix} -a_{n-1} & 1 & 0 & \dots & 0 \\ -a_{n-2} & 0 & 1 & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ -a_1 & 0 & \vdots & \dots & 1 \\ -a_0 & 0 & 0 & \dots & 0 \end{bmatrix} \quad B_o = \begin{bmatrix} b_{n-1} - a_{n-1}b_n \\ b_{n-2} - a_{n-2}b_n \\ \vdots \\ b_1 - a_1b_n \\ b_0 - a_0b_n \end{bmatrix}$$

$$C_o = [1 \ 0 \ \dots \ 0 \ 0] \quad D = b_n$$

The coefficients a_i , $i = 0, \dots, n-1$ and b_i , $i = 0, \dots, n$ are the denominator and numerator coefficients of the transfer function resulting from Eqs. (2.5) and (2.6). The following example demonstrates how to use Eq. (2.44) to convert a state-space system to its observable canonical form.

Converting a State-Space System to its Observable Canonical Form

Example 2.3.2. Converting a state-space system to its observable canonical form

Convert the following state-space system to its observable canonical form using the transformation matrix of Eq. (2.44).

$$\dot{x} = \begin{bmatrix} -2 & 4 & 1 \\ 1 & 0 & 3.2 \\ -1 & 2 & 4.8 \end{bmatrix} x + \begin{bmatrix} 2 \\ -8.6 \\ 1.4 \end{bmatrix} u$$

$$y = \begin{bmatrix} 3 & 0 & -9.3 \end{bmatrix} x$$

Solution: To calculate the transformation matrix of Eq. (2.44), we actually must calculate the observable canonical matrices A_o and C_o first. We begin by finding the transfer function using Eq. (2.5):

$$\begin{aligned}
 \frac{Y}{U} &= C(sI - A)^{-1}B + D \\
 &= \begin{bmatrix} 3 & 0 & -9.3 \end{bmatrix} \left(s \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} - \begin{bmatrix} -2 & 4 & 1 \\ 1 & 0 & 3.2 \\ -1 & 2 & 4.8 \end{bmatrix} \right)^{-1} \begin{bmatrix} 2 \\ -8.6 \\ 1.4 \end{bmatrix} \\
 &= \frac{-7.02s^2 + 24.72s + 474}{s^3 - 2.8s^2 - 19s + 17.2}
 \end{aligned}$$

Knowing the numerator and denominator coefficients of the transfer function allows us to get the observable canonical form matrices (see Section 2.2.3):

$$\begin{aligned}
 A_o &= \begin{bmatrix} 2.8 & 1 & 0 \\ 19 & 0 & 1 \\ -17.2 & 0 & 0 \end{bmatrix} & B_o &= \begin{bmatrix} -7.02 \\ 24.72 \\ 474 \end{bmatrix} \\
 C_o &= \begin{bmatrix} 1 & 0 & 0 \end{bmatrix} & D &= 0
 \end{aligned}$$

Although part of the goal was to find the observable canonical matrices, we are not finished yet because the goal was to use the transformation matrix of Eq. (2.44) to find the matrices. The transformation matrix is

$$T = [\mathcal{O}(A, C)]^{-1} \mathcal{O}(A_o, C_o)$$

where the observability matrix of the original system is

$$\begin{aligned}
 \mathcal{O}(A, C) &= \begin{bmatrix} \begin{bmatrix} 3 & 0 & -9.3 \end{bmatrix} \\ \begin{bmatrix} 3 & 0 & -9.3 \end{bmatrix} \begin{bmatrix} -2 & 4 & 1 \end{bmatrix} \\ \begin{bmatrix} 3 & 0 & -9.3 \end{bmatrix} \begin{bmatrix} 1 & 0 & 3.2 \end{bmatrix} \\ \begin{bmatrix} 3 & 0 & -9.3 \end{bmatrix} \begin{bmatrix} -1 & 2 & 4.8 \end{bmatrix}^2 \end{bmatrix} \\
 &= \begin{bmatrix} 3 & 0 & -9.3 \\ 3.3 & -6.6 & -41.64 \\ 28.44 & -70.08 & -217.692 \end{bmatrix}
 \end{aligned}$$

2.3 Coordinate Transformations in State-Space Systems

The observability matrix of the observable canonical form is

$$\begin{aligned}\mathcal{O}(A_o, C_o) &= \left[\begin{array}{c} \begin{bmatrix} 1 & 0 & 0 \\ 1 & 0 & 0 \\ 1 & 0 & 0 \end{array} \begin{bmatrix} 1 & 0 & 0 \\ 2.8 & 1 & 0 \\ 19 & 0 & 1 \\ -17.2 & 0 & 0 \end{bmatrix} \end{array} \right] \\ &= \begin{bmatrix} 1 & 0 & 0 \\ 2.8 & 1 & 0 \\ 26.84 & 2.8 & 1 \end{bmatrix}\end{aligned}$$

Therefore, the transformation matrix is

$$\begin{aligned}T &= [\mathcal{O}(A, C)]^{-1} \mathcal{O}(A_o, C_o) \\ &\approx \begin{bmatrix} 0.3228 & -0.1188 & 0.0152 \\ -0.2415 & 0.0309 & -0.0233 \\ -0.0034 & -0.0383 & 0.0049 \end{bmatrix}\end{aligned}$$

Now, we can use the transformation matrix T to get the matrices for the observable canonical form:

$$\begin{aligned}A_o &= T^{-1}AT = \begin{bmatrix} 2.8 & 1 & 0 \\ 19 & 0 & 1 \\ -17.2 & 0 & 0 \end{bmatrix} & B_o &= T^{-1}B = \begin{bmatrix} -7.02 \\ 24.72 \\ 474 \end{bmatrix} \\ C_o &= CT = \begin{bmatrix} 1 & 0 & 0 \end{bmatrix} & D &= 0\end{aligned}$$

which is exactly the same as what was already calculated above. Admittedly, it seems like a fairly useless exercise to use the transformation matrix to calculate the observable canonical form, because it actually relies on knowing the observable canonical form in the first place! However, as we will see in Section 13.1.1 and Section 9.3.3, this approach will become useful for pole-placement techniques.

2.3.3 Coordinate Transforms do not Affect Transfer Functions

Linear transformations of the type $x = Tz$ do not affect transfer functions. For example, the controllable canonical form and observable canonical forms of a state-space system will have the same transfer function as the original state-space system. The transfer function from U to Y was given by Eq. (2.5) as $\frac{Y}{U} = C(sI - A)^{-1}B + D$. Using a transformed coordinate system, it becomes

$$\begin{aligned}
 \frac{Y}{U} &= C_T (sI - A_T)^{-1} B_T + D \\
 &= CT(sI - T^{-1}AT)^{-1} T^{-1}B + D \\
 &= CT(sT^{-1}T - T^{-1}AT)^{-1} T^{-1}B + D \\
 &= CT(T^{-1}(sI - A)T)^{-1} T^{-1}B + D \\
 &= CTT^{-1}(sI - A)^{-1} TT^{-1}B + D \\
 &= C(sI - A)^{-1} B + D
 \end{aligned}$$

Therefore, coordinate transformations of state-space systems do not affect their transfer functions.

2.4 Poles and Zeros of Laplace Transfer Functions

2.4.1 Poles of Transfer Functions

Section 1.6 discussed how eigenvalues of the state matrix (A) are the poles of state-space systems. Eigenvalues with negative real parts were stable. Eigenvalues with zero real parts were marginally stable, and eigenvalues with positive real parts were unstable. A system is only as stable as its least stable pole. These poles also determined the response time and oscillatory behavior of stable systems (see Figure 1.4). Transfer functions also have poles. Poles of transfer functions are exactly equivalent to poles of state-space systems. The general form of the transfer function from U to Y is

$$\frac{Y}{U} = \frac{b_n s^n + b_{n-1} s^{n-1} + \dots + b_1 s + b_0}{s^n + a_{n-1} s^{n-1} + \dots + a_1 s + a_0} \quad (2.46)$$

The denominator of the transfer function is the **characteristic polynomial**. The roots of the denominator are therefore the **poles** of the transfer function. The **transfer function order** is equal to its number of poles.

Poles of a 2nd Order Transfer Function

Example 2.4.1. Poles of a 2nd order transfer function

Given the second order transfer function

$$\frac{Y}{U} = \frac{3s + 2}{s^2 + 2s + 6}$$

Determine the following properties:

1. transfer function poles
2. transfer function stability
3. transfer function time-constant τ
4. natural frequency ω_n
5. damping ratio ζ

2.4 Poles and Zeros of Laplace Transfer Functions

Solution: The denominator of the transfer function is the characteristic polynomial. It is $s^2 + 2s + 6$. All of the requested properties can be determined from the characteristic polynomial.

1. The transfer function poles are the roots of the characteristic polynomial. Using the quadratic formula, the roots are determined to be

$$s_1 = -1 + j2.2361$$

$$s_2 = -1 - j2.2361$$

where $j = \sqrt{-1}$ is the imaginary number.

2. The real part of both s_1 and s_2 is -1. Because the real part of every pole is negative, the transfer function is stable.
3. Because the system is stable, it has a time-constant. Using Eq. (1.44), and recognizing that $\lambda_{\text{slowest}} = s_{\text{slowest}}$ is the pole closest to the imaginary axis, the time-constant is

$$\begin{aligned}\tau &= \frac{-1}{\text{Real}(\lambda_{\text{slowest}})} \\ &= \frac{-1}{-1} \\ &= 1\end{aligned}$$

The time-constant of the transfer function is therefore $\tau = 1$.

4. To get the natural frequency ω_n and damping ratio ζ , we must compare the characteristic polynomial $s^2 + 2s + 6$ with the general form of a second order polynomial $s^2 + 2\zeta\omega_n s + \omega_n^2$ (see Section 1.6.3). The coefficients must match, i.e., $2\zeta\omega_n = 2$ and $\omega_n^2 = 6$. Therefore, the natural frequency of the transfer function is $\omega_n = \sqrt{6} \approx 2.4495$.
5. Since $2\zeta\omega_n = 2$ and $\omega_n = \sqrt{6}$, the damping ratio is

$$\begin{aligned}\zeta &= \frac{2}{2\omega_n} \\ &= \frac{1}{\sqrt{6}} \\ &\approx 0.4082\end{aligned}$$

2.4.2 Zeros of Transfer Functions

Zeros are the roots of the polynomial in the numerator of a transfer function. Like poles, zeros are complex numbers having real and imaginary parts. If the real part of the zero is negative (more specifically ≤ 0), the zero is a **minimum phase zero**. If the real part is positive (> 0), the zero is a **non-minimum phase zero**. Minimum phase zeros are more amenable to feedback control than non-minimum phase zeros. Non-minimum phase zeros create complications in feedback control because their inverses are unstable. If you have ever tried riding a swing bike (a bike that is steered using the back wheel), you may agree that

non-minimum phase systems are more difficult to control. One reason is because closed-loop poles of feedback controllers are drawn towards open-loop zeros as discussed in Section 13.5. Zeros, especially non-minimum phase zeros, can affect the response-time, overshoot, and oscillatory behavior of dynamic systems.

Poles and zeros are sometimes mapped on a graph of the complex plane. Figure 2.1 shows poles as x's and zeros as o's. The poles in Figure 2.1 are located at $p_{1,2} = -1 \pm 5j$, $p_{3,4} = \pm 8$ and $p_5 = 2$. The zeros are located at $z_1 = -5$, $z_2 = 0$, and $z_{3,4} = 4 \pm 3j$. The zeros at $z_{3,4} = 4 \pm 3j$ are non-minimum phase zeros.

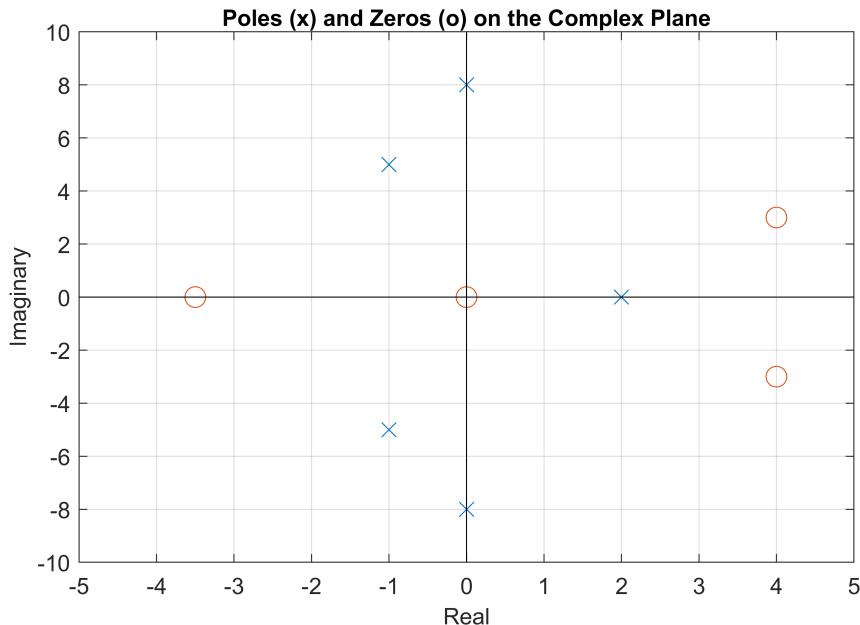


Figure 2.1 Poles and zeros in the complex plane

Non-minimum phase systems have the unusual characteristic that they must “go the wrong way first”. A familiar example is an inverted pendulum. Consider balancing a broom in the palm of your hand so that the brushed end is vertically upward. To move the broom to the right while keeping it balanced, you must first move your hand to the left. This causes the broom to tilt to the right. Once tilted to the right, you can move your hand, and the broom, to the right while keeping it balanced. Notice that you must move the wrong way first.

A traditional airplane is another example of a non-minimum phase system. To cause the airplane to climb, the elevators are tilted upwards. Air passing over the elevators pushes the back wing *down* causing the airplane to pitch upwards. Once pitched upwards, the airplane can climb to a higher altitude. Notice, however, that the tail first had to go down to go up.

Transfer Function Zeros

Example 2.4.2. Zeros of a transfer function

Given the transfer function

2.4 Poles and Zeros of Laplace Transfer Functions

$$\frac{Y}{U} = \frac{s - 4}{(s + 3)(s + 5)}$$

Determine the following properties:

1. transfer function zeros
2. minimum or non-minimum phase
3. transfer function stability

Plot the response of the transfer function to a unit step input and discuss the effect of the zeros on the transient-time and oscillatory behavior. Does the response “go the wrong way first”?

Solution:

1. The roots of the numerator are the zeros of the transfer function. Setting the numerator to zero, $s - 4 = 0$, reveals that the transfer function has only one zero at $s = 4$.
2. Since the real part of the numerator is positive, the zero is a non-minimum phase zero.
3. The roots of the denominator are the poles of the transfer function. By inspection, if s is replaced by $s = -3$ or $s = -5$, the denominator is equal to zero. Therefore, $s = -3$ and $s = -5$ are poles of the transfer function. Since both have negative real parts, the transfer function is stable.

This example will use MATLAB to plot the response of the transfer function to a unit step input. To do so, it first converts the transfer function to state-space using the controllable canonical form (see Section 2.2.3):

$$\begin{aligned}\dot{x} &= \begin{bmatrix} 0 & 1 \\ -15 & -8 \end{bmatrix}x + \begin{bmatrix} 0 \\ 1 \end{bmatrix}u \\ y &= \begin{bmatrix} -4 & 1 \end{bmatrix}x + 0u\end{aligned}$$

Where, by definition of a unit step input, $u = 1$ for $t \geq 0$. The initial conditions are zero by the assumptions of transfer functions. The MATLAB implementation is shown below:

```
close all %Close all open figures
clear all %Clear all variables and functions
clc %Clear the command window

% The state-space form of the transfer function Y/U=(s-4)/((s+3)*(s+5))
A = [0,1;-15,-8];
B = [0;1];
C = [-4,1];
D = 0;

% Discrete timestep
```

```

dt = 0.01; %(s)

% Time vector 0 to 6 s
t = 0:dt:6; %(s)

% The unit step input
u = 1 * ones(size(t));

%Discrete-time state-transition and input-transition matrices
%Exact method
Fd = expm([A*dt,B*dt;zeros(1,3)]);
Ad = Fd(1:2,1:2);
Bd = Fd(1:2,3);

%Allocate memory to store the output trajectory
N = length(t); %Number of timesteps
y = zeros(1,N);

%Initial condition
x = [0;0];

%Run the simulation
for ii = 1:N
    %Solve the output equation and store the result
    y(ii) = C*x+D*u(ii);
    %Solve the state equation
    x = Ad*x+Bd*u(ii);
end

%Graph the resulting trajectory
figure %create a figure
%Plot the results
plot(t,y)
xlabel('Time (t)') %Label the x-axis
ylabel('Output Trajectory (y)') %Label the y-axis
grid on %Turn on the grid

```

The resulting graph is shown below. Notice that the output trajectory eventually converges to a final value of approximately -0.267 . However, the trajectory first goes the wrong way to positive numbers before eventually converging towards its negative final value. This is a result of the non-minimum phase zero. Also, because the slowest pole is $s = -3$, the time-constant is $\tau = \frac{1}{3}$, and the transient-time is $4\tau \approx 1.33$. Although possible, the non-minimum phase zero appears to have almost no effect on the transient time. Both poles had zero imaginary parts, which characteristic results in no overshoot nor oscillations. Although the transfer function trajectory goes the wrong way first, there is no overshoot nor oscillations around the final value.

2.4 Poles and Zeros of Laplace Transfer Functions

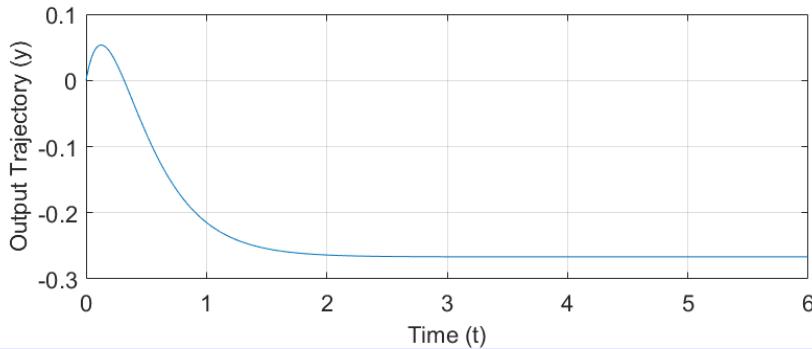


Figure 2.2 Trajectory of the transfer function output to a unit step input. The non-minimum phase behavior is evident in the fact that the trajectory goes the wrong way first.

2.4.3 Pole-Zero Cancellation and Stability

It is possible for zeros in a transfer function to cancel the effect of poles. This is only true, however, for stable poles being canceled by minimum-phase zeros. A non-minimum phase zero mathematically can cancel an unstable pole (a pole on the right-half of the complex plane). However, because the system is unstable, the unstable pole cancellation does not change the stability of the system. It is still internally unstable, and its response will grow unbounded with time. This is shown in the example below.

Unstable Zero-Pole Cancellation

Example 2.4.3. Unstable zero-pole cancellation

The state-space system

$$\begin{aligned}\dot{x} &= \begin{bmatrix} 0 & 1 \\ 2 & -1 \end{bmatrix} x + \begin{bmatrix} 0 \\ 1 \end{bmatrix} u \\ y &= \begin{bmatrix} -1 & 1 \end{bmatrix} x\end{aligned}$$

has poles at $s = -2$ and $s = 1$. It has a zero at $s = 1$. Show that, mathematically, the unstable pole at $s = 1$ is canceled by the non-minimum phase zero and that the resulting transfer function appears to be stable. Also simulate the state-space system with a constant input of $u = 5$ and zero initial conditions for 35 s to show that the response eventually grows unbounded. Comment on how this demonstrates the internal instability of the system.

Solution: We begin by using Eq. (2.5) to find the transfer function of the state-space system to be

$$\begin{aligned}\frac{Y}{U} &= C(sI - A)^{-1} B + D \\ &= \frac{s - 1}{(s - 1)(s + 2)} \\ &= \frac{1}{s + 2}, \quad \text{because } (s - 1) \text{ in the numerator cancels } (s - 1) \text{ in the denominator}\end{aligned}$$

The controllable canonical form of $\frac{Y}{U} = \frac{1}{s+2}$ is

$$\begin{aligned}\dot{q} &= -2q + u \\ y &= q\end{aligned}$$

The following MATLAB code simulates the response of both realizations and compares the responses.

```

close all % Close all figures
clear % Clear the variables
clc % Clear the command window

%Get time parameters
dt = 0.1; %(s) simulation timestep
t = 0:dt:35; %(s) time vector
N = length(t); %Number of simulation timesteps

%Set the constant input
u = 5;

%Write the state-space matrices for the original system
A = [0,1;2,-1]; %State matrix
B = [0;1]; %Input matrix
C = [-1,1]; %Output matrix
Fd = expm([A*dt,B*dt;zeros(1,3)]); %Solution matrix
Ad = Fd(1:2,1:2); %Discrete state-transition matrix
Bd = Fd(1:2,3); %Discrete input-transition matrix

%Write the state-space matrices for the pole-zero cancellation system
Ac = -2;
Bc = 1;
Cc = 1;
Fdc = expm([Ac*dt,Bc*dt;0,0]);
Adc = Fdc(1,1);
Bdc = Fdc(1,2);

%Allocate memory to store the outputs
y = zeros(1,N); %output of the original system
yc = zeros(1,N); %output of the zero-pole cancellation system

%Set initial conditions
x = [0;0]; %Initial state of the original system
q = 0; %Initial state of the pole-zero cancellation system

%Run the simulation
for ii = 1:N
    %Store the outputs
    y(ii) = C*x; %Original system output
    yc(ii) = Cc*q; %Zero-pole cancellation system output

```

2.5 Zero-Pole-Gain Form of a Transfer Function

```
%Update the states
x = Ad*x+Bd*u;
q = Adc*q+Bdc*u;
end

%Plot the output
plot(t,y,t,yc,'--')
legend('Original','Pole-Zero','Location','best')
title('A Non-Minimum Phase Zero Cannot Cancel an Unstable Pole')
ylabel('Output (y)')
xlabel('Time (s)')
grid on
```

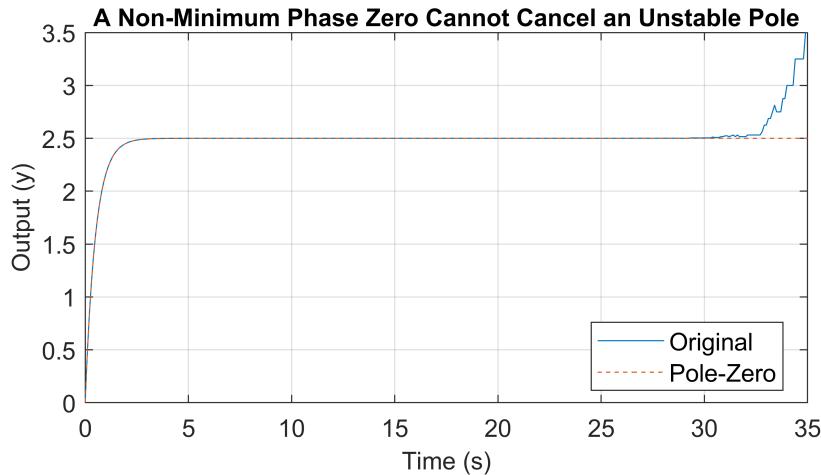


Figure 2.3 Although the zero cancels the unstable pole mathematically, the system remains unstable. The response of the original system at first follows the pole-zero cancellation system but eventually grows unbounded.

Unlike unstable poles, stable poles can be canceled by nearby or collocated zeros. The effects of the canceled pole are lost in the system's response. Because of this, zeros can affect the rise-time, overshoot, oscillations and other characteristics of a system's response.

2.5 Zero-Pole-Gain Form of a Transfer Function

A transfer function can be factored into its zero-pole-gain form as follows:

$$\frac{Y}{U} = k \frac{(s - z_1)(s - z_2) \cdots (s - z_m)}{(s - p_1)(s - p_2) \cdots (s - p_n)} \quad (2.47)$$

$$= k \frac{\prod_{j=1}^m (s - z_j)}{\prod_{i=1}^n (s - p_i)} \quad (2.48)$$

where each z_j , $j = 1, 2, \dots, m$ is a zero of the transfer function, and each p_i , $i = 1, 2, \dots, n$ is a pole. The transfer function gain is k . It is *not* the steady-state or DC gain of the transfer function.

Zero-Pole-Gain Form of a Transfer Function

Example 2.5.1. Finding the zero-pole-gain form of a transfer function

Convert the following transfer function

$$\frac{Y}{U} = \frac{6s^2 + 12s + 3}{s^2 + 3s + 2}$$

to its zero-pole-gain form. Show that the transfer function gain k is different from the DC or steady-state gain of the transfer function.

Solution: To convert the transfer function to its zero-pole-gain form, we must first find the zeros and poles. Zeros are the roots of the numerator polynomial. Using the quadratic formula, we find the roots of $6s^2 + 12s + 3$ to be

$$z_1 = -1.7071$$

$$z_2 = -0.2929$$

Poles are the roots of the denominator polynomial $s^2 + 3s + 2$. The quadratic formula finds them to be

$$p_1 = -2$$

$$p_2 = -1$$

The gain k of the transfer function is the value of the highest-order non-zero coefficient in the numerator (see Eq. (2.46)) when the highest order denominator coefficient (a_n) is one. Since the transfer function of this example is already in the general form, the transfer function gain k is

$$k = 6$$

which is the coefficient multiplied by s^2 in the numerator. We need to compare this gain with the DC or steady-state gain of the transfer function. Section 2.6.2 explained that the steady-state gain of a transfer function is found by setting $s = 0$. Therefore, the steady-state gain of the transfer function of this example is $\frac{3}{2}$.

2.6 Final Value Theorem

The final value theorem provides a way to determine what value a system will eventually reach over time by evaluating the system's Laplace transfer function. It applies only to stable systems. If $G(s)$ is the stable Laplace transfer function from $U(s)$ to $Y(s)$, i.e.,

2.6 Final Value Theorem

$$\frac{Y(s)}{U(s)} = G(s) \quad (2.49)$$

then the final value theorem states that

$$\lim_{t \rightarrow \infty} y(t) = \lim_{s \rightarrow 0} sG(s)U(s) \quad (2.50)$$

2.6.1 Final Value for Constant Inputs

If the input $u(t)$ is a constant or a step function, *i.e.*, $u(t) = c$, then its Laplace transform $U(s)$ is (see Section 2.1)

$$U(s) = \frac{1}{s}c \quad (2.51)$$

As a result, if the input $u = c$ is constant, the final value theorem states that

$$\begin{aligned} \lim_{t \rightarrow \infty} y(t) &= \lim_{s \rightarrow 0} sG(s)\frac{1}{s}c \\ &= \lim_{s \rightarrow 0} G(s)c \end{aligned} \quad (2.52)$$

for stable transfer functions $G(s)$.

Using the Final Value Theorem

Example 2.6.1. Using the Final Value Theorem

Determine the final value of the transfer function

$$\frac{Y}{U} = \frac{s^2 + 4}{s^2 + 4s + 8}$$

assuming that the input u is constant.

Solution: Because the input u is constant, we can use Eq. (2.52). The final value of the transfer function is therefore

$$\lim_{s \rightarrow 0} \frac{s^2 + 4}{s^2 + 4s + 8} = \frac{1}{2}$$

The steady-state (ss) relationship between u and y is

$$y_{ss} = \frac{1}{2}u$$

2.6.2 Steady-State Gain of Stable Transfer Functions

The **steady-state** gain (or DC gain) of a transfer function G is defined by its final value, assuming its input is a step or constant input, *i.e.* $\lim_{s \rightarrow 0} G$. The steady-state gain is only defined for stable transfer functions.

2.7 Transfer Functions as Complex Numbers

Laplace-domain transfer functions can be analyzed in the frequency domain by replacing the Laplace independent variable with $s = j\omega$. The frequency ω (usually rad/s) is the sinusoidal frequency of the input U and steady-state output Y . The imaginary number is j . With the substitution $s = j\omega$, the transfer function becomes a complex function of the frequency. A review of complex numbers may be helpful, and it is presented next. Readers with a strong background in complex math are welcome to skip the review.

2.7.1 Review of Complex Numbers

The definition of the imaginary number j is

$$j^2 = -1 \text{ or } j = \sqrt{-1}$$

A number Z is **complex** if it has real (Re) and imaginary (Im) parts:

$$\begin{aligned} Z &= \text{Re} + j \cdot \text{Im} \\ Z &= a + jb \end{aligned}$$

where a is the real part and b is the imaginary part. For example, in the complex number $Z = 5 + j4$, the imaginary part is 4 because it is multiplied by the imaginary number, and the real part is 5.

It is often convenient to represent complex numbers graphically with the x-axis representing the real part and the y-axis representing the imaginary part of the complex number. The graph is called the **complex plane**. Figure 2.4 shows several examples of complex numbers in the complex plane.

The trigonometric form of complex numbers can be found by expressing the point in the complex plane by its radius from the origin and the angle from the positive real axis as can be seen in Figure 2.5.

Complex Number Notation

Complex numbers can be represented in different ways. The **rectangular form** is:

$$z = a + jb$$

The **trigonometric form** of complex numbers is expressed as:

$$z = r \cos(\theta) + j r \sin(\theta)$$

The **exponential form** of complex numbers uses Euler's identity, $e^{j\theta} = \cos(\theta) + j \sin(\theta)$, to simplify the trigonometric form to:

$$z = re^{j\theta}$$

A shortcut for writing the exponential form is:

$$z = r \angle \theta$$

2.7 Transfer Functions as Complex Numbers

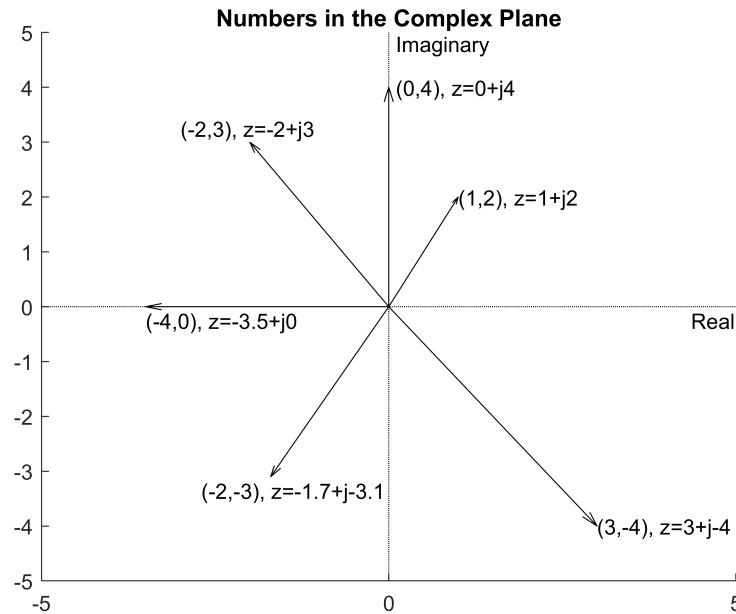


Figure 2.4 Various complex numbers inside a complex plane. The term $j-3.1$ should be interpreted as $-j3.1$, and $j-4$ should be interpreted as $-j4$.

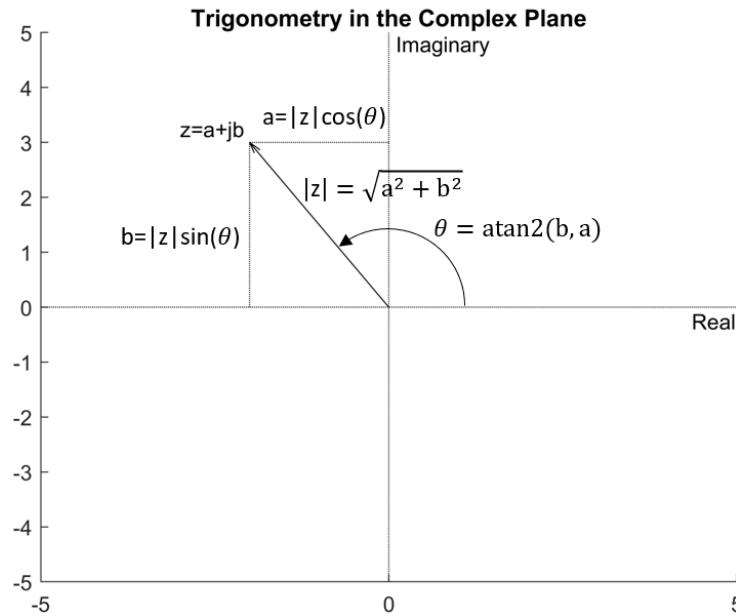


Figure 2.5 Depiction of complex numbers in trigonometric form

Trigonometric Coordinate Conversions

In trigonometric form, the radius or magnitude is calculated as:

$$r = \sqrt{a^2 + b^2}$$

The angle from the positive real axis is calculated from:

$$\tan(\theta) = \frac{b}{a}$$

Its solution uses the four-quadrant arc-tangent function, *i.e.*, the atan2 function:

$$\theta = \text{atan2}(b, a)$$

The atan2 function is defined as follows

$$\text{atan2}(b, a) = \begin{cases} \tan^{-1}\left(\frac{b}{a}\right) & \text{if } a > 0 \\ \tan^{-1}\left(\frac{b}{a}\right) + \pi & \text{if } a < 0 \\ \frac{\pi}{2} & \text{if } a = 0 \text{ and } b > 0 \\ -\frac{\pi}{2} & \text{if } a = 0 \text{ and } b < 0 \\ \text{undefined} & \text{if } a = 0 \text{ and } b = 0 \end{cases}$$

which basically means that the atan2 function keeps track of the signs of a and b . The signs of a and b indicate which quadrant the complex number is in. It then returns a value between $\pm\pi$. The \tan^{-1} function (not atan2) usually only gives a value between $\pm\frac{\pi}{2}$.

2.7.2 Conversions between Complex Number Forms

Complex numbers can be converted from one form to another. These conversions are presented here:

Rectangular Form \Rightarrow Trigonometric Form:

$$\begin{aligned} r &= \sqrt{a^2 + b^2} \\ \theta &= \text{atan2}(b, a) \\ z &= r \cos(\theta) + j r \sin(\theta) \end{aligned}$$

Rectangular Form \Rightarrow Exponential Form

$$\begin{aligned} r &= \sqrt{a^2 + b^2} \\ \theta &= \text{atan2}(b, a) \\ z &= r e^{j\theta} \end{aligned}$$

Trigonometric Form \Rightarrow Rectangular Form

$$\begin{aligned} a &= r \cos(\theta) \\ b &= r \sin(\theta) \\ z &= a + jb \end{aligned}$$

2.7 Transfer Functions as Complex Numbers

Trigonometric Form \Rightarrow Exponential Form

$$z = r \cos(\theta) + j r \sin(\theta)$$
$$z = r e^{j\theta}$$

Exponential Form \Rightarrow Rectangular Form

$$a = r \cos(\theta)$$
$$b = r \sin(\theta)$$
$$z = a + jb$$

Exponential Form \Rightarrow Trigonometric Form

$$z = r e^{j\theta}$$
$$z = r \cos(\theta) + j r \sin(\theta)$$

2.7.3 Mathematical Operations of Complex Numbers

Addition and subtraction is simplest in rectangular form because the real and imaginary components are already separated so that the real parts are added or subtracted together, then the imaginary parts are added or subtracted together:

$$z_1 = a_1 + jb_1$$
$$z_2 = a_2 + jb_2$$
$$z_1 + z_2 = (a_1 + a_2) + j(b_1 + b_2)$$
$$z_1 - z_2 = (a_1 - a_2) + j(b_1 - b_2)$$

Multiplication and division are simplest in exponential form because the radii are multiplied or divided, then the angles are added or subtracted, respectfully:

$$z_1 = r_1 e^{j\theta_1}$$
$$z_2 = r_2 e^{j\theta_2}$$
$$z_1 z_2 = r_1 r_2 e^{j(\theta_1 + \theta_2)}$$
$$\frac{z_1}{z_2} = \frac{r_1}{r_2} e^{j(\theta_1 - \theta_2)}$$

2.7.4 Magnitude and Phase Angle of Complex Fractions

If you are presented with a complex fraction with imaginary components in the denominator, the imaginary components in the denominator need to be removed before it can be broken into real and imaginary components, or to determine the magnitude and phase angle.

One approach to remove the imaginary component in the denominator is to multiply the numerator and the denominator by the complex conjugate.

A more efficient approach would be to convert the numerator and denominator to exponential form and then use the solution approach above. For example, consider the following complex number:

$$z = \frac{a + jb}{c + jd}$$

the numerator can be converted to exponential form as:

$$z_{num} = \sqrt{a^2 + b^2} e^{j\text{atan2}(b,a)} = r_{num} e^{j\theta_{num}}$$

the denominator can be converted to exponential form as:

$$z_{den} = \sqrt{c^2 + d^2} e^{j\text{atan2}(d,c)} = r_{den} e^{j\theta_{den}}$$

thus the complex fraction becomes:

$$\begin{aligned} z &= \frac{\sqrt{a^2 + b^2} e^{j\text{atan2}(b,a)}}{\sqrt{c^2 + d^2} e^{j\text{atan2}(d,c)}} \\ &= \frac{\sqrt{a^2 + b^2}}{\sqrt{c^2 + d^2}} e^{j(\text{atan2}(b,a) - \text{atan2}(d,c))} \end{aligned}$$

or:

$$z = \frac{r_{num} e^{j\theta_{num}}}{r_{den} e^{j\theta_{den}}} = \frac{r_{num}}{r_{den}} e^{j(\theta_{num} - \theta_{den})}$$

Thus, we can see from the above example that if we have a complex fraction, then the magnitude of the entire fraction is simply the magnitude of the numerator divided by the magnitude of the denominator:

$$r = \frac{r_{num}}{r_{den}}$$

Also, the phase angle of the entire fraction is simply the phase angle of the numerator minus the phase angle of the denominator:

$$\theta = \theta_{num} - \theta_{den}$$

If the real and imaginary values are then needed, they can be calculated from this trigonometric form.

2.7.5 Magnitude and Phase of a Transfer Function

By substituting $s = j\omega$, a transfer function becomes a complex fraction. We can therefore determine the magnitude and phase of a transfer function. The **magnitude of the transfer function** is:

$$r_{TF} = \frac{r_{num}}{r_{den}} = \frac{\sqrt{\text{Re}_{num}^2 + \text{Im}_{num}^2}}{\sqrt{\text{Re}_{den}^2 + \text{Im}_{den}^2}} \quad (2.53)$$

The **phase angle of the transfer function** is:

$$\theta_{TF} = \theta_{num} - \theta_{den} = \text{atan2}(\text{Im}_{num}, \text{Re}_{num}) - \text{atan2}(\text{Im}_{den}, \text{Re}_{den}) \quad (2.54)$$

2.8 Frequency Response Plots and Bode Plots

Magnitude and Phase of a Transfer Function

Example 2.7.1. Magnitude and phase of a transfer function

Given the following transfer function:

$$TF(j\omega) = \frac{s+2}{3s^2+2s+1}$$

Find its magnitude and phase angle as a function of ω .

Solution: Substituting $s = j\omega$, and realizing that $j^2 = -1$, the transfer function can be rewritten as follows:

$$TF(j\omega) = \frac{2 + j\omega}{(1 - 3\omega^2) + j(2\omega)}$$

The magnitude is the magnitude of the numerator divided by the magnitude of the denominator:

$$r_{TF} = \frac{\sqrt{2^2 + \omega^2}}{\sqrt{(1 - 3\omega^2)^2 + (2\omega)^2}}$$

and the phase angle is the angle of the numerator minus the angle of the denominator:

$$\theta_{TF} = \text{atan2}(\omega, 2) - \text{atan2}(2\omega, 1 - 3\omega^2)$$

2.8 Frequency Response Plots and Bode Plots

Frequency response plots and Bode plots are graphical ways to represent transfer functions. This section shows how to construct them analytically. Section 12.3.3 will show that they can be constructed from experimental data as well. Therefore, they are valuable for system identification and machine learning. They also aid feedback control design, as discussed in Section 13.6.

2.8.1 Frequency Response Plots

A **frequency response plot** contains two graphs: transfer function magnitude r_{TF} versus frequency and transfer function phase angle θ_{TF} versus frequency. The frequency is on the x-axis of the graph, and it is usually graphed in log-scale. Eq. (2.53) calculates the transfer function magnitude r_{TF} , which is plotted on the first graph. Eq. (2.54) calculates the transfer function phase angle θ_{TF} , which is plotted on the second graph.

Frequency Response Plot

Example 2.8.1. Frequency response plot of a transfer function

Create a frequency response plot of the transfer function of Example 2.4.2:

$$\frac{Y}{U} = \frac{s - 4}{(s + 3)(s + 5)}$$

Solution: Substituting $s = j\omega$, the transfer function becomes a complex function of ω :

$$\frac{Y}{U} = \frac{-4 + j\omega}{(15 - \omega^2) + j8\omega}$$

The magnitude r_{TF} of the transfer function is found from Eq. (2.53) to be

$$r_{TF} = \frac{\sqrt{4^2 + \omega^2}}{\sqrt{(15 - \omega^2)^2 + (8\omega)^2}}$$

The phase angle is found by applying Eq. (2.54):

$$\theta_{TF} = \text{atan2}(\omega, -4) - \text{atan2}(8\omega, 15 - \omega^2)$$

The following MATLAB script is used to create the frequency response plot:

```
close all %Close all open figures
clear all %Clear all variables and functions
clc %Clear the command window

% Get the frequency
w = logspace(-2,3,100); %log-scale frequencies from 0.01 to 1000

%Calculate the transfer function magnitude
rTF = sqrt(4^2+w.^2)./sqrt((15-w.^2).^2+(8*w).^2);

%Calculate the transfer function phase angle (deg)
thetaTF = atan2d(w,-4)-atan2d(8*w,15-w.^2);

%Create the frequency response plot
figure %open a figure for plotting
subplot(211) %Create 2x1 subplots, write on the first one
semilogx(w, rTF) %Plot of rTF versus w. The x-axis is log-scale
title('Frequency Response Plot') %Create a title
ylabel('Magnitude r_T_F') %Label the y-axis
grid on %Turn on the grid
subplot(212) %Create 2x1 subplots, write on the second one
semilogx(w,thetaTF) %Plot of thetaTF versus w. The x-axis is log-scale
ylabel('Phase Angle \theta_T_F (deg)') %Label the y-axis
xlabel('Frequency (rad/s)') %Label the x-axis
grid on %Turn on the grid
```

The resulting frequency response plot is shown below.

2.8 Frequency Response Plots and Bode Plots

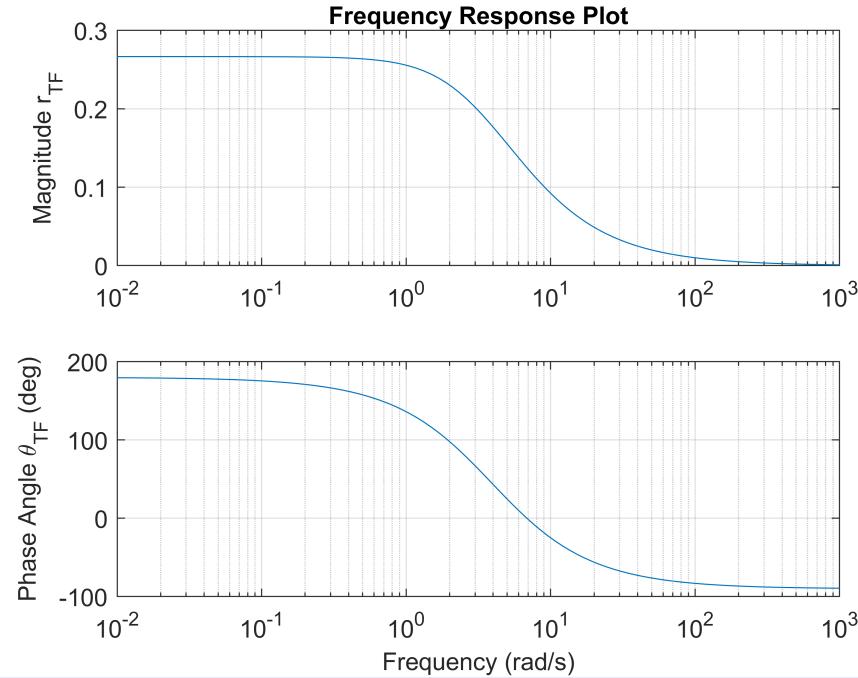


Figure 2.6 Frequency response plot of the transfer function

The frequency response plots show how the output of the transfer function behaves at steady-state in response to a sinusoidal input at a given frequency. For example, at a frequency of $\omega = 3$ rad/s, if the input U to the transfer function of Figure 2.6 was sinusoidal with an amplitude of one, the output Y would be sinusoidal and have an amplitude of 0.2. This is because the magnitude r_{TF} of the transfer function is 0.2 at $\omega = 3$ rad/s. If the input amplitude was 2, the output amplitude would be 0.4, etc. The phase angle graph shows that the output Y would be shifted from the input phase by a phase angle of approximately $\theta_{TF} = 65^\circ$ at the frequency $\omega = 3$ rad/s.

2.8.2 Bode Plots

A **Bode plot** is a type of frequency response plot where the transfer function magnitude is displayed in decibels (dB):

$$dB = 20 \log_{10} (r_{TF}) \quad (2.55)$$

That is the only difference. In other aspects, the Bode plot and the frequency response plot are the same.

Bode Plot

Example 2.8.2. Bode plot of a transfer function

Create a Bode plot of the transfer function of Examples 2.4.2 and 2.8.1:

$$\frac{Y}{U} = \frac{s - 4}{(s + 3)(s + 5)}$$

Solution: Substituting $s = j\omega$, Example 2.8.1 showed that the transfer function magnitude r_{TF} is

$$r_{TF} = \frac{\sqrt{4^2 + \omega^2}}{\sqrt{(15 - \omega^2)^2 + (8\omega)^2}}$$

The phase angle is

$$\theta_{TF} = \text{atan2}(\omega, -4) - \text{atan2}(8\omega, 15 - \omega^2)$$

The following MATLAB script is used to create the frequency response plot:

```
close all %Close all open figures
clear all %Clear all variables and functions
clc %Clear the command window

% Get the frequency
w = logspace(-2,3,100); %log-scale frequencies from 0.01 to 1000

%Calculate the transfer function magnitude
rTF = sqrt(4^2+w.^2)./sqrt((15-w.^2).^2+(8*w).^2);
dB = 20*log10(rTF);

%Calculate the transfer function phase angle (deg)
thetaTF = atan2d(w,-4)-atan2d(8*w,15-w.^2);

%Create the frequency response plot
figure %open a figure for plotting
subplot(211) %Create 2x1 subplots, write on the first one
semilogx(w, dB) %Plot of rTF versus w. The x-axis is log-scale
title('Bode Plot') %Create a title
ylabel('Magnitude dB') %Label the y-axis
grid on %Turn on the grid
subplot(212) %Create 2x1 subplots, write on the second one
semilogx(w,thetaTF) %Plot of thetaTF versus w. The x-axis is log-scale
ylabel('Phase Angle \theta_T_F (deg)') %Label the y-axis
xlabel('Frequency (rad/s)') %Label the x-axis
grid on %Turn on the grid
```

The resulting Bode plot is shown below.

2.8 Frequency Response Plots and Bode Plots

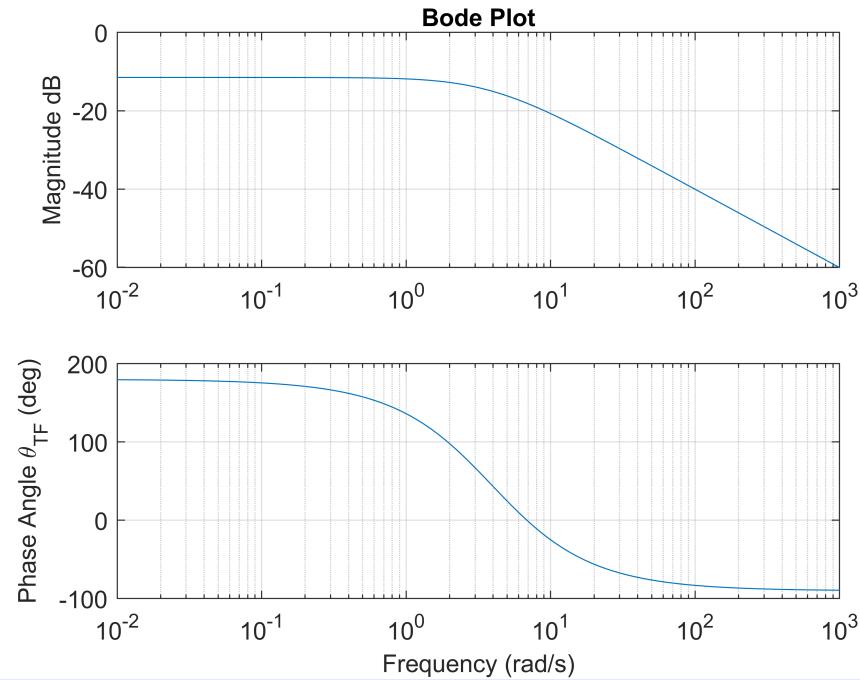


Figure 2.7 Bode plot of the transfer function

Bode plots can help identify properties of transfer functions. The magnitude plot changes its slope near pole locations, for example. This is demonstrated in Section 2.8.3, which draws rough Bode plots by hand. Bode plots can help determine the relative order between the polynomials in the numerator and denominator. This is because poles and zeros affect the limiting slope of the magnitude plot. The limiting slope is the slope in the limit as the frequency goes to infinity. Regardless of the pole's stability, each pole of a transfer function decreases the limiting slope of the magnitude by 20 dB/decade. Regardless of minimum or non-minimum phase, each zero adds 20 dB/decade to the limiting slope. The magnitude plot in Figure 2.7 has a limiting slope of -20 dB/decade because it has two poles (two roots in the denominator) and one zero (one root in the numerator). Without the zero, the Bode magnitude of Figure 2.7 would have a limiting slope of -40 dB/decade. This demonstrates one way that zeros can cancel the effect of poles.

Bode Magnitude Slopes and Relative Order

Example 2.8.3. Bode magnitude slopes and relative order of transfer functions

How many more poles than zeros does the transfer function represented by the following Bode plot magnitude have? Provide reasoning for your answer.

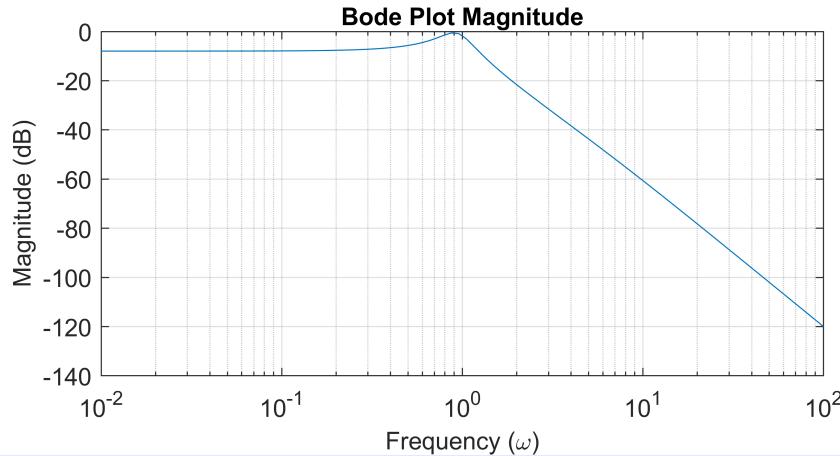


Figure 2.8 Bode plot magnitude

Solution: The magnitude is -60 dB at $\omega = 10^1$. One decade later at $\omega = 10^2$, the magnitude is -120 dB. Using rise over run to calculate the slope results in $\frac{-120 - (-60)}{\log_{10}(10^2) - \log_{10}(10^1)} = -60 \text{ dB/decade}$. Since each pole decreases the slope by 20 dB/decade, the transfer function has **three** more poles than it has zeros. (The transfer function is, in fact, $\frac{s-4}{s^4+8s^3+15s^2+11s+10}$).

2.8.3 Drawing Bode Magnitude Plots by Hand

Computers are able to create Bode plots from transfer functions by substituting $s = j\omega$ and following the procedures of Section 2.8.2. However, it is still useful to know how to create Bode plots by hand. One reason is that Bode plots can be used for system identification. Section 12.3.3 teaches how experimental data is used to create a Bode plot. The methods of this section can then be used to determine the poles and zeros of the governing transfer function. Once the transfer function is known, it can be converted to state-space using a canonical representation (see Section 2.2.3).

Transfer function numerators can be written as products of $(s - z_j)$, $j = 1, \dots, m$, where z_j are the transfer function zeros. Denominators can be written as products of $(s - p_i)$, $i = 1, \dots, n$, where p_i are the transfer function poles. Specifically, a transfer function $G(s)$ can be written in **pole-zero form** as follows:

$$G(s) = A \frac{(s - z_1)(s - z_2) \cdots (s - z_m)}{(s - p_1)(s - p_2) \cdots (s - p_n)} \quad (2.56)$$

where A is a constant scalar gain that can be positive or negative. Since creating Bode plots by hand is an approximate method anyways, note that the following approximation can be helpful for irreducible second-order polynomials:

$$as^2 + bs + c \approx (\sqrt{a}s + \sqrt{c})^2 \quad (2.57)$$

Bode-plots are log-scale graphs. Using the properties of logarithms, the logarithm of Eq. (2.56) becomes:

2.8 Frequency Response Plots and Bode Plots

$$20\log_{10}(|G(s)|) = 20\log_{10}\left(\left|A \frac{(s-z_1)(s-z_2)\cdots(s-z_m)}{(s-p_1)(s-p_2)\cdots(s-p_n)}\right|\right) \quad (2.58)$$

$$= 20\log_{10}(|A|) + \sum_{j=1}^m 20\log_{10}(|s-z_j|) - \sum_{i=1}^n 20\log_{10}(|s-p_i|) \quad (2.59)$$

From Eq. (2.59), we see that at each value of s where the frequency is a zero, or $\omega = |z_j|$, the slope of the Bode magnitude plot increases by 20 dB/decade. At each value of s where the frequency is a pole, or $\omega = |p_i|$, the slope of the Bode magnitude plot decreases by 20 dB/decade. The constant term $20\log_{10}(|A|)$ shifts the Bode magnitude plot up or down. The initial value $20\log_{10}(|r_{TF}(\omega_1)|)$ – see Eq. (2.53) – of the Bode magnitude plot at the initial frequency ω_1 must be calculated to know where to start the Bode magnitude plot. The following example demonstrates how to draw a Bode magnitude plot by hand.

Drawing Bode Magnitude Plots by Hand

Example 2.8.4. Construct a Bode magnitude plot by hand

Construct a Bode magnitude plot by hand of the following transfer function over the range of frequencies $\omega \in [10^{-2}, 10^3]$:

$$\frac{Y}{U} = \frac{-100s^3 + 980s^2 + 200s}{20s^4 + 796s^3 - 10543s^2 - 18919s - 88450}$$

Solution: The transfer function can be factored into pole-zero form as follows:

$$\frac{Y}{U} = -5 \frac{s(s+0.2)(s-10)}{(s+1+2.5j)(s+1-2.5j)(s-12.2)(s+50)}$$

Because of the complex poles, it may be better to use the second-order form: $(s+1+2.5j)(s+1-2.5j) = s^2 + 2s + 7.25$. For the handmade Bode plot, this can be approximated as $s^2 + 2s + 7.25 \approx (s+2.7)(s+2.7)$. With this approximation, the poles are

$$p_1 = -2.7$$

$$p_2 = -2.7$$

$$p_3 = 12.2$$

$$p_4 = -50$$

The zeros are

$$z_1 = 0$$

$$z_2 = -0.2$$

$$z_3 = 10$$

and the gain A is

$$A = -5$$

We want to graph the Bode plot over the frequencies $\omega \in [10^{-2}, 10^3]$; therefore, the initial frequency is $\omega_1 = 10^{-2}$. We need to determine the Bode plot magnitude at this frequency. To do so, we begin by substituting $s = j\omega_1 = j10^{-2}$ into the transfer function to get

$$\begin{aligned} \frac{Y}{U} &= \frac{-100(j10^{-2})^3 + 980(j10^{-2})^2 + 200(j10^{-2})}{20(j10^{-2})^4 + 796(j10^{-2})^3 - 10543(j10^{-2})^2 - 18919(j10^{-2}) - 88450} \\ &\approx \frac{-0.098 + 2j}{-88448.9 - 189.19j} \end{aligned}$$

The Bode magnitude plot will therefore begin at a value of

$$\begin{aligned} 20\log_{10}(|r_{TF}(\omega_1)|) &= 20\log_{10}\left(\left|\frac{-0.098 + 2j}{-88448.9 - 189.19j}\right|\right) \\ &= 20\log_{10}\left(\frac{\sqrt{0.098^2 + 2^2}}{\sqrt{88448.9^2 + 189.19^2}}\right) \\ &= -92.9 \text{ dB} \end{aligned}$$

We can order the poles and zeros from least to greatest by ignoring their \pm signs to get 0, 0.2, 2.7, 2.7, 10, 12.2, 50. At these frequencies, the slope of the Bode magnitude graph will change by ± 20

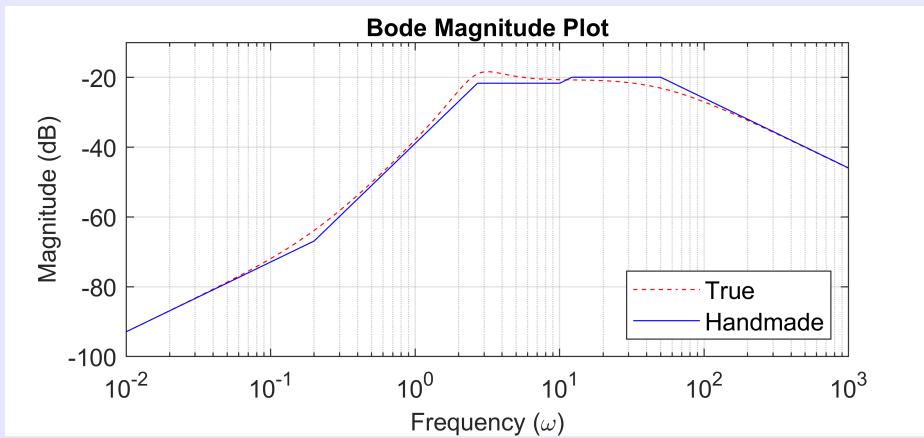


Figure 2.9 The handmade Bode magnitude plot approximates the true one

2.8 Frequency Response Plots and Bode Plots

2.8.4 Drawing Bode Phase Plots by Hand

Bode phase plots can also be approximated by hand. As done in Section 2.8.3, the transfer function can be written in pole-zero form as follows:

$$G(s) = A \frac{(s - z_1)(s - z_2) \cdots (s - z_m)}{(s - p_1)(s - p_2) \cdots (s - p_n)}$$

The approximation of Eq. (2.57) can again be helpful for irreducible second-order polynomials. It is instructive to observe the effect of a single pole or zero on the phase angle. For example, we will consider the effect of a zero z_j in $(s - z_j)$. We replace s with $s = j\omega$ to get $(j\omega - z_j)$. The phase angle is illustrated in Figure 2.10.

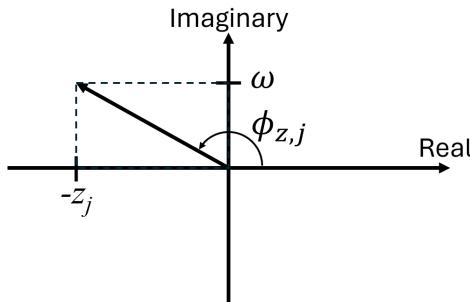


Figure 2.10 Phase angle of a positive (unstable or non-minimum phase) zero z_j

The phase angle $\phi_{z,j}$ is calculated from the positive real axis using the two-argument arctangent:

$$\phi_{z,j} = \text{atan2}(\omega, -z_j) \quad (2.60)$$

The zero z_j is constant, but the frequency ω varies from $0 \rightarrow \infty$. If the zero is positive, $z_j > 0$, its Bode phase plot is shown in Figure 2.11. The phase angle begins at 180° , but as the frequency ω approaches the zero $|z_j|$, the phase angle decreases. At z_j , the phase angle is equal to 135° . It continues decreasing and asymptotically approaches 90° as $\omega \rightarrow \infty$. As shown in the figure, a handmade phase plot for a single zero can be approximated by drawing a horizontal line at 180° from $\omega = 0 \rightarrow \frac{z_j}{10}$, then drawing a line with a slope of $-45^\circ/\text{decade}$ that decreases from 180° to 90° over the frequency range $\omega = \frac{z_j}{10} \rightarrow 10z_j$. Another horizontal line is drawn at 90° from $\omega = 10z_j \rightarrow \infty$.

The complete Bode phase plot is made by first making a phase plot for each pole and zero individually, then adding them together. The following steps are used to make a phase plot for each pole. For the i^{th} pole,

- For a stable pole, $p_i \leq 0$, begin at 0° deg. Decrease the slope by $45^\circ/\text{decade}$ beginning one decade before, i.e., at $\omega = \frac{|p_i|}{10}$. Return to a slope of $0^\circ/\text{decade}$ one decade after, i.e., at $\omega = 10|p_i|$. The total change in phase angle should be -90° from 0 to -90° deg.
- For an unstable pole, $p_i > 0$, begin at 180° deg. Increase the slope by $45^\circ/\text{decade}$ beginning one decade before, i.e., at $\omega = \frac{p_i}{10}$. Return to a slope of $0^\circ/\text{decade}$ one decade after, i.e., at $\omega = 10p_i$. The total change in phase angle should be $+90^\circ$ from 180 to 270 deg. (Notice that since

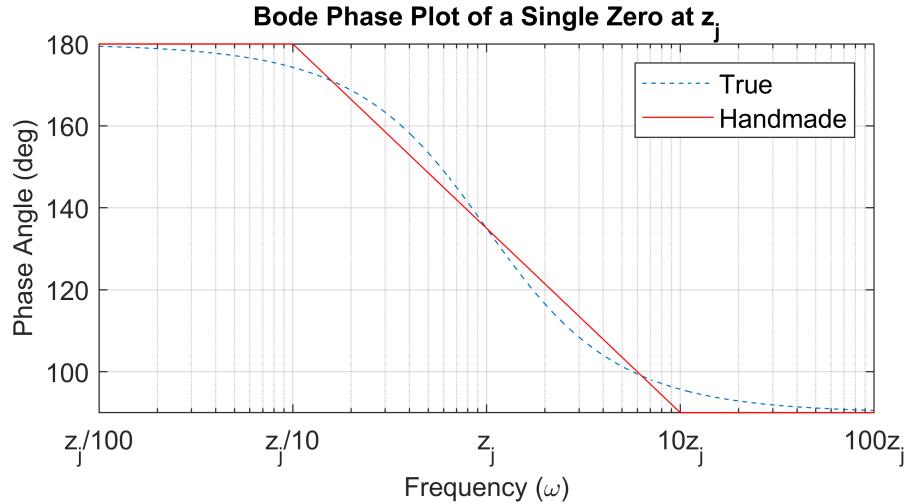


Figure 2.11 Bode phase plot of a positive (unstable or non-minimum phase) zero z_j

$\theta = \theta \pm n360^\circ$ where n is a positive integer, an equivalent Bode phase diagram would be to change from -180° to -90° .)

The following steps are used to make a phase plot for each zero. For the j^{th} zero,

1. For a minimum phase (stable) zero, $z_j \leq 0$, begin at 0 deg. Increase the slope by 45 deg/decade beginning one decade before, *i.e.*, at $\omega = \frac{|z_j|}{10}$. Return to a slope of 0 deg/decade one decade after, *i.e.*, at $\omega = 10|z_j|$. The total change in phase angle should be +90 deg from 0 to 90 deg.
2. For a non-minimum phase (unstable) zero, $z_j > 0$, begin at 180 deg. Decrease the slope by 45 deg/decade beginning one decade before, *i.e.*, at $\omega = \frac{z_j}{10}$. Return to a slope of 0 deg/decade one decade after, *i.e.*, at $\omega = 10z_j$. The total change in phase angle should be -90 deg from 180 deg to 90 deg.

After creating a Bode phase plot for each pole and zero individually, add all of them together to create the final Bode phase plot for the overall transfer function. If the transfer function gain A is negative, add -180 deg to the final phase plot.

2.8.5 Natural Frequency and Damping Ratios on Bode Plots

To demonstrate the effect of natural frequencies and damping ratios on Bode plots, consider a second order ODE system of the form:

$$\ddot{x} + 2\zeta\omega_n\dot{x} + \omega_n^2x = f(t)$$

where ζ is the damping ratio, ω_n is the natural frequency, and the input is a sinusoidal input oscillating at a frequency of ω :

$$f(t) = A \sin(\omega t)$$

2.8 Frequency Response Plots and Bode Plots

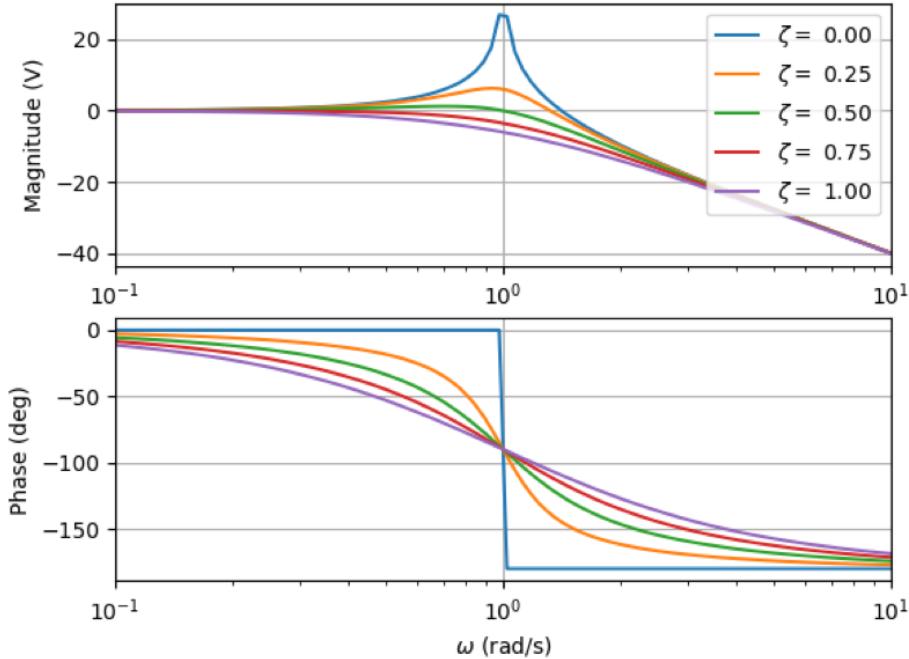


Figure 2.12 Bode plot of a second order system at various damping ratios, ζ .

Once again the output will oscillate at the same input frequency, but will only differ from the input in magnitude and phase:

$$x(t) = A \cdot r_{TF} \sin(\omega t + \theta_{TF})$$

The magnitude and phase are calculated from Eq. (2.53) and Eq. (2.54). In this case, the magnitude and phase will be:

$$r_{TF} = \frac{1/\omega_n^2}{\sqrt{\left(1 - \frac{\omega^2}{\omega_n^2}\right)^2 + \left(2\zeta\frac{\omega}{\omega_n}\right)^2}}$$

$$\theta_{TF} = -\text{atan2}\left(2\zeta\frac{\omega}{\omega_n}, 1 - \frac{\omega^2}{\omega_n^2}\right)$$

It can be seen in the above equations that both the damping ratio ζ and the natural frequency ω_n will affect the magnitude and phase of the output.

A Bode plot of the second order ODE can be made using various values of the damping ratio ζ and by setting the natural frequency to $\omega_n = 1$ rad/s, as shown in Figure 2.12, from which we can observe the following:

- The value of ζ affects the peak of the magnitude plot. The resonance peak occurs at the resonance frequency

$$\omega_R = \omega_n \sqrt{1 - 2\zeta^2},$$

which is found by setting the derivative of r_{TF} with respect to ω equal to zero. The peak magnitude will be

$$r_{TF}(\omega_R) = \frac{1}{2\omega_n\zeta\sqrt{1-\zeta^2}}.$$

- Values of ζ below 0.707 will have a raised peak.
- When $\zeta = 0$ the magnitude plot goes to infinity at the natural frequency, indicating that a second order ODE with no damping will resonate when the input frequency equals the natural frequency.
- The magnitude plot also appears to follow two asymptotes.
- One asymptote is flat.
- The second has a slope that decreases -40 dB between 1 and 10 rad/s. This is twice the change in dB as the first order Bode plot.
- The two asymptotes intersect at 1 rad/s, which corresponds to the natural frequency ω_n of the second order ODE for this example. The cutoff frequency for a second order Bode plot is equal to the natural frequency

$$\omega_c = \omega_n$$

which in this example is:

$$\omega_c = \omega_n = 1 \text{ rad/s}$$

- The phase plot changes from 0° to -180° .
- At the cutoff frequency the phase shift is half way between its net change of 180° , or -90° .

2.9 Nyquist Plots of Transfer Functions

A Nyquist plot displays the same information as a Bode plot or frequency response plot; however, a Nyquist plot shows both the phase and magnitude information on a single graph. The Nyquist plot is graphed in the complex plane. The x-axis is the real axis and the y-axis is the imaginary axis. The magnitude is the distance from the origin to any point on the Nyquist curve, and the phase angle is the corresponding angle from the positive real axis. Nyquist plots are often graphed as a function of the phase angle in the range of 0 to 2π radians. The following example illustrates how the Bode and Nyquist plots show the same information in different ways.

Nyquist Plots

Example 2.9.1. Nyquist plots

Compare the Nyquist and frequency response plots for the transfer function

$$\frac{Y}{U} = \frac{1}{s + 0.1}$$

Solution: To create the Nyquist and frequency response plots, we first make the substitution $j\omega \rightarrow s$

2.9 Nyquist Plots of Transfer Functions

in the transfer function to get

$$\frac{Y}{U} = \frac{1}{j\omega + 0.1}$$

Next, we convert the numerator and denominator to exponential form:

$$\begin{aligned}\frac{Y}{U} &= \frac{1e^{j0}}{\sqrt{0.01 + \omega^2} e^{j\tan^{-1} \frac{\omega}{0.1}}} \\ &= \frac{1}{\sqrt{0.01 + \omega^2}} e^{-j\tan^{-1}(10\omega)}\end{aligned}$$

The magnitude of the transfer function is

$$r_{TF} = \frac{1}{\sqrt{0.01 + \omega^2}}$$

and the angle is

$$\theta_{TF} = -\tan^{-1}(10\omega)$$

For the Nyquist plot, the angle ranges from 0 to 2π , and by solving the phase angle equation above for the frequency ω , we find that the frequency ranges from $-\infty$ to ∞ . The Bode plot only shows information for positive frequencies $\omega > 0$. To find the Nyquist data, we can calculate the real part of the Nyquist curve using the equation

$$\text{Real} = r_{TF} \cos(\theta_{TF})$$

The imaginary data is calculated by

$$\text{Imag} = r_{TF} \sin(\theta_{TF})$$

The following MATLAB code was used to create the frequency response plot of Figure 2.13 and the Nyquist plot in Figure 2.14. Observe that the highlighted points on the Nyquist plot have the same magnitude and phase as the corresponding points on the frequency response plot.

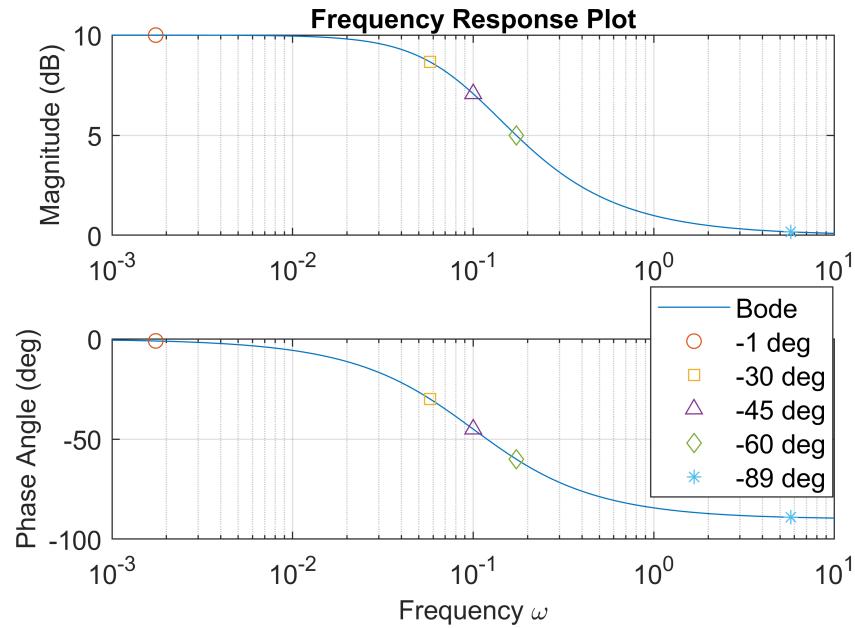


Figure 2.13 Frequency Response Plot

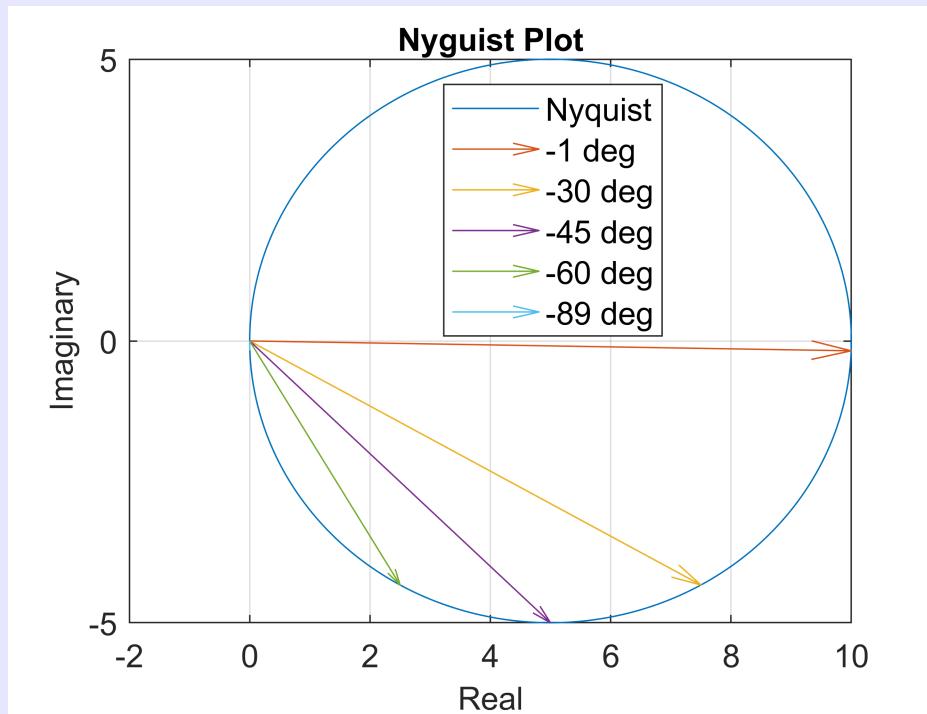


Figure 2.14 Nyquist Plot

2.9 Nyquist Plots of Transfer Functions

```
close all %close all open figures
clear all %clear all variables and functions
clc %clear the command window

%let the angle range from 0 to 2*pi radians
theta = 0:0.01:2*pi; %Transfer function angle
w = tan(-theta)/10; %solve the phase angle equation for omega
rTF = 1./sqrt(0.01+w.^2); %get the transfer function magnitude
thetaTF = -atan(10*w);
Real = rTF.*cos(thetaTF); %The real part of the nyquist plot
Imag = rTF.*sin(thetaTF); %The imaginary part of the nyquist plot
%do it again for the angles 0, -pi/6, -pi/4, -pi/3, and -pi/2
theta_A = -1*pi/180; wA = tan(-theta_A)/10; rTF_A = 1./sqrt(0.01+wA.^2);
Real_A = rTF_A.*cos(theta_A); Imag_A = rTF_A.*sin(theta_A);
theta_B = -pi/6; wB = tan(-theta_B)/10; rTF_B = 1./sqrt(0.01+wB.^2);
Real_B = rTF_B.*cos(theta_B); Imag_B = rTF_B.*sin(theta_B);
theta_C = -pi/4; wC = tan(-theta_C)/10; rTF_C = 1./sqrt(0.01+wC.^2);
Real_C = rTF_C.*cos(theta_C); Imag_C = rTF_C.*sin(theta_C);
theta_D = -pi/3; wD = tan(-theta_D)/10; rTF_D = 1./sqrt(0.01+wD.^2);
Real_D = rTF_D.*cos(theta_D); Imag_D = rTF_D.*sin(theta_D);
theta_E = -89*pi/180; wE = tan(-theta_E)/10; rTF_E = 1./sqrt(0.01+wE.^2);
Real_E = rTF_E.*cos(theta_E); Imag_E = rTF_E.*sin(theta_E);

figure, plot(Real, Imag)%Create the nyquist plot
hold on
quiver(0,0,Real_A,Imag_A,0) %Draw arrows to the specific points
quiver(0,0,Real_B,Imag_B,0)
quiver(0,0,Real_C,Imag_C,0)
quiver(0,0,Real_D,Imag_D,0)
quiver(0,0,Real_E,Imag_E)
hold off
legend('Nyquist','-1 deg','-30 deg','-45 deg','-60 deg',' -89 deg')
title('Nyquist Plot') %Give the figure a title
xlabel('Real') %label the x-axis
ylabel('Imaginary') %label the y-axis
grid on

%create a Frequency Response plot,
%(MATLAB will ignore any negative frequencies and give a warning)
figure, %Create a new figure to draw on
subplot(211) %Create a subplot
semilogx(w, (rTF),...
wA, (rTF_A), 'o',...
wB, (rTF_B), 's',...
wC, (rTF_C), '^',...
wD, (rTF_D), 'd',...
wE, (rTF_E), '*') %Draw the frequency response magnitude
title('Frequency Response Plot') %give it a title
ylabel('Magnitude (dB)') %Label the y-axis
xlim([10^-3, 10]) %set the graph x-limits
grid on %draw grid-lines on the graph
subplot(212) %Create a second subplot
semilogx(w,thetaTF*180/pi,...
```

```
wA, theta_A*180/pi,'o',...
wB, theta_B*180/pi,'s',...
wC, theta_C*180/pi,'^',...
wD, theta_D*180/pi,'d',...
wE, theta_E*180/pi,'*') %Draw the phase angle
legend('Bode',' -1 deg', '-30 deg', '-45 deg', '-60 deg', '-89 deg')
ylabel('Phase Angle (deg)') %Label the y-axis
xlabel('Frequency \omega') %Label the x-axis
xlim([10^-3, 10]) %set the graph x-limits
grid on %draw grid-lines on the graph
```

Chapter 3

Discrete-Time Linear System Modeling

Contents

3.1	Discrete State Space	89
3.1.1	Difference Equations	90
3.1.2	The Z-Transform and Transfer Functions	92
3.1.3	Zero-Pole-Gain Form of a Z-Domain Transfer Function	94
3.1.4	Converting Between Laplace and Z-Domain Transfer Functions	94
3.1.5	Conversion from Discrete State-Space to a Difference Equation	97
3.1.6	Converting Difference Equations to Discrete State-Space	100
3.1.7	Stability of Discrete-Time Systems	102
3.2	Converting Discrete to Continuous State-Space	104

3.1 Discrete State Space

Some dynamic systems are inherently discrete in time, such as digital networks and computers. Others are continuous in time, but must be converted to discrete-time to be solved with a digital computer. Section 1.3 explained how to convert continuous state-space systems to discrete-time so they can be solved with a computer. Whether inherently discrete-time, or converted to discrete-time, linear systems can be written in **discrete state-space form** as

$$y_k = Cx_k + Du_k \quad (3.1)$$

$$x_{k+1} = A_d x_k + B_d u_k \quad (3.2)$$

Table 1.2 provided various techniques to convert from continuous LTI state-space systems to discrete state-space systems. That table is provided again here (see Table 3.1) for convenience. The output C and feed-through D matrices are the same in both continuous and discrete-time systems; no conversion is needed. However, the continuous state-matrix A becomes the discrete state-transition matrix A_d , and the continuous input matrix B is converted to the discrete input-transition matrix B_d . The conversion from continuous to discrete-time is most accurate when the piecewise exact method of Table 3.1 is used.

Table 3.1 State- and Input-Transition Matrices for Numerical Methods Ordered from Most to Least Accurate

Method	State-Transition Matrix A_d	Input-Transition Matrix B_d
Piecewise Exact Method	$A_d = \text{expm}\{A\Delta t\} = \sum_{k=0}^{\infty} \frac{1}{k!} (A\Delta t)^k$	$\begin{bmatrix} A_d & B_d \\ \mathbf{0} & I \end{bmatrix} = \text{expm}\left\{ \begin{bmatrix} A\Delta t & B\Delta t \\ \mathbf{0} & \mathbf{0} \end{bmatrix} \right\}$
4th Order Runge-Kutta	$A_d = \sum_{k=0}^4 \frac{1}{k!} (A\Delta t)^k$	$B_d = \left(\sum_{k=0}^3 \frac{1}{(k+1)!} (A\Delta t)^k \right) B\Delta t$
Tustin's Method	$A_d = \left(I - \frac{\Delta t}{2} A \right)^{-1} \left(I + \frac{\Delta t}{2} A \right)$	$B_d = \left(I - \frac{\Delta t}{2} A \right)^{-1} B\Delta t$
Backward Euler	$A_d = (I - A\Delta t)^{-1}$	$B_d = (I - A\Delta t)^{-1} B\Delta t$
Forward Euler	$A_d = I + A\Delta t$	$B_d = B\Delta t$

3.1.1 Difference Equations

Continuous time systems can be written as ordinary differential equations (ODEs). Similarly, discrete-time systems can be written as difference equations. A continuous time ODE has the form

$$x^{(n)} + a_{n-1}x^{(n-1)} + \cdots + a_2\ddot{x} + a_1\dot{x} + a_0x = b_m u^{(m)} + b_{m-1}u^{(m-1)} + \cdots + b_1\dot{u} + b_0u$$

whereas a **discrete-time difference equation** has the form:

$$y_k + a_1y_{k-1} + \cdots + a_{n-1}y_{k-(n-1)} + a_ny_{k-n} = b_0u_k + b_1u_{k-1} + \cdots + b_mu_{k-m} \quad (3.3)$$

where y_k is the value of the output y at time t_k , y_{k-1} is the value of y at time t_{k-1} , and so forth. u_k is the value of the input u at time t_k , u_{k-1} is the value of u at time t_{k-1} , etc. The coefficients a_1, \dots, a_n and b_0, \dots, b_m are scalar constants from the set of real numbers.

We must know previous values of the output y_{k-1}, \dots, y_{k-n} and the input u_k, \dots, u_{k-m} to solve a difference equation. This is demonstrated in the following example.

Numerically Simulating A Difference Equation

Example 3.1.1. Difference Equations

3.1 Discrete State Space

Simulate the difference equation

$$y_k = -0.8y_{k-1} + 0.3y_{k-2} + 0.1u_k - 0.8u_{k-1}$$

where

$$u = [1, 2.5, 3, 1.5, 1]$$

starting at $k = 0$. The initial conditions are

$$y_{-1} = 0$$

$$y_{-2} = 1$$

$$u_{-1} = 2$$

Plot the output trajectory y on the same graph as the input u .

Solution: We must find the trajectory of y given the input u and initial conditions. We will do this in MATLAB using the following code:

```
%set the initial conditions
y1 = 0;
y2 = 1;
u1 = 2;

%set the input
u = [1,2.5,3,1.5,1];
N = length(u);

%allocate memory to store the output
y = zeros(1,N);
kVec = 0:N-1;

for ii=1:N
    %get the value of y
    y(ii) = -0.8*y1+0.3*y2+0.1*u(ii)-0.8*u1;
    %store the previous values of y and u
    y2 = y1;
    y1 = y(ii);
    u1 = u(ii);
end
%plot the output
plot(kVec, u, '--', kVec, y)
legend('u','y')
xlabel('k ')
grid on
```

The resulting graph is

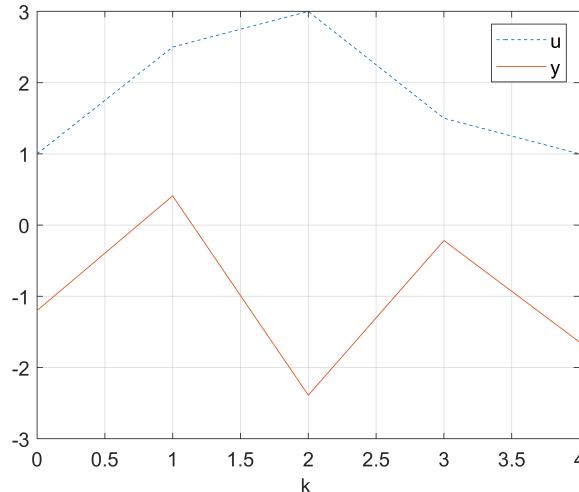


Figure 3.1 Trajectory of the difference equation output.

A discrete-time system can be converted from state-space form to a difference equation and vice-versa. To do so, however, we will first introduce the Z-transform and Z-domain transfer functions.

3.1.2 The Z-Transform and Transfer Functions

For continuous-time systems, we applied the Laplace transform to find transfer functions in the s-domain (Laplace domain). Discrete-time systems can be written as transfer functions in the Z-domain via the Z-transform. The Z-transform is the discrete-time equivalent of the Laplace transform.

Multiplying a signal X by the variable s in the Laplace domain was equivalent to taking the derivative of the signal with respect to time in the time domain:

$$\mathcal{L}^{-1}\{sX\} = \dot{x}$$

Multiplying a signal X by z in the Z-domain is equivalent to a time-advance in the time-domain:

$$Z^{-1}\{zX\} = x_{k+1}$$

or

$$Z\{x_{k+1}\} = zX - zx_0$$

where x_0 is the initial condition. Other properties of the Z-transform are shown in Table 3.2.

Applying the Z-transform to the difference equation Eq. (3.3) results in

$$Y + a_1z^{-1}Y + \cdots + a_{n-1}z^{-(n-1)}Y + a_nz^{-n}Y = b_0U + b_1z^{-1}U + \cdots + b_mz^{-m}U \quad (3.4)$$

or

$$Y(1 + a_1z^{-1} + \cdots + a_{n-1}z^{-(n-1)} + a_nz^{-n}) = U(b_0 + b_1z^{-1} + \cdots + b_mz^{-m}) \quad (3.5)$$

3.1 Discrete State Space

Table 3.2 Table of Z-transform properties used in this book (ignoring initial conditions)

Time Domain	Z-Domain
$x_k = Z^{-1}\{X\}$	$X = Z\{x_k\}$
$x_{k+n} = Z^{-1}\{z^n X\}$	$Z\{x_{k+n}\} = z^n X$
$x_{k-n} = Z^{-1}\{z^{-n} X\}$	$Z\{x_{k-n}\} = z^{-n} X$
$a_1 x_k + a_2 y_k = Z^{-1}\{a_1 X + a_2 Y\}$	$Z\{a_1 x_k + a_2 y_k\} = a_1 X + a_2 Y$

We can now rearrange Eq. (3.5) into transfer function form:

$$\frac{Y(z)}{U(z)} = \frac{b_0 + b_1 z^{-1} + \dots + b_m z^{-m}}{1 + a_1 z^{-1} + \dots + a_n z^{-n}} \quad (3.6)$$

The following example converts a difference equation to a Z-domain transfer function.

Difference Equation to Z-Domain Transfer Function

Example 3.1.2. Converting a difference equation to a Z-domain transfer function

Convert the difference equation

$$y_k = -0.8y_{k-1} + 0.3y_{k-2} + 4u_{k-1}$$

to a discrete-time transfer function.

Solution: We apply the Z-transform to both sides of the equation. Using the properties of Z-transforms in Table 3.2, we find that

$$Y = -0.8z^{-1}Y + 0.3z^{-2}Y + 4z^{-1}U$$

We move all of the terms with Y to the left-hand side of the equation and factor out a Y :

$$Y(1 + 0.8z^{-1} - 0.3z^{-2}) = 4z^{-1}U$$

Now we can arrange the equation into transfer function form

$$\frac{Y}{U} = \frac{4z^{-1}}{1 + 0.8z^{-1} - 0.3z^{-2}}$$

and the example is complete.

A Z-domain transfer function can also be converted to a difference equation, as shown in the following example.

Z-Domain Transfer Function to Difference Equation

Example 3.1.3. Converting a Z-domain transfer function to a difference equation

Convert the Z-domain transfer function

$$\frac{Y}{U} = \frac{5 + 4z + 3z^2}{6 + 2z + 7z^2}$$

to a difference equation.

Solution: Our first step is to multiply the numerator and denominator by $z^{-2}/7$

$$\frac{Y}{U} = \frac{(5 + 4z + 3z^2)(z^{-2}/7)}{(6 + 2z + 7z^2)(z^{-2}/7)}$$

to get

$$\frac{Y}{U} = \frac{5/7z^{-2} + 4/7z^{-1} + 3/7}{6/7z^{-2} + 2/7z^{-1} + 1}$$

Now, we cross-multiply to get

$$Y(1 + 2/7z^{-1} + 6/7z^{-2}) = U(3/7 + 4/7z^{-1} + 5/7z^{-2})$$

Finally, we apply the inverse Z-transform (see the properties of Table 3.2) to get

$$y_k + 2/7y_{k-1} + 6/7y_{k-2} = 3/7u_k + 4/7u_{k-1} + 5/7u_{k-2}$$

and the example is complete.

3.1.3 Zero-Pole-Gain Form of a Z-Domain Transfer Function

As with Laplace transfer functions (see Section 2.5), a Z-domain transfer function can be converted to zero-pole-gain form:

$$\frac{Y}{U} = k_d \frac{(z - z_{d,1})(z - z_{d,2}) \cdots (z - z_{d,m})}{(z - p_{d,1})(z - p_{d,2}) \cdots (z - p_{d,n})} \quad (3.7)$$

$$= k_d \frac{\prod_{j=1}^m (z - z_{d,j})}{\prod_{i=1}^n (z - p_{d,i})} \quad (3.8)$$

where each $z_{d,j}$, $j = 1, 2, \dots, m$ is a zero of the transfer function, and each $p_{d,i}$, $i = 1, 2, \dots, n$ is a pole. The transfer function gain is k_d . It is *not* the steady-state or DC gain of the transfer function. The **steady-state or DC gain** of a Z-domain transfer function $G(z)$ is found by $\lim_{z \rightarrow 1} G(z)$.

3.1.4 Converting Between Laplace and Z-Domain Transfer Functions

The Laplace transform of a time shift by Δt is $e^{s\Delta t}$. Multiplying by z in the Z-domain causes a time-shift in the time-domain. Conversion between Laplace transfer functions and Z-domain transfer functions is accomplished by applying the substitution

$$z = e^{s\Delta t} \quad (3.9)$$

3.1 Discrete State Space

where Δt is the sampling time-step. If the zero-pole-gain form is used, Eq. (3.9) can be used to convert continuous-time poles and zeros to discrete-time and vice-versa. The Z-domain transfer function gain k_d is related to the Laplace transfer function gain k by the following equation:

$$k_d \frac{\prod_{j=1}^m (1 - z_{d,j})}{\prod_{i=1}^n (1 - p_{d,i})} = k \frac{\prod_{j=1}^m (0 - z_j)}{\prod_{i=1}^n (0 - p_i)} \quad (3.10)$$

Eq. (3.10) is valid as long as no discrete-time poles or zeros are at $z = 1$ and no continuous-time poles or zeros are at $s = 0$. Alternatively, sometimes the **Tustin approximation**

$$s \approx \frac{2(z-1)}{\Delta t(z+1)} \text{ or } z \approx \frac{1+s\Delta t/2}{1-s\Delta t/2} \quad (3.11)$$

is used.

Convert from Laplace Transfer Function Using the Tustin Approximation

Example 3.1.4. Convert a Laplace transfer function to discrete-time using the Tustin Approximation

A mass-spring-damper system can be modeled by the following continuous-time state-space equation (see Eq. (6.1)).

$$\begin{aligned}\dot{x} &= \begin{bmatrix} 0 & 1 \\ -\frac{k}{m} & -\frac{b}{m} \end{bmatrix} x + \begin{bmatrix} 0 \\ \frac{1}{m} \end{bmatrix} u \\ y &= \begin{bmatrix} 1 & 0 \end{bmatrix} x\end{aligned}$$

Consider a mass-spring-damper system with the following parameters:

$$\begin{aligned}m &= 5 \text{ kg} \\ b &= 3 \text{ kg/s} \\ k &= 24 \text{ N/m}\end{aligned}$$

Using a time-step of $\Delta t = 0.01$ s, convert the continuous time state-space equations to a Laplace transfer function. Then use the Tustin approximation to convert it to a Z-domain transfer function. Compare the transfer function using the Tustin approximation with the exact Z-domain transfer function as calculated in Example 3.1.5.

Solution: Plugging the values of m , b , and k into the continuous-time state-space equations gives

$$\begin{aligned}\dot{x} &= \begin{bmatrix} 0 & 1 \\ -4.8 & -0.6 \end{bmatrix} x + \begin{bmatrix} 0 \\ 0.2 \end{bmatrix} u \\ y &= \begin{bmatrix} 1 & 0 \end{bmatrix} x\end{aligned}$$

The Laplace transfer function is found by using Eq. (2.5)

$$\begin{aligned}
 \frac{Y}{U} &= C(sI - A)^{-1} B + D \\
 &= \begin{bmatrix} 1 & 0 \end{bmatrix} \left(\begin{bmatrix} s & 0 \\ 0 & s \end{bmatrix} - \begin{bmatrix} 0 & 1 \\ -4.8 & -0.6 \end{bmatrix} \right)^{-1} \begin{bmatrix} 0 \\ 0.2 \end{bmatrix} \\
 &= \frac{1}{5s^2 + 3s + 24}
 \end{aligned}$$

Using the Tustin approximation, we replace each s in the transfer function with $s = \frac{2(z-1)}{\Delta t(z+1)}$:

$$\frac{Y}{U} \approx \frac{1}{5 \left(\frac{2(z-1)}{\Delta t(z+1)} \right)^2 + 3 \left(\frac{2(z-1)}{\Delta t(z+1)} \right) + 24}$$

Simplifying, we get

$$\frac{Y}{U} \approx \frac{4.9844E-06z^2 + 9.9689E-06z + 4.9844E-06}{z^2 - 1.9935z + 0.994}$$

where $(E-06) = 10^{-6}$. Multiplying the numerator and denominator by z^{-2} results in the desired Z-domain transfer function:

$$\frac{Y}{U} \approx \frac{4.9844E-06 + 9.9689E-06z^{-1} + 4.9844E-06z^{-2}}{1 - 1.9935z^{-1} + 0.994z^{-2}} \quad (3.12)$$

Comparing this Tustin approximation in Eq. (3.12) with the exact Z-domain transfer function of Eq. (3.21), we see that both transfer functions have the same denominator. The numerators, however, are different. One difference is that the numerator in the Tustin approximation has an additional feed-through term $4.9844E-06$ that is not in the exact transfer function. This feed-through term causes a time-shift in the time-domain. The other difference is in the coefficients for the z^{-2} terms in the numerator. The z^{-1} numerator coefficients for both transfer functions are the same to within 3 significant digits.

Using the input forcing function

$$u = 7 \sin\left(\frac{2}{100}t^2 + t\right)$$

the response of the exact transfer function and the Tustin approximation are plotted as a function of time in the following figure:

3.1 Discrete State Space

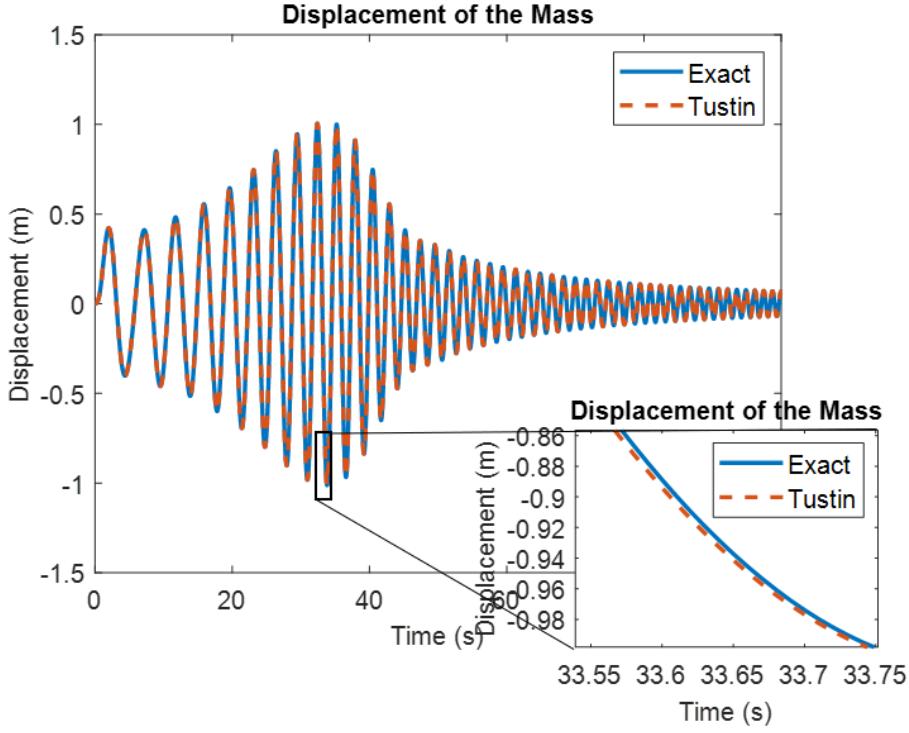


Figure 3.2 Tustin approximation versus the exact solution

By zooming in, we can see the effect of the time-shift caused by the feed-through term in the numerator.

3.1.5 Conversion from Discrete State-Space to a Difference Equation

With an understanding of the Z-transform, we can now introduce a method to convert from a discrete state-space equation to a difference equation. To do so, we first use the Z-transform to convert the discrete state-space equations to a Z-domain transfer function. The Z-transform of the discrete state-equation Eq. (3.2) is

$$Z\{x_{k+1} = A_d x_k + B_d u_k\} \quad (3.13)$$

$$= zX - zx_0 = A_d X + B_d U \quad (3.14)$$

Eq. (3.14) can be rearranged as follows:

$$(zI - A_d) X = B_d U + zx_0 \quad (3.15)$$

Solving for X gives

$$X = (zI - A_d)^{-1} B_d U + (zI - A_d)^{-1} zx_0 \quad (3.16)$$

Now we convert the discrete output equation Eq. (3.1) to the Z-domain:

$$Z\{y_k = Cx_k + Du_k\} \quad (3.17)$$

$$= Y = CX + DU \quad (3.18)$$

Using Eq. (3.16) as a substitution for X in Eq. (3.18) provides

$$Y = (C(zI - A_d)^{-1} B_d + D) U + C(zI - A_d)^{-1} z x_0 \quad (3.19)$$

If we ignore initial conditions, the Z-domain transfer function from U to Y is

$$\frac{Y}{U} = (C(zI - A_d)^{-1} B_d + D) \quad (3.20)$$

Comparing this with the Laplace transform Eq. (2.5) of a continuous time transfer function, we can see how the Z-transform is the discrete-time equivalent of the Laplace transform. We can use the methods described in Section 3.1.2 to convert Eq. (3.20) to a difference equation. This is demonstrated in the following example.

Continuous-to-Discrete-Time Conversions

Example 3.1.5. Converting continuous-time systems to discrete-time

A mass-spring-damper system can be modeled by the following continuous-time state-space equation (see Eq. (6.1)).

$$\begin{aligned}\dot{x} &= \begin{bmatrix} 0 & 1 \\ -\frac{k}{m} & -\frac{b}{m} \end{bmatrix} x + \begin{bmatrix} 0 \\ \frac{1}{m} \end{bmatrix} u \\ y &= \begin{bmatrix} 1 & 0 \end{bmatrix} x\end{aligned}$$

Consider a mass-spring-damper system with the following parameters:

$$m = 5 \text{ kg}$$

$$b = 3 \text{ kg/s}$$

$$k = 24 \text{ N/m}$$

Using a time-step of $\Delta t = 0.01$ s, convert the continuous time state-space equations to discrete state-space equations. Next, convert the discrete state-space equations to a Z-domain transfer function. Finally, convert the Z-domain transfer function to a difference equation.

Solution: Plugging the values of m , b , and k into the continuous-time state-space equations gives

$$\begin{aligned}\dot{x} &= \begin{bmatrix} 0 & 1 \\ -4.8 & -0.6 \end{bmatrix} x + \begin{bmatrix} 0 \\ 0.2 \end{bmatrix} u \\ y &= \begin{bmatrix} 1 & 0 \end{bmatrix} x\end{aligned}$$

3.1 Discrete State Space

To convert to discrete state-space, we use the matrix exponential equation of the piecewise exact method of Table 3.1.

$$\exp\left(\begin{bmatrix} \begin{bmatrix} 0 & 1 \\ -4.8 & -0.6 \\ \begin{bmatrix} 0 & 0 \end{bmatrix} & \begin{bmatrix} 0 \\ 0.2 \\ 0 \end{bmatrix} \end{bmatrix} 0.01 & \begin{bmatrix} 0 \\ 0.01 \\ 0 \end{bmatrix} \end{bmatrix}\right) = \begin{bmatrix} \begin{bmatrix} 0.9998 & 0.01 \\ -0.0479 & 0.9938 \\ \begin{bmatrix} 0 & 0 \end{bmatrix} & \begin{bmatrix} 0 \\ 0.002 \\ 1 \end{bmatrix} \end{bmatrix}$$

to find that

$$A_d = \begin{bmatrix} 0.9998 & 0.01 \\ -0.0479 & 0.9938 \end{bmatrix}$$

and

$$B_d = \begin{bmatrix} 0 \\ 0.002 \end{bmatrix}$$

Converting from continuous time to discrete time does not change the C or D output matrices. Therefore, the discrete state-space equations are

$$\begin{aligned} x_{k+1} &= \begin{bmatrix} 0.9998 & 0.01 \\ -0.0479 & 0.9938 \end{bmatrix} x_k + \begin{bmatrix} 0 \\ 0.002 \end{bmatrix} u_k \\ y_k &= \begin{bmatrix} 1 & 0 \end{bmatrix} x_k \end{aligned}$$

We must now convert the discrete state-space equations to a Z-domain transfer function. Using Eq. (3.20), we get

$$\begin{aligned} \frac{Y}{U} &= \begin{bmatrix} 1 & 0 \end{bmatrix} \left(\begin{bmatrix} z & 0 \\ 0 & z \end{bmatrix} - \begin{bmatrix} 0.9998 & 0.01 \\ -0.0479 & 0.9938 \end{bmatrix} \right)^{-1} \begin{bmatrix} 0 \\ 0.002 \end{bmatrix} \\ &= \frac{(9.9796E-06)z + (9.9597E-06)}{z^2 - 1.9935z + 0.994} \end{aligned}$$

where $(E-06) = 10^{-6}$.

The final step is to convert the transfer function to a difference equation. We begin by multiplying the numerator and denominator by z^{-2} to get

$$\frac{Y}{U} = \frac{(9.9796E-06)z^{-1} + (9.9597E-06)z^{-2}}{1 - 1.9935z^{-1} + 0.994z^{-2}} \quad (3.21)$$

We can cross-multiply to get

$$Y(1 - 1.9935z^{-1} + 0.994z^{-2}) = U((9.9796E-06)z^{-1} + (9.9597E-06)z^{-2})$$

Finally, we distribute Y and U and take the inverse Z-transform to get

$$y_k - 1.9935y_{k-1} + 0.994y_{k-2} = (9.9796E-06)u_{k-1} + (9.9597E-06)u_{k-2}$$

and the example is complete.

3.1.6 Converting Difference Equations to Discrete State-Space

The previous section described how to convert discrete state-space equations to difference equations. This section explains the reverse: how to convert a difference equation to state-space. To do so, this section uses the Z-transform (see Section 3.1.2) to convert the difference equation to a transfer function. Then it uses the derivation of the controllable canonical form to convert the Z-domain transfer function to discrete state-space equations.

Applying the Z-transform to a linear difference equation of the form

$$y_k + a_1 y_{k-1} + \cdots + a_{n-1} y_{k-n+1} + a_n y_{k-n} = b_0 u_k + b_1 u_{k-1} + \cdots + b_m u_{k-m} \quad (3.22)$$

and rearranging, we can derive the following transfer function.

$$\frac{Y}{U} = \frac{b_0 + b_1 z^{-1} + \cdots + b_m z^{-m}}{1 + a_1 z^{-1} + \cdots + a_n z^{-n}} \quad (3.23)$$

Before continuing, we make the following definitions:

$$N = \max(n, m) \quad (3.24)$$

$$a_i = 0 \text{ for } i = n+1, \dots, N \text{ if } m > n \quad (3.25)$$

$$b_i = 0 \text{ for } i = m+1, \dots, N \text{ if } n > m \quad (3.26)$$

Using these definitions, we can rewrite Eq. (3.23) to have the same order in the numerator and the denominator.

$$Y = \frac{b_0 + b_1 z^{-1} + \cdots + b_N z^{-N}}{1 + a_1 z^{-1} + \cdots + a_N z^{-N}} U \quad (3.27)$$

Now we make one more definition

$$X = \frac{1}{1 + a_1 z^{-1} + \cdots + a_N z^{-N}} U \quad (3.28)$$

so that we can write Eq. (3.27) as

$$Y = (b_0 + b_1 z^{-1} + \cdots + b_N z^{-N}) X \quad (3.29)$$

From Eq. (3.28), we can multiply both sides by the denominator $1 + a_1 z^{-1} + \cdots + a_N z^{-N}$ and take the inverse Z-transform to get the following difference equation:

$$x_k + a_1 x_{k-1} + \cdots + a_N x_{k-N} = u_k \quad (3.30)$$

3.1 Discrete State Space

Now we can define the discrete state variables to be

$$q_k = \begin{bmatrix} x_{k-N} \\ x_{k-N+1} \\ \vdots \\ x_{k-1} \end{bmatrix} \quad (3.31)$$

so that we get

$$q_{k+1} = \begin{bmatrix} x_{k-N+1} \\ x_{k-N+2} \\ \vdots \\ x_k \end{bmatrix} \quad (3.32)$$

If we solve Eq. (3.30) for x_k , we get

$$x_k = -a_1 x_{k-1} - \cdots - a_N x_{k-N} + u_k \quad (3.33)$$

and we can write q_{k+1} in terms of the state q_k :

$$q_{k+1} = \begin{bmatrix} 0 & 1 & 0 & \dots & 0 \\ 0 & 0 & 1 & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \dots & 1 \\ -a_N & -a_{N-1} & -a_{N-2} & \dots & -a_1 \end{bmatrix} q_k + \begin{bmatrix} 0 \\ 0 \\ \vdots \\ 0 \\ 1 \end{bmatrix} u_k \quad (3.34)$$

We have derived the state equation. Now we will use Eq. (3.29) to derive the output equation. We take the inverse Z-transform of Eq. (3.29) to get

$$y_k = b_0 x_k + b_1 x_{k-1} + \cdots + b_N x_{k-N} \quad (3.35)$$

Although x_{k-N}, \dots, x_{k-1} are state variables (they are in q_k), x_k is not a state variable. Fortunately, Eq. (3.33) solved for x_k in terms of the state variables. We substitute x_k from Eq. (3.33) into Eq. (3.35) to get

$$y_k = (b_1 - a_1 b_0) x_{k-1} + (b_2 - a_2 b_0) x_{k-2} + \cdots + (b_N - a_N b_0) x_{k-N} + b_0 u_k \quad (3.36)$$

Which can be written in the output equation form:

$$y_k = [(b_N - a_N b_0) \quad \dots \quad (b_2 - a_2 b_0) \quad (b_1 - a_1 b_0)] q_k + b_0 u_k \quad (3.37)$$

Summary: Conversion from Difference to Discrete State-Space

Step 1: Use the Z-transform to convert the difference equation

$$y_k + a_1 y_{k-1} + \cdots + a_{n-1} y_{k-n+1} + a_n y_{k-n} = b_0 u_k + b_1 u_{k-1} + \cdots + b_m u_{k-m}$$

to a Z-domain transfer function

$$Y = \frac{b_0 + b_1 z^{-1} + \cdots + b_N z^{-N}}{1 + a_1 z^{-1} + \cdots + a_N z^{-N}} U$$

where

$$N = \max(n, m)$$

$$a_i = 0 \text{ for } i = n+1, \dots, N \text{ if } m > n$$

$$b_i = 0 \text{ for } i = m+1, \dots, N \text{ if } n > m$$

Step 2: Arrange the coefficients a_1, \dots, a_N and b_0, \dots, b_N into the controllable canonical form matrices:

$$q_{k+1} = \begin{bmatrix} 0 & 1 & 0 & \dots & 0 \\ 0 & 0 & 1 & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \dots & 1 \\ -a_N & -a_{N-1} & -a_{N-2} & \dots & -a_1 \end{bmatrix} q_k + \begin{bmatrix} 0 \\ 0 \\ \vdots \\ 0 \\ 1 \end{bmatrix} u_k$$

$$y_k = [(b_N - a_N b_0) \quad \dots \quad (b_2 - a_2 b_0) \quad (b_1 - a_1 b_0)] q_k + b_0 u_k$$

3.1.7 Stability of Discrete-Time Systems

In continuous-time systems, the eigenvalues of the state-matrix A of $\dot{x} = Ax + Bu$ determined the stability of the system (see Section 1.6). If the eigenvalues were all on the left-hand side of the complex plane, the system was stable. Likewise, we can use the eigenvalues of the state-transition matrix A_d in $x_{k+1} = A_d x_k + B_d u_k$ to determine the stability of a discrete-time system; however, the discrete-time stability criteria are different than for continuous-time systems.

BIBO (Bounded Input Bounded Output) stability means that if the absolute value of the input never exceeds some finite constant, then the absolute value of the output will also never exceed some other finite constant. In a BIBO stable system, a bounded input will always result in a bounded output. The criteria for a linear discrete-time system to be BIBO stable are summarized below.

BIBO Stability Criteria for Discrete-Time Systems

Consider a linear, discrete-time, system in discrete state-space form

3.1 Discrete State Space

$$y_k = Cx_k + Du_k$$

$$x_{k+1} = A_d x_k + B_d u_k$$

If the linear, discrete-time, system is not in this form, refer to Section 3.1.6 for instructions on how to convert it to this form. The eigenvalues (λ) of the state-transition A_d matrix are the roots of the characteristic equation

$$\det(\lambda I - A_d) = 0$$

If all eigenvalues of A_d are within the unit circle, $|\lambda| < 1$, in the complex plane, then the discrete-time system is **BIBO stable**. If any eigenvalue of A_d is outside of the unit circle $|\lambda| > 1$, the discrete-time system is **unstable**. If any eigenvalue of A_d lies on the unit circle, $|\lambda| = 1$, and no eigenvalue is outside the unit circle, the system is **marginally stable**.

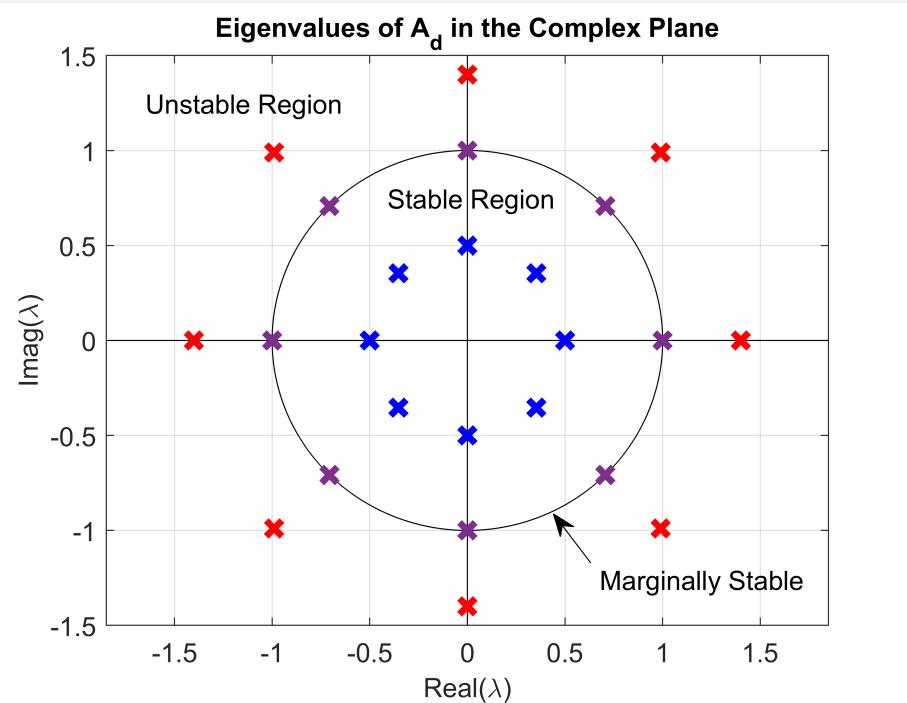


Figure 3.3 Stability Regions in the Complex Plane for Discrete Systems

Mapping of Continuous-Time Eigenvalues to Discrete-Time Eigenvalues

If the exact solution (see Eq. (1.23)) is used to convert a continuous-time system to a discrete-time system, we can use Eq. (3.9) ($z = e^{s\Delta t}$) to map eigenvalues (s) of continuous-time systems to discrete-time eigenvalues (z). Note that a stable, continuous-time, eigenvalue ($\text{Real}(s) < 0$) maps to a stable, discrete-time, eigenvalue ($|z| < 1$) because $z = e^{s\Delta t}$ is within the unit circle if $\text{Real}(s) < 0$. An unstable, continuous-time, eigenvalue maps to an unstable, discrete-time, eigenvalue because $z = e^{s\Delta t}$ is outside the unit circle if

$\text{Real}(s) > 0$. A marginally stable, continuous-time, eigenvalue maps to a marginally stable, discrete-time, eigenvalue because $z = e^{s\Delta t}$ is on the unit circle if $\text{Real}(s) = 0$.

Discrete Transient Response Based on Eigenvalue Location

As with continuous-time systems (see Figure 1.4), we can qualitatively determine the transient response of discrete-time systems based on the location of the eigenvalues. The response is largely determined by the eigenvalue with the largest complex magnitude. We refer to it as the dominant eigenvalue: $|\lambda_{dom}| = \max_i |\lambda_i|$. Notice that BIBO stability can be determined from the dominant eigenvalue. If $|\lambda_{dom}| < 1$, the system is stable. If $|\lambda_{dom}| > 1$ the system is unstable. If $|\lambda_{dom}| = 1$ the system is marginally stable.

Whether the system oscillates or not is also largely determined by λ_{dom} . If λ_{dom} is real, and $\lambda_{dom} > 0$, the system will not oscillate. If λ_{dom} is real, and $\lambda_{dom} < 0$ the system will oscillate back and forth between positive and negative values. If λ_{dom} has a non-zero imaginary value, the system will oscillate. Qualitative responses based on the dominant eigenvalue are shown in Figure 3.4.

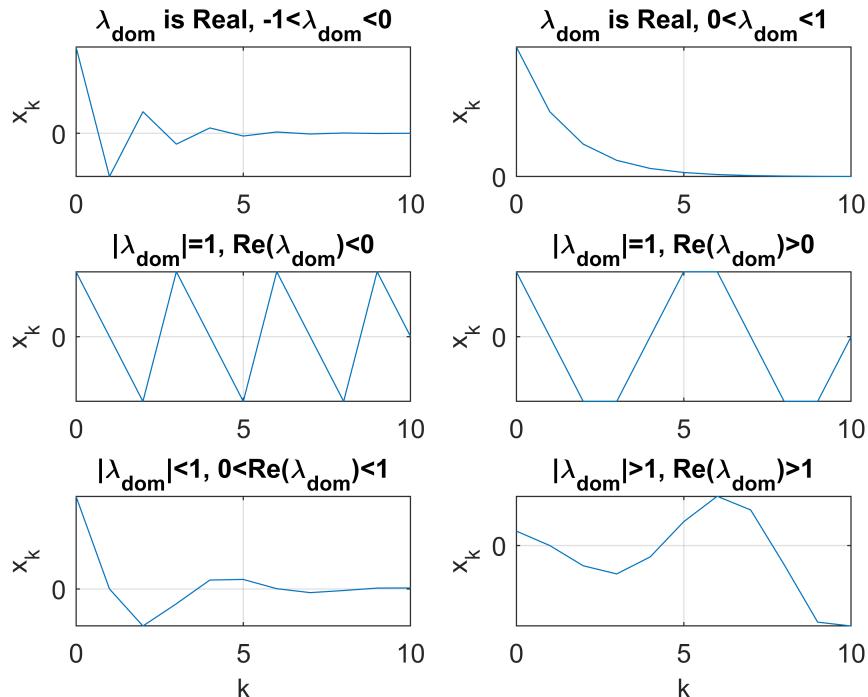


Figure 3.4 Transient responses of discrete-time systems based on pole locations

3.2 Converting Discrete to Continuous State-Space

Table 3.1 provided equations that convert continuous state-space matrices A and B to discrete state-space A_d and B_d . In some cases, the inverse relationships of Table 3.1 can convert discrete state-space to continuous state-space. For example, inverting the forward Euler approximation shows that $A =$

3.2 Converting Discrete to Continuous State-Space

$\frac{1}{\Delta t}(A_d - I)$ and $B = \frac{1}{\Delta t}B_d$. Since the matrix logarithm is often not well defined for some state-space models of physical systems, the inverse of the piecewise exact method may not be a practical approach. Instead, an indirect method can be used to find the piecewise exact solution. The indirect method first converts the discrete state-space system to the zero-pole-gain form of a Z-domain transfer function (see Section 3.1.3). It then converts it to the zero-pole-gain form of the Laplace transfer function using $z = \exp(s\Delta t)$ or $s = \ln(z)/\Delta t$ (which works when there are no negative real discrete-time poles or zeros). Finally, it uses a canonical state-space form to find the continuous state-space system. This indirect approach is demonstrated in Example 3.2.1.

Common direct methods for converting from discrete to continuous state-space are summarized in Table 3.3. The Tustin method is the most accurate as long as the state-transition matrix A_d has no eigenvalues at -1 or 1 .

Table 3.3 Equations to Convert from Discrete to Continuous State-Space

Method	Continuous State-Space
Tustin's Method	$A = \frac{2}{\Delta t}(A_d - I)(A_d + I)^{-1}$ $B = \frac{2}{\Delta t}(I - (A_d - I)(A_d + I)^{-1})B_d$ $C = C_d(A_d + I)^{-1}$ $D = D_d - C_d(A_d + I)^{-1}B_d$
Backward Euler	$A = \frac{1}{\Delta t}(I - A_d^{-1})$ $B = \frac{1}{\Delta t}(I - A_d^{-1})B_d$ $C = C_dA_d^{-1}$ $D = D_d - C_dA_d^{-1}B_d$
Forward Euler	$A = \frac{1}{\Delta t}(A_d - I)$ $B = \frac{1}{\Delta t}B_d$ $C = C_d$ $D = D_d$

The following example demonstrates converting from discrete to continuous state-space using the indirect piecewise exact method. The piecewise exact method (also known as the zero-order-hold method) is more accurate than the Tustin method when there are no negative real poles or zeros.

Conversion from Discrete to Continuous State-Space using the Piecewise Exact Method

Example 3.2.1. Conversion from Discrete to Continuous State-Space

Convert the discrete state-space system

$$\begin{aligned}x_{k+1} &= \begin{bmatrix} 0.8 & 0.1 \\ -0.1 & 0.9 \end{bmatrix} x_k + \begin{bmatrix} 0.1 \\ 0.2 \end{bmatrix} u_k \\y_k &= \begin{bmatrix} 1 & 0 \end{bmatrix} x_k\end{aligned}$$

where $\Delta t = 0.1$ s to continuous state-space using the piecewise exact solution.

Solution: To convert from discrete to continuous state-space, this solution first finds the Z-domain transfer function using Eq. (3.20), $\frac{Y}{U} = C_d(zI - A_d)^{-1}B_d + D_d$:

$$\begin{aligned}\frac{Y}{U} &= \begin{bmatrix} 1 & 0 \end{bmatrix} \left(\begin{bmatrix} z & 0 \\ 0 & z \end{bmatrix} - \begin{bmatrix} 0.8 & 0.1 \\ -0.1 & 0.9 \end{bmatrix} \right)^{-1} \begin{bmatrix} 0.1 \\ 0.2 \end{bmatrix} \\&= \frac{0.1z - 0.07}{z^2 - 1.7z + 0.73}\end{aligned}$$

Next, this solution converts the Z-domain transfer function to its zero-pole-gain form. The numerator has one root at $z = 0.7$, so the transfer function zero is $z_{d,1} = 0.7$. The denominator has two roots, which are the poles. The quadratic formula determines them to be the complex conjugate pair $p_{d,1} = 0.85 + j0.0866$ and $p_{d,2} = 0.85 - j0.0866$. The gain is the coefficient (0.1) of the highest-power numerator term divided by the coefficient (1) of the highest-power denominator term:

$$\begin{aligned}k_d &= \frac{0.1}{1} \\&= 0.1\end{aligned}$$

The zero-pole-gain form is therefore

$$\frac{0.1z - 0.07}{z^2 - 1.7z + 0.73} = 0.1 \frac{z - 0.7}{(z - 0.85 - j0.0866)(z - 0.85 + j0.0866)}$$

Now, this solution uses $z = \exp(s\Delta t)$, or $s = \frac{\ln(z)}{\Delta t}$, to convert the discrete-time poles and zeros to continuous time:

$$\begin{aligned}z_{d,1} &= 0.7 \rightarrow z_1 = -3.5667 \\p_{d,1} &= 0.85 + j0.0866 \rightarrow p_1 = -1.5736 + j1.0153 \\p_{d,2} &= 0.85 - j0.0866 \rightarrow p_1 = -1.5736 - j1.0153\end{aligned}$$

The gain k of the Laplace transfer function is found from Eq. (3.10):

3.2 Converting Discrete to Continuous State-Space

$$\begin{aligned}
k &= k_d \frac{1 - z_{d,1}}{(1 - p_{d,1})(1 - p_{d,2})} \frac{(0 - p_1)(0 - p_2)}{0 - z_1} \\
&= 0.1 \frac{1 - 0.7}{(1 - 0.85 - j0.0866)(1 - 0.85 + j0.0866)} \frac{(0 + 1.5736 - j1.0153)(0 + 1.5736 + j1.0153)}{0 + 3.5667} \\
&= 0.9833
\end{aligned}$$

Knowing the zero (z_1), poles (p_1 and p_2), and gain (k) of the Laplace transform allows us to write the continuous time transfer function in its zero-pole-gain form:

$$\begin{aligned}
\frac{Y}{U} &= k \frac{s - z_1}{(s - p_1)(s - p_2)} \\
&= 0.9833 \frac{s + 3.5667}{(s + 1.5736 + j1.0153)(s + 1.5736 - j1.0153)}
\end{aligned}$$

which can be simplified to its general form:

$$\frac{Y}{U} = \frac{0.9833s + 3.507}{s^2 + 3.1471s + 3.507}$$

The final step is to use controllable canonical form, see Section 2.2.3, to convert the transfer function to its continuous state-space representation:

$$\begin{aligned}
\dot{x} &= \begin{bmatrix} 0 & 1 \\ -3.507 & -3.1471 \end{bmatrix} x + \begin{bmatrix} 0 \\ 1 \end{bmatrix} u \\
y &= \begin{bmatrix} 3.507 & 0.9983 \end{bmatrix} x
\end{aligned}$$

and the solution is complete.

The following examples provide proofs for the backward Euler and Tustin methods.

Proof of the Backward Euler Method of Table 3.3

Example 3.2.2. Derive the backward Euler method to convert discrete to continuous state-space Table 3.3 claims that if the backward Euler method is used, the discrete state-space system

$$\begin{aligned}
x_{k+1} &= A_d x_k + B_d u_k \\
y_k &= C_d x_k + D_d u_k
\end{aligned}$$

can be converted to continuous state-space

$$\begin{aligned}
\dot{x} &= Ax + Bu \\
y &= Cx + Du
\end{aligned}$$

using the following conversions

$$\begin{aligned} A &= \frac{1}{\Delta t} (I - A_d^{-1}) \\ B &= \frac{1}{\Delta t} (I - A_d^{-1}) B_d \\ C &= C_d A_d^{-1} \\ D &= D_d - C_d A_d^{-1} B_d \end{aligned}$$

where Δt is the timestep. Prove this claim.

Solution: The proof is given in two steps. The first step is to show for the backward Euler method that the conversion from a discrete pole z to Laplace pole s is

$$z = \frac{1}{1 - s\Delta t} \quad (3.38)$$

The backward Euler method can be written as follows:

$$\frac{x_{k+1} - x_k}{\Delta t} = Ax_{k+1} + Bu_{k+1}$$

and its Z-transform is

$$\frac{zX - X}{\Delta t} = AzX + BzU$$

Grouping z terms, it becomes

$$z((I - A\Delta t)X - B\Delta tU) = X \quad (3.39)$$

The proof will return to Eq. (3.39) shortly, but first it will calculate the Laplace transform of $\dot{x} = Ax + Bu$, which is

$$sX = AX + BU$$

We can multiply each term by Δt to get

$$\Delta tsX = A\Delta tX + B\Delta tU$$

Then we subtract both sides of the equation from X to get

$$X - \Delta tsX = X - A\Delta tX - B\Delta tU$$

which can be factored as follows

$$(1 - s\Delta t)X = (I - A\Delta t)X - B\Delta tU$$

3.2 Converting Discrete to Continuous State-Space

or

$$\frac{1}{1-s\Delta t} ((I - A\Delta t)X - B\Delta tU) = X$$

Compared with Eq. (3.39), we see for the backward Euler method that

$$z = \frac{1}{1-s\Delta t}$$

and the first part of the proof is complete. The second part of the proof begins with the Z-transform of the discrete state-transition equation $x_{k+1} = A_d x_k + B_d u_k$ which is

$$zX = A_d X + B_d U$$

Using the backward Euler substitution $z = \frac{1}{1-s\Delta t}$ produces

$$\frac{1}{1-s\Delta t} X = A_d X + B_d U$$

which can be rearranged grouping s terms as follows

$$s(A_d X + B_d U) = \frac{1}{\Delta t} ((A_d - I)X + B_d U) \quad (3.40)$$

If we define $\tilde{X} = A_d X + B_d U$, then $X = A_d^{-1}(\tilde{X} - B_d U)$, and Eq. (3.40) can be written as

$$s\tilde{X} = \frac{1}{\Delta t} ((A_d - I)A_d^{-1}(\tilde{X} - B_d U) + B_d U)$$

whose inverse Laplace transform can be written as follows

$$\dot{\tilde{x}} = \frac{1}{\Delta t} (A_d - I) A_d^{-1} \tilde{x} + \frac{1}{\Delta t} (I - A_d^{-1}) B_d u$$

The state redefinition $x = A_d^{-1}(\tilde{x} - B_d u)$ must also be substituted into the output equation $y = C_d x + D_d u$ to get

$$y = C_d A_d^{-1} \tilde{x} + (D_d - C_d A_d^{-1} B_d) u$$

Therefore, the continuous state-space system is

$$\dot{\tilde{x}} = A\tilde{x} + Bu$$

$$y = C\tilde{x} + Du$$

Where the matrices are

$$\begin{aligned} A &= \frac{1}{\Delta t} (I - A_d^{-1}) \\ B &= \frac{1}{\Delta t} (I - A_d^{-1}) B_d \\ C &= C_d A_d^{-1} \\ D &= D_d - C_d A_d^{-1} B_d \end{aligned}$$

and the proof is complete.

The next example provides the proof for the Tustin method.

Proof of the Tustin Method of Table 3.3

Example 3.2.3. Prove the Tustin method of Table 3.3 given $z = \frac{1+s\Delta t/2}{1-s\Delta t/2}$
Given the Tustin approximation

$$z = \frac{1 + s\Delta t/2}{1 - s\Delta t/2} \quad (3.41)$$

prove that the discrete to continuous state-space conversion results in the matrices

$$\begin{aligned} A &= \frac{2}{\Delta t} (A_d - I) (A_d + I)^{-1} \\ B &= \frac{2}{\Delta t} (I - (A_d - I) (A_d + I)^{-1}) B_d \\ C &= C_d (A_d + I)^{-1} \\ D &= D_d - C_d (A_d + I)^{-1} B_d \end{aligned}$$

Solution: The Z-transform of the discrete state equation $x_{k+1} = A_d x_k + B_d u_k$ is

$$zX = A_d X + B_d U$$

Substituting $z = \frac{1+s\Delta t/2}{1-s\Delta t/2}$ yields

$$\frac{1 + s\Delta t/2}{1 - s\Delta t/2} X = A_d X + B_d U$$

which, after some algebraic manipulation to group s terms, becomes

$$s((I + A_d)X + B_d U) = \frac{2}{\Delta t} (A_d - I)X + \frac{2}{\Delta t} B_d U \quad (3.42)$$

If we define $\tilde{X} = (I + A_d)X + B_d U$, we can solve for X to get

$$X = (I + A_d)^{-1} (\tilde{X} - B_d U)$$

Substituting X back into Eq. (3.42) yields

$$s\tilde{X} = \frac{2}{\Delta t} (A_d - I) (I + A_d)^{-1} \tilde{X} + \frac{2}{\Delta t} (I - (A_d - I) (I + A_d)^{-1}) B_d U$$

The state redefinition $x = (I + A_d)^{-1} (\tilde{X} - B_d U)$ must also be substituted into the output equation $y = C_d x + D_d u$ to get

$$y = C_d (I + A_d)^{-1} \tilde{X} + (D_d - C_d (I + A_d)^{-1} B_d) u$$

3.2 Converting Discrete to Continuous State-Space

Therefore, the continuous state-space system is

$$\begin{aligned}\dot{\tilde{x}} &= A\tilde{x} + Bu \\ y &= C\tilde{x} + Du\end{aligned}$$

Where the matrices are

$$\begin{aligned}A &= \frac{2}{\Delta t} (A_d - I) (I + A_d)^{-1} \\ B &= \frac{2}{\Delta t} \left(I - (A_d - I) (I + A_d)^{-1} \right) B_d \\ C &= C_d (I + A_d)^{-1} \\ D &= D_d - C_d (I + A_d)^{-1} B_d\end{aligned}$$

and the proof is complete.

Chapter 4

Numerical Solutions of Nonlinear Systems

Contents

4.1	The Euler Method for Multivariable $\dot{x} = f(x, u)$	113
4.2	Converting Nonlinear ODEs to $\dot{x} = f(x, u)$	114
4.2.1	Simulation Examples	116
4.3	The Runge-Kutta Method for Multivariable $\dot{x} = f(x, u)$	120
4.3.1	Simulation Examples	121
4.4	Linearizing Nonlinear Systems	127
4.4.1	Taylor Series Expansions	127
4.4.2	Multivariable Taylor Series Expansion	127
4.4.3	Linearization of $\dot{x} = f(x, u)$ around x^* and u^*	127
4.4.4	Linearization around Constant x^* and u^*	130
4.4.5	Linearization around the Values $x^* = x_k$ and $u^* = u_k$	130

As presented in Section 1.2, a state-space equation can have the linear form $\dot{x} = Ax + Bu$ or it can be nonlinear. This chapter introduces numerical methods for solving nonlinear state-space equations that are in the form $\dot{x} = f(x, u)$. It discusses methods for converting nonlinear equations to the form $\dot{x} = f(x, u)$. It converts continuous-time nonlinear systems to discrete-time so they can be numerically solved or simulated with a computer. It also provides a method using Taylor series expansions to linearize nonlinear equations so they can be solved using the methods of Chapter 1.

4.1 The Euler Method for Multivariable $\dot{x} = f(x, u)$

Perhaps the easiest way to numerically solve a differential equation in the form $\dot{x} = f(x, u)$ is to use the Euler method. To use the Euler method, we replace the derivative \dot{x} with its approximation

$$\dot{x} \approx \frac{x_{k+1} - x_k}{\Delta t} \quad (4.1)$$

Then the Euler method solves one step in time from t_k to t_{k+1} to get x_{k+1} as follows

$$x_{k+1} \approx x_k + \Delta t f(x_k, u_k) \quad (4.2)$$

We replaced x by its value x_k at the discrete time t_k . We also replaced u by its value u_k at time t_k . The time-step Δt is the difference between time t_{k+1} and t_k .

$$\Delta t = t_{k+1} - t_k \quad (4.3)$$

The Euler method applies to multivariable state-equations with n state variables and p inputs of the form

$$\begin{bmatrix} \dot{x}_1 \\ \dot{x}_2 \\ \vdots \\ \dot{x}_n \end{bmatrix} = \begin{bmatrix} f_1(x_1, \dots, x_n, u_1, \dots, u_p) \\ f_2(x_1, \dots, x_n, u_1, \dots, u_p) \\ \vdots \\ f_n(x_1, \dots, x_n, u_1, \dots, u_p) \end{bmatrix} \quad (4.4)$$

The multivariable Euler method applies Eq. (4.2) to each state equation individually to get

$$\begin{bmatrix} x_{1,k+1} \\ x_{2,k+1} \\ \vdots \\ x_{n,k+1} \end{bmatrix} \approx \begin{bmatrix} x_{1,k} \\ x_{2,k} \\ \vdots \\ x_{n,k} \end{bmatrix} + \Delta t \begin{bmatrix} f_1(x_{1,k}, \dots, x_{n,k}, u_{1,k}, \dots, u_{p,k}) \\ f_2(x_{1,k}, \dots, x_{n,k}, u_{1,k}, \dots, u_{p,k}) \\ \vdots \\ f_n(x_{1,k}, \dots, x_{n,k}, u_{1,k}, \dots, u_{p,k}) \end{bmatrix} \quad (4.5)$$

4.2 Converting Nonlinear ODEs to $\dot{x} = f(x, u)$

With the Euler method, sometimes the most difficult part of the problem is to get the equation into the multivariable $\dot{x} = f(x, u)$ format. There are a few tips and tricks for doing so. The first tip that often works for higher order differential equations is to solve the equation for the highest order derivative. Then choose the state variables to be all of the lower order derivatives. This tip is illustrated in the following example.

Conversion to Nonlinear State-Space Form

Example 4.2.1. Converting a differential equation to nonlinear state-space form

Arrange the differential equation $\ddot{x} + \dot{x}^5 + \sin(x) = u_1 u_2 \cos(x)$ into the multivariable form $\dot{x} = f(x, u)$. Then use the Euler method to find an equation to solve for x at the next time-step x_{k+1} .

Solution: The first step is to solve the equation for the highest order derivative, which is \ddot{x} in this case:

$$\ddot{x} = -\dot{x}^5 - \sin(x) + u_1 u_2 \cos(x)$$

Next, we make all of the lower order derivatives, \dot{x} and x , the state variables. If the state variables are x and \dot{x} , we can write the equation in the desired form $\dot{x} = f(x, u)$ as follows:

$$\begin{bmatrix} \dot{x} \\ \ddot{x} \end{bmatrix} = \begin{bmatrix} \dot{x} \\ -\dot{x}^5 - \sin(x) + u_1 u_2 \cos(x) \end{bmatrix}$$

4.2 Converting Nonlinear ODEs to $\dot{x} = f(x, u)$

The first equation $\dot{x} = \dot{x}$ may seem trivial, but it is very useful. Now we can apply the Euler method Eq. (4.5) to get an equation to solve for x at the next time-step x_{k+1}

$$\begin{bmatrix} x_{k+1} \\ \dot{x}_{k+1} \end{bmatrix} = \begin{bmatrix} x_k \\ \dot{x}_k \end{bmatrix} + \Delta t \begin{bmatrix} \dot{x}_k \\ -\dot{x}_k^5 - \sin(x_k) + u_{1,k} u_{2,k} \cos(x_k) \end{bmatrix}$$

which was the goal of this example problem.

Sometimes solving the differential equations for the highest-order derivative can be challenging. The following example illustrates a second tip that solves a set of equations simultaneously using linear algebra to get an equation in terms of the highest order derivatives.

Simultaneous Solutions to Convert to Nonlinear State-Space

Example 4.2.2. Using simultaneous solutions to convert to nonlinear state-space

The equations of motion for a certain inverted pendulum robot^a are

$$\begin{aligned} (J_1 + J_2 \cos(\theta))\ddot{\theta} + J_2 \cos(\theta)\ddot{\alpha} &= a_3 \dot{\alpha} + a_4 \sin(\theta) - bu \\ (J_3 + J_2 \cos(\theta))\ddot{\theta} + J_3 \ddot{\alpha} &= J_2 \sin(\theta)\dot{\theta}^2 - a_3 \dot{\alpha} + bu \end{aligned}$$

Arrange these equations into the multivariable format $\dot{x} = f(x, u)$. The values of J_1, J_2, J_3, a_3, a_4 , and b are known constants.

Solution: Each of the pendulum equations has two highest-order derivatives $\ddot{\theta}$ and $\ddot{\alpha}$. The first trick is to solve these equations simultaneously using linear algebra to get them in terms of their highest order derivatives. The equations can be written in matrix form.

$$\begin{bmatrix} (J_1 + J_2 \cos(\theta)) & J_2 \cos(\theta) \\ (J_3 + J_2 \cos(\theta)) & J_3 \end{bmatrix} \begin{bmatrix} \ddot{\theta} \\ \ddot{\alpha} \end{bmatrix} = \begin{bmatrix} a_3 \dot{\alpha} + a_4 \sin(\theta) - bu \\ J_2 \sin(\theta)\dot{\theta}^2 - a_3 \dot{\alpha} + bu \end{bmatrix}$$

Solving for the highest order derivatives

$$\begin{bmatrix} \ddot{\theta} \\ \ddot{\alpha} \end{bmatrix} = \begin{bmatrix} (J_1 + J_2 \cos(\theta)) & J_2 \cos(\theta) \\ (J_3 + J_2 \cos(\theta)) & J_3 \end{bmatrix}^{-1} \begin{bmatrix} a_3 \dot{\alpha} + a_4 \sin(\theta) - bu \\ J_2 \sin(\theta)\dot{\theta}^2 - a_3 \dot{\alpha} + bu \end{bmatrix}$$

Now we can apply an earlier trick and make the lower-order derivatives $-\theta, \alpha, \dot{\theta}$, and $\dot{\alpha}$ – the state variables. If we do so, the robot equations can be written in the multivariable $\dot{x} = f(x, u)$ format:

$$\begin{bmatrix} \dot{\theta} \\ \dot{\alpha} \\ \ddot{\theta} \\ \ddot{\alpha} \end{bmatrix} = \begin{bmatrix} \theta \\ \alpha \\ \begin{bmatrix} (J_1 + J_2 \cos(\theta)) & J_2 \cos(\theta) \\ (J_3 + J_2 \cos(\theta)) & J_3 \end{bmatrix}^{-1} \begin{bmatrix} a_3 \dot{\alpha} + a_4 \sin(\theta) - bu \\ J_2 \sin(\theta)\dot{\theta}^2 - a_3 \dot{\alpha} + bu \end{bmatrix} \end{bmatrix}$$

The Euler method solution is

$$\begin{bmatrix} \theta_{k+1} \\ \alpha_{k+1} \\ \dot{\theta}_{k+1} \\ \dot{\alpha}_{k+1} \end{bmatrix} = \begin{bmatrix} \theta_k \\ \alpha_k \\ \dot{\theta}_k \\ \dot{\alpha}_k \end{bmatrix} + \Delta t \begin{bmatrix} \dot{\theta}_k \\ \dot{\alpha}_k \\ \begin{bmatrix} (J_1 + J_2 \cos(\theta_k)) & J_2 \cos(\theta_k) \\ (J_3 + J_2 \cos(\theta_k)) & J_3 \end{bmatrix}^{-1} \begin{bmatrix} a_3 \dot{\alpha}_k + a_4 \sin(\theta_k) - bu_k \\ J_2 \sin(\theta_k) \dot{\theta}_k^2 - a_3 \dot{\alpha}_k + bu_k \end{bmatrix} \end{bmatrix}$$

which was the goal of this example problem.

^aSee B. Pence and A. Dean, “Balancing the MinSeg Robot”, BYU-Idaho, 2020, https://byui.instructure.com/files/37636384/download?download_frd=1

4.2.1 Simulation Examples

This section provides simulation examples that use the Euler method to solve differential equations.

Scalar Example

The first simulation solves the differential equation

$$\dot{x} = -ax + bu \quad (4.6)$$

where

$$u = \sin(wt) \quad (4.7)$$

and the initial condition is $x(0) = x_0$. This differential equation has the known analytical solution

$$x = \left(x_0 + \sin\left(\tan^{-1}\left(\frac{w}{a}\right)\right) \frac{b}{\sqrt{w^2 + a^2}} \right) \exp(-at) + \frac{b}{\sqrt{w^2 + a^2}} \sin\left(wt - \tan^{-1}\left(\frac{w}{a}\right)\right) \quad (4.8)$$

We will use the analytical solution to verify the accuracy of the Euler method approximate solution.

```
%set the parameter values
b = 3;
a = 5;
w = 6;
x0 = -2;

%Set the simulation parameters
dt = 0.1;
t = 0:dt:4;
N = length(t);
u = sin(w*t);
x = x0;
xEuler = zeros(1,N);
for ii = 1:N
    xEuler(ii) = x; %Store the solution
    %Use the Euler method to find the next value of x
    x = x + dt*(-a*x+b*sin(w*t(ii)));
end

% Calculate the analytical solution
```

4.2 Converting Nonlinear ODEs to $\dot{x} = f(x, u)$

```

xTrue = (x0+sin(atan(w/a))*b/sqrt(w^2+a^2))*exp(-a*t)+...
b/sqrt(w^2+a^2)*sin(w*t-atan(w/a));

%repeat the simulation, but this time with a faster time-step
%Set the simulation parameters
dt = 0.001;
t1 = 0:dt:4;
N = length(t1);
u = sin(w*t);
x = x0;
xEulerFast = zeros(1,N);
for ii = 1:N
    xEulerFast(ii) = x; %Store the solution
    %Use the Euler method to find the next value of x
    x = x + dt*(-a*x+b*sin(w*t1(ii)));
end

%Plot the results comparing the analytical and Euler solutions
figure, plot(t, xTrue, 'o', t, xEuler, '--', t1, xEulerFast)
xlabel('Time (s)')
ylabel('x')
legend('Analytical', 'Euler \Delta t=0.1', 'Euler \Delta t=0.001', ...
'Location', 'SouthEast')
title('Analytical vs. Euler')

```

The results of the MATLAB simulation are shown in Figure 4.1. It shows that the Euler approximation improves with smaller time steps Δt .

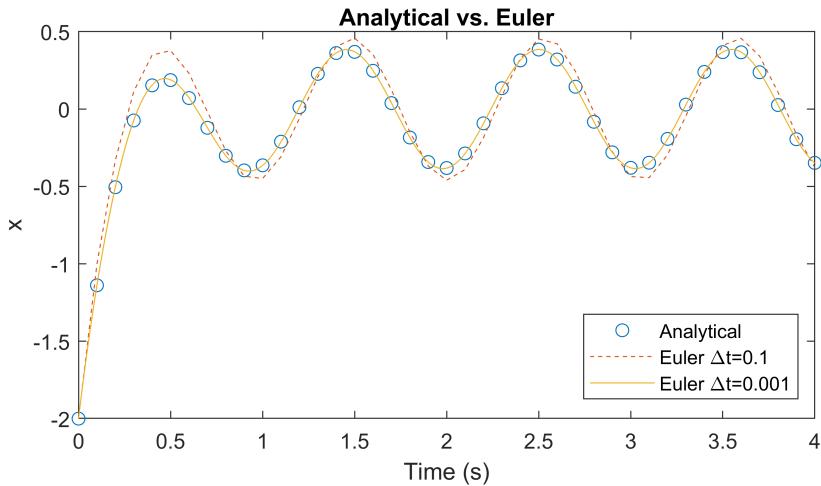


Figure 4.1 Simulation results of $\dot{x} = -ax + bu$ comparing the Euler method and the analytical solution

Multivariable Example

This section uses the Euler method to solve a multivariable differential equation. It will simulate the inverted pendulum robot in Example 4.2.2 above. Initially, the robot is in the vertical upright position, but as time continues, it tips over and oscillates around the upside-down position as shown in Figure 4.2 and Figure 4.3.

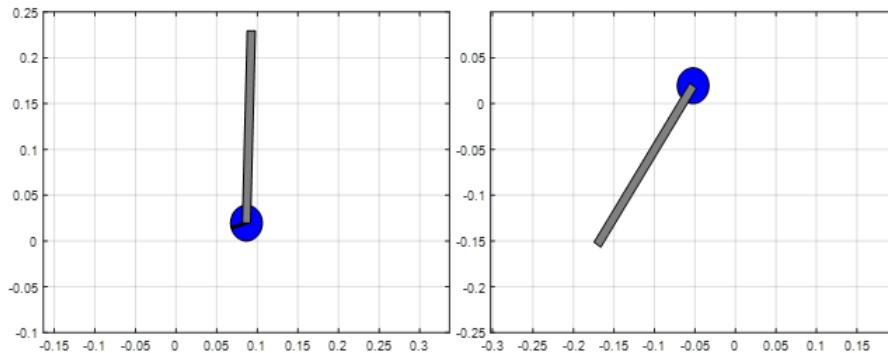


Figure 4.2 Pendulum robot in its vertical upright and tipping over positions

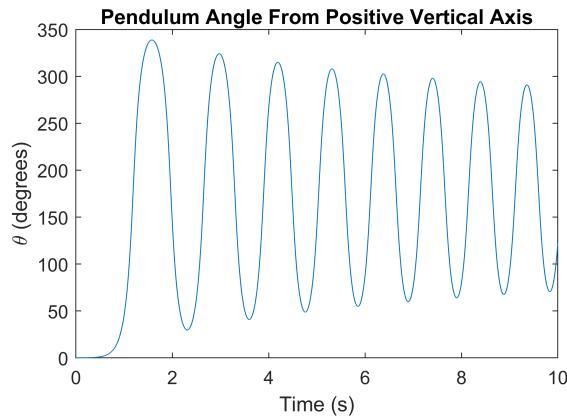


Figure 4.3 Simulation results of the pendulum robot

```

close all
clear all %clear the persistent variable
clc

rc = 0.105; %(m) pendulum distance to center of gravity
Lp = 2*rc; %(m) pendulum length
rw = 0.0195; %(m) wheel radius
w = 0.01; %(m) width of pendulum
th = 0:pi/50:2*pi; %get theta from 0 to 2*pi

dt = 0.001; %(s) simulation time step

```

4.2 Converting Nonlinear ODEs to $\dot{x} = f(x, u)$

```

t = 0:dt:10; %(s) time array
N = length(t); %length of the time array
theta = zeros(1,N);
dtheta = zeros(1,N);
dalpha = zeros(1,N);
alpha = zeros(1,N);

figure(1)
for ii = 1:N
    Vin = 0;
    [theta(ii),dtheta(ii),dalpha(ii),alpha(ii)] = MinSeg(Vin,dt);

    %Animate the pendulum
    beta = theta(ii)+alpha(ii);
    xAxe = rw*beta; %x-position of the axle
    yAxe = rw; %y-position of the axle
    pendulumX = [xAxe-w/2*cos(theta(ii)), xAxe+w/2*cos(theta(ii)),...
        xAxe+Lp*sin(theta(ii))+w/2*cos(theta(ii)),...
        xAxe+Lp*sin(theta(ii))-w/2*cos(theta(ii))];
    pendulumY = [yAxe+w/2*sin(theta(ii)), yAxe-w/2*sin(theta(ii)),...
        yAxe+Lp*cos(theta(ii))-w/2*sin(theta(ii)),...
        yAxe+Lp*cos(theta(ii))+w/2*sin(theta(ii))];
    figure(1)

    if ~mod(ii,10)
        xunit = rw * cos(theta(ii)) + xAxe; %get the x-indices of the ball
        yunit = rw * sin(theta(ii)) + yAxe; %get the y-indices of the ball
        fill(xunit, yunit, 'b'); %draw the ball and make it red
        hold on %use the currently open figure
        plot([xAxe, xAxe+rw*sin(beta)], [yAxe, yAxe+rw*cos(beta)],...
            'k','LineWidth',3)
        fill(pendulumX, pendulumY, [0.5,0.5,0.5]);
        xlim([xAxe-0.25, xAxe+0.25])
        ylim([-0.1, 0.25])
        %ylim([-0.25, 0.1])
        grid on
        drawnow
        hold off %stop using the currently open figure
    end
end
figure, plot(t, theta*180/pi)
title('Pendulum Angle From Positive Vertical Axis')
ylabel('\theta (degrees)')
xlabel('Time (s)')

function [theta, thetaDot, alphaDot, alpha] = MinSeg(Vin, dt)

persistent xMinSeg

if isempty(xMinSeg)
    xMinSeg = [0.001;0;0;0];
end

Vin = max(-5,min(5, Vin)); %Limit the voltage from the batteries

```

```

kT = 0.32; %(Nm/A)
b = 0.0048; %(Nms/rad)
J = 9e-4; %(kg m2)
Jw = 1e-4; %(kgm2) wheel moment of inertia
rc = 0.105; %(m) pendulum distance to center of gravity
R = 4.6; %(Ohm)
m = 0.283; %(kg) pendulum mass without wheels
mw = 0.0344; %(kg) combined wheel mass
rw = 0.0195; %(m) wheel radius
g = 9.81; %(m/s2) gravitational acceleration

%parameters for the pendulum robot
J1 = J+m*rc^2;
J2 = m*rc*rw;
J3 = Jw+(m+mw)*rw^2;
a3 = (kT^2/R+b);
a4 = m*g*rc;
b = kT/R;

M = [(J1+J2*cos(xMinSeg(1))), J2*cos(xMinSeg(1)); ...
       (J3+J2*cos(xMinSeg(1))), J3];

Q = [a3*xMinSeg(4)+a4*sin(xMinSeg(1))-b*Vin; ...
      J2*sin(xMinSeg(1))*xMinSeg(3)^2-a3*xMinSeg(4)+b*Vin];

%get the MinSeg state derivatives
dxMinSeg = [xMinSeg(3);...
            xMinSeg(4); ...
            M\Q];

%Numerically integrate (Euler method) to get the states
xMinSeg = xMinSeg+dt*dxMinSeg;

%Extract the state variables from the state vector
theta = xMinSeg(1);
alpha = xMinSeg(2);
thetaDot = xMinSeg(3);
alphaDot = xMinSeg(4);
end

```

4.3 The Runge-Kutta Method for Multivariable $\dot{x} = f(x, u)$

Another common technique used to solve nonlinear differential equations is the 4th-order Runge-Kutta method. It generally requires more computational resources than the Euler method, but it produces more accurate results (assuming both methods use the same step-size Δt).

The Runge-Kutta method works by finding a weighted average of four variables k_1, \dots, k_4 that are defined below. Given the multivariable state-equations

4.3 The Runge-Kutta Method for Multivariable $\dot{x} = f(x, u)$

$$\begin{bmatrix} \dot{x}_1 \\ \dot{x}_2 \\ \vdots \\ \dot{x}_n \end{bmatrix} = \begin{bmatrix} f_1(x_1, \dots, x_n, u_1, \dots, u_p) \\ f_2(x_1, \dots, x_n, u_1, \dots, u_p) \\ \vdots \\ f_n(x_1, \dots, x_n, u_1, \dots, u_p) \end{bmatrix} \quad (4.9)$$

k_1, \dots, k_4 are defined as follows¹:

$$k_1 = \begin{bmatrix} k_{1,1} \\ k_{1,2} \\ \vdots \\ k_{1,n} \end{bmatrix} = \begin{bmatrix} f_1(x_{1,k}, \dots, x_{n,k}, u_{1,k}, \dots, u_{p,k}) \\ f_2(x_{1,k}, \dots, x_{n,k}, u_{1,k}, \dots, u_{p,k}) \\ \vdots \\ f_n(x_{1,k}, \dots, x_{n,k}, u_{1,k}, \dots, u_{p,k}) \end{bmatrix} \quad (4.10)$$

$$k_2 = \begin{bmatrix} k_{2,1} \\ k_{2,2} \\ \vdots \\ k_{2,n} \end{bmatrix} = \begin{bmatrix} f_1(x_{1,k} + \frac{\Delta t}{2} k_{1,1}, \dots, x_{n,k} + \frac{\Delta t}{2} k_{1,n}, u_{1,k}, \dots, u_{p,k}) \\ f_2(x_{1,k} + \frac{\Delta t}{2} k_{1,1}, \dots, x_{n,k} + \frac{\Delta t}{2} k_{1,n}, u_{1,k}, \dots, u_{p,k}) \\ \vdots \\ f_n(x_{1,k} + \frac{\Delta t}{2} k_{1,1}, \dots, x_{n,k} + \frac{\Delta t}{2} k_{1,n}, u_{1,k}, \dots, u_{p,k}) \end{bmatrix} \quad (4.11)$$

$$k_3 = \begin{bmatrix} k_{3,1} \\ k_{3,2} \\ \vdots \\ k_{3,n} \end{bmatrix} = \begin{bmatrix} f_1(x_{1,k} + \frac{\Delta t}{2} k_{2,1}, \dots, x_{n,k} + \frac{\Delta t}{2} k_{2,n}, u_{1,k}, \dots, u_{p,k}) \\ f_2(x_{1,k} + \frac{\Delta t}{2} k_{2,1}, \dots, x_{n,k} + \frac{\Delta t}{2} k_{2,n}, u_{1,k}, \dots, u_{p,k}) \\ \vdots \\ f_n(x_{1,k} + \frac{\Delta t}{2} k_{2,1}, \dots, x_{n,k} + \frac{\Delta t}{2} k_{2,n}, u_{1,k}, \dots, u_{p,k}) \end{bmatrix} \quad (4.12)$$

$$k_4 = \begin{bmatrix} k_{4,1} \\ k_{4,2} \\ \vdots \\ k_{4,n} \end{bmatrix} = \begin{bmatrix} f_1(x_{1,k} + \Delta t k_{3,1}, \dots, x_{n,k} + \Delta t k_{3,n}, u_{1,k}, \dots, u_{p,k}) \\ f_2(x_{1,k} + \Delta t k_{3,1}, \dots, x_{n,k} + \Delta t k_{3,n}, u_{1,k}, \dots, u_{p,k}) \\ \vdots \\ f_n(x_{1,k} + \Delta t k_{3,1}, \dots, x_{n,k} + \Delta t k_{3,n}, u_{1,k}, \dots, u_{p,k}) \end{bmatrix} \quad (4.13)$$

Then the Runge-Kutta method solves one step in time from t_k to t_{k+1} to get x_{k+1} as follows

$$\begin{bmatrix} x_{1,k+1} \\ x_{2,k+1} \\ \vdots \\ x_{n,k+1} \end{bmatrix} \approx \begin{bmatrix} x_{1,k} \\ x_{2,k} \\ \vdots \\ x_{n,k} \end{bmatrix} + \frac{\Delta t}{6} \left(\begin{bmatrix} k_{1,1} \\ k_{1,2} \\ \vdots \\ k_{1,n} \end{bmatrix} + 2 \begin{bmatrix} k_{2,1} \\ k_{2,2} \\ \vdots \\ k_{2,n} \end{bmatrix} + 2 \begin{bmatrix} k_{3,1} \\ k_{3,2} \\ \vdots \\ k_{3,n} \end{bmatrix} + \begin{bmatrix} k_{4,1} \\ k_{4,2} \\ \vdots \\ k_{4,n} \end{bmatrix} \right) \quad (4.14)$$

4.3.1 Simulation Examples

The simulations in this section are the same as those in Section 4.2.1 except this section uses the Runge-Kutta 4th-order method instead of the Euler method.

¹If at time t_k the values of the inputs u_1, \dots, u_p are known at the future times $t_k + \frac{\Delta t}{2}$ and t_{k+1} , then the predictions of k_1 through k_4 can be improved. Some texts include these improvements; however, in mechatronic applications, it is rare that future values of the inputs are known in advance.

Scalar Example

The first simulation solves the differential equation

$$\dot{x} = -ax + bu \quad (4.15)$$

where

$$u = \sin(wt) \quad (4.16)$$

and the initial condition is $x(0) = x_0$. This differential equation has the known analytical solution

$$x = \left(x_0 + \sin\left(\tan^{-1}\left(\frac{w}{a}\right)\right) \right) \exp(-at) + \frac{b}{\sqrt{w^2 + a^2}} \sin\left(wt - \tan^{-1}\left(\frac{w}{a}\right)\right) \quad (4.17)$$

We will use the analytical solution to verify the accuracy of the Runge-Kutta method approximate solution.

```

close all
clear
clc

%set the parameter values
b = 3;
a = 5;
w = 6;
x0 = -2;
%define the function for dx = -a*x+b*u
f = @(x,u) -a*x+b*u;

%Set the simulation parameters
dt = 0.1;
t = 0:dt:4;
N = length(t);
u = sin(w*t);
x = x0;
xRungeKutta = zeros(1,N);
for ii = 1:N
    xRungeKutta(ii) = x; %Store the solution
    %Use the Runge Kutta method to find the next value of x
    k1 = f(x,u(ii));
    k2 = f(x+dt/2*k1,u(ii));
    k3 = f(x+dt/2*k2,u(ii));
    k4 = f(x+dt*k3,u(ii));
    x = x + dt/6*(k1+2*k2+2*k3+k4);
end

% Calculate the analytical solution
xTrue = (x0+sin(atan(w/a))*b/sqrt(w^2+a^2))*exp(-a*t)+...
b/sqrt(w^2+a^2)*sin(w*t-atan(w/a));

%repeat the simulation, but this time with the Euler Method
%Set the simulation parameters
dt = 0.1;
t1 = 0:dt:4;
N = length(t1);

```

4.3 The Runge-Kutta Method for Multivariable $\dot{x} = f(x, u)$

```

u = sin(w*t);
x = x0;
xEuler = zeros(1,N);
for ii = 1:N
    xEuler(ii) = x; %Store the solution
    %Use the Euler method to find the next value of x
    x = x + dt*(-a*x+b*u(ii));
end

%Plot the results comparing the analytical and Euler solutions
figure, plot(t, xTrue, 'o', t, xRungeKutta, '--', t1, xEuler)
xlabel('Time (s)')
ylabel('x')
legend('Analytical','Runge-Kutta \Delta t=0.1','Euler \Delta t=0.1',...
'Location','SouthEast')
title('Runge-Kutta vs. Analytical and Euler')

```

The results of the MATLAB simulation are shown in Figure 4.4. It shows that the Runge-Kutta approximation compared with the Euler and analytical solutions.

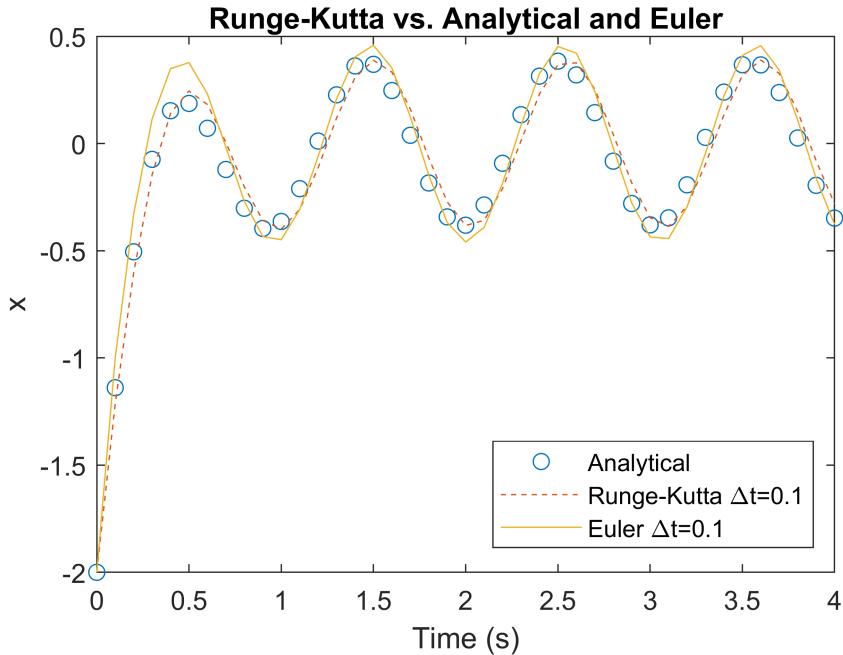


Figure 4.4 Simulation results of $\dot{x} = -ax + bu$ comparing the Runge-Kutta method with the Euler method and the analytical solution

Multivariable Example

This section uses the Runge-Kutta method to solve a multivariable differential equation. It will simulate the inverted pendulum robot in Example 4.2.2 above. It compares the Runge-Kutta results with the Euler results from the previous chapter. Initially the robot is in the vertical upright position, but as time continues, it tips over and oscillates around the upside-down position.

The results comparing the Runge-Kutta and Euler solutions are shown in Figures 4.5 and 4.6. Notice that as the timestep for the Euler method gets shorter, the Euler method solution approaches the solution from the Runge-Kutta method. This suggests that, for the same simulation time-step, the Runge-Kutta method produced a more accurate result.

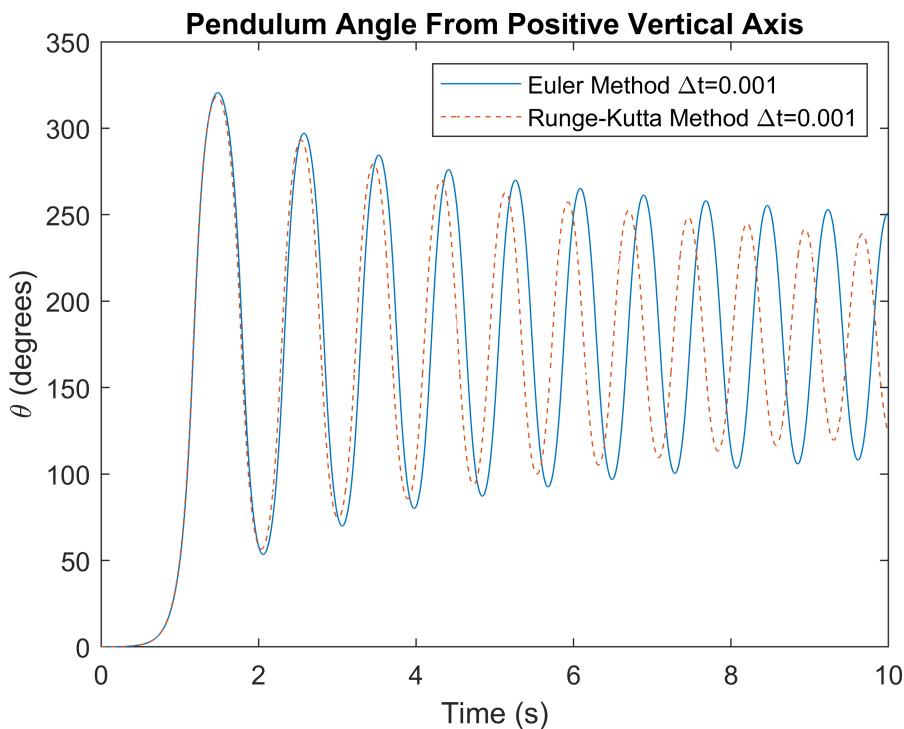


Figure 4.5 Simulation results for the pendulum robot. Both the Runge-Kutta and Euler methods used the same time-step $\Delta t = 0.001$ s.

```

close all
clear all %clear the persistent variable
clc

rc = 0.105; %(m) pendulum distance to center of gravity
Lp = 2*rc; %(m) pendulum length
rw = 0.0195; %(m) wheel radius
w = 0.01; %(m) width of pendulum
th = 0:pi/50:2*pi; %get theta from 0 to 2*pi

dt = 0.001; %(s) simulation time step

```

4.3 The Runge-Kutta Method for Multivariable $\dot{x} = f(x, u)$

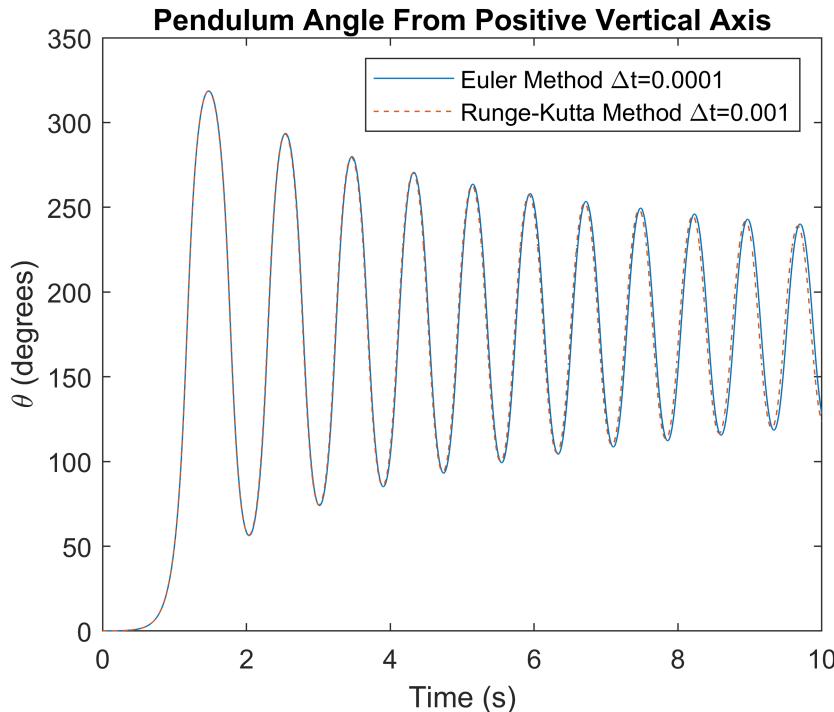


Figure 4.6 Simulation results for the pendulum robot. The Runge-Kutta method used a time-step of $\Delta t = 0.001$ s, but the Euler method used a time-step that was 10-times faster $\Delta t = 0.0001$ s.

```
t = 0:dt:10; %(s) time array
N = length(t); %length of the time array
theta = zeros(1,N);
dtheta = zeros(1,N);
dalpha = zeros(1,N);
alpha = zeros(1,N);
for ii = 1:N
    Vin = 0;
    [theta(ii),dtheta(ii),dalpha(ii),alpha(ii)] = MinSeg(Vin,dt);
end
figure, plot(t, theta*180/pi)
title('Pendulum Angle From Positive Vertical Axis')
ylabel('\theta (degrees)')
xlabel('Time (s)')

function k = fMinSeg(xMinSeg,Vin)
kT = 0.32; %(Nm/A)
b = 0.0048; %(Nms/rad)
J = 9e-4; %(kg m2)
Jw = 1e-4; %(kgm2) wheel moment of inertia
rc = 0.105; %(m) pendulum distance to center of gravity
R = 4.6; %(Ohm)
m = 0.283; %(kg) pendulum mass without wheels
mw = 0.0344; %(kg) combined wheel mass
```

```

rw = 0.0195; %(m) wheel radius
g = 9.81; %(m/s^2) gravitational acceleration

%parameters for the pendulum robot
J1 = J+m*rc^2;
J2 = m*rc*rw;
J3 = Jw+(m+mw)*rw^2;
a3 = (kT^2/R+b);
a4 = m*g*rc;
b = kT/R;

M = [(J1+J2*cos(xMinSeg(1))), J2*cos(xMinSeg(1)); ...
       (J3+J2*cos(xMinSeg(1))), J3];

Q = [a3*xMinSeg(4)+a4*sin(xMinSeg(1))-b*Vin; ...
      J2*sin(xMinSeg(1))*xMinSeg(3)^2-a3*xMinSeg(4)+b*Vin];

%get the MinSeg state derivatives
k = [xMinSeg(3);...
      xMinSeg(4); ...
      M \ Q];
end

function [theta, thetaDot, alphaDot, alpha] = MinSeg(Vin, dt)

persistent xMinSeg

if isempty(xMinSeg)
    xMinSeg = [0.001;0;0;0];
end

Vin = max(-5,min(5, Vin)); %Limit the voltage from the batteries

%Numerically integrate (Runge-Kutta method) to get the states
k1 = fMinSeg(xMinSeg,Vin);
k2 = fMinSeg(xMinSeg+dt/2*k1,Vin);
k3 = fMinSeg(xMinSeg+dt/2*k2,Vin);
k4 = fMinSeg(xMinSeg+dt*k3,Vin);
xMinSeg = xMinSeg+dt/6*(k1+2*k2+2*k3+k4);

%Extract the state variables from the state vector
theta = xMinSeg(1);
alpha = xMinSeg(2);
thetaDot = xMinSeg(3);
alphaDot = xMinSeg(4);
end

```

4.4 Linearizing Nonlinear Systems

4.4 Linearizing Nonlinear Systems

This section uses Taylor series expansions to linearize nonlinear equations. Linearization of nonlinear systems is helpful in many applications, especially for using feedback control or Kalman filtering. As with the Euler and Runge-Kutta methods, the nonlinear system must be in the form $\dot{x} = f(x, u)$. Section 4.2 introduced some tips for converting nonlinear ODEs to the $\dot{x} = f(x, u)$ form.

4.4.1 Taylor Series Expansions

Taylor series expansions can be used to approximate functions. The Taylor series expansion of the scalar function $f(x)$ around the known scalar value x^* is

$$f(x) \approx f(x^*) + \frac{1}{1!} \frac{df(x^*)}{dx} (x - x^*) + \frac{1}{2!} \frac{d^2f(x^*)}{dx^2} (x - x^*)^2 + \dots \quad (4.18)$$

$$f(x) \approx f(x^*) + \sum_{k=1}^{\infty} \frac{1}{k!} \frac{d^k f(x^*)}{dx^k} (x - x^*)^k \quad (4.19)$$

These equations use the notation $\frac{d^k f(x^*)}{dx^k}$, which is the k^{th} derivative of $f(x)$ with respect to x evaluated at $x = x^*$.

4.4.2 Multivariable Taylor Series Expansion

We rarely deal with scalar functions of a single scalar independent variable. More often, the state-equations are vectors of functions with multiple variables. The Taylor series expansion of a scalar function $f(x_1, x_2, \dots, x_n)$ with multiple independent variables x_1, x_2, \dots, x_n about the known values $x_1^*, x_2^*, \dots, x_n^*$ is

$$\begin{aligned} f(x_1, \dots, x_n) &\approx f(x_1^*, \dots, x_n^*) + \frac{1}{1!} \sum_{i=1}^n \frac{df(x_1^*, \dots, x_n^*)}{dx_i} (x_i - x_i^*) \\ &+ \frac{1}{2!} \sum_{i=1}^n \sum_{j=1}^n \frac{d^2f(x_1^*, \dots, x_n^*)}{dx_i dx_j} (x_i - x_i^*)(x_j - x_j^*) + \dots \end{aligned} \quad (4.20)$$

For vectors of functions, we apply the Taylor series expansion to each function in the vector. This will be demonstrated in Example 4.4.1.

4.4.3 Linearization of $\dot{x} = f(x, u)$ around x^* and u^*

Nonlinear state equations are generally multivariable because the state x is a vector and the input u may also be a vector. To linearize the equation $\dot{x} = f(x, u)$ around the vectors x^* and u^* , we apply the first two terms in the multivariable Taylor series expansion of Eq. (4.20). If x has p elements x_1, \dots, x_p , and u has q elements u_1, \dots, u_q , then f also has p elements $f_1(x, u), \dots, f_p(x, u)$. The linearization of $\dot{x} = f(x, u)$ around x^* and u^* is therefore

$$\begin{bmatrix} \dot{x}_1 \\ \vdots \\ \dot{x}_p \end{bmatrix} \approx \begin{bmatrix} f_1(x^*, u^*) + \sum_{i=1}^p \frac{df_1(x^*, u^*)}{dx_i} (x_i - x_i^*) + \sum_{i=1}^q \frac{df_1(x^*, u^*)}{du_i} (u_i - u_i^*) \\ \vdots \\ f_p(x^*, u^*) + \sum_{i=1}^p \frac{df_p(x^*, u^*)}{dx_i} (x_i - x_i^*) + \sum_{i=1}^q \frac{df_p(x^*, u^*)}{du_i} (u_i - u_i^*) \end{bmatrix} \quad (4.21)$$

The terms in Eq. (4.21) that are multiplied by the state variables x_1, \dots, x_p become the A matrix in $\dot{x} = Ax + Bu$. Other terms are grouped into the B matrix and input vector u .

The resulting A matrix is the **Jacobian matrix**

$$A = \begin{bmatrix} \frac{df_1}{dx_1} & \cdots & \frac{df_1}{dx_p} \\ \vdots & \ddots & \vdots \\ \frac{df_p}{dx_1} & \cdots & \frac{df_p}{dx_p} \end{bmatrix} \quad (4.22)$$

evaluated at x^* and u^* . The first q columns of the B matrix are

$$B_{1:q} = \begin{bmatrix} \frac{df_1}{du_1} & \cdots & \frac{df_1}{du_q} \\ \vdots & \ddots & \vdots \\ \frac{df_p}{du_1} & \cdots & \frac{df_p}{du_q} \end{bmatrix} \quad (4.23)$$

evaluated at x^* and u^* . The last column of the B matrix consists of the remaining terms from the linearization, namely

$$B_{q+1} = \begin{bmatrix} f_1(x^*, u^*) - \sum_{i=1}^p \frac{df_1(x^*, u^*)}{dx_i} x_i^* - \sum_{i=1}^q \frac{df_1(x^*, u^*)}{du_i} u_i^* \\ \vdots \\ f_p(x^*, u^*) - \sum_{i=1}^p \frac{df_p(x^*, u^*)}{dx_i} x_i^* - \sum_{i=1}^q \frac{df_p(x^*, u^*)}{du_i} u_i^* \end{bmatrix} \quad (4.24)$$

The input is expanded to include one more element to be

$$u = \begin{bmatrix} u \\ 1 \end{bmatrix} \quad (4.25)$$

This is demonstrated in the following example.

Linearizing a Nonlinear State Equation

Example 4.4.1. Linearizing a nonlinear state equation

Linearize the nonlinear state equation

$$\begin{bmatrix} \dot{x}_1 \\ \dot{x}_2 \end{bmatrix} = \begin{bmatrix} -x_1 x_2 + \ln(u) \\ x_1 u - \frac{u x_1}{x_2} \end{bmatrix}$$

around the values $x^* = [0, 1]^T$ and $u^* = 4$ to get a linear state equation in the form $\dot{x} = Ax + Bu$.

Solution: The nonlinear function $f(x, u)$ is a vector of two functions $f_1(x, u) = -x_1 x_2 + \ln(u)$ and

4.4 Linearizing Nonlinear Systems

$f_2(x, u) = x_1 u - \frac{u^{x_1}}{x_2}$. We first find the first two terms of the Taylor series expansion Eq. (4.20) for $f_1(x, u)$ to get

$$f_1(x, u) \approx -x_1^* x_2^* + \ln(u^*) - x_2^*(x_1 - x_1^*) - x_1^*(x_2 - x_2^*) + \frac{1}{u^*}(u - u^*)$$

which can be simplified to

$$f_1(x, u) \approx \begin{bmatrix} -x_2^* & -x_1^* \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} + \frac{1}{u^*} u + (x_1^* x_2^* + \ln(u^*) - 1) \quad (4.26)$$

Next, we use the first two terms of the Taylor series expansion of $f_2(x, u)$ to get

$$\begin{aligned} f_2(x, u) \approx & x_1^* u^* - \frac{u^{*x_1^*}}{x_2^*} + u^*(x_1 - x_1^*) - \frac{\ln(u^*)}{x_2^*} u^{*x_1^*} (x_1 - x_1^*) \\ & + \frac{u^{*x_1^*}}{x_2^{*2}} (x_2 - x_2^*) + x_1^*(u - u^*) - \frac{x_1^*}{x_2^*} u^{*x_1^*-1} (u - u^*) \end{aligned}$$

which can be simplified to

$$\begin{aligned} f_2(x, u) \approx & \left[u^* - \frac{\ln(u^*)}{x_2^*} u^{*x_1^*} \quad \frac{u^{*x_1^*}}{x_2^{*2}} \right] \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} + \left(x_1^* - \frac{x_1^*}{x_2^*} u^{*x_1^*-1} \right) u \\ & - 2 \frac{u^{*x_1^*}}{x_2^*} - x_1^* u^* + \frac{x_1^*}{x_2^*} u^{*x_1^*} + \frac{\ln(u^*) x_1^*}{x_2^*} u^{*x_1^*} \end{aligned} \quad (4.27)$$

Eqs. (4.26) and (4.27) can be combined to get the desired $\dot{x} = Ax + Bu$ form:

$$\begin{aligned} \begin{bmatrix} \dot{x}_1 \\ \dot{x}_2 \end{bmatrix} \approx & \begin{bmatrix} -x_2^* & -x_1^* \\ u^* - \frac{\ln(u^*)}{x_2^*} u^{*x_1^*} & \frac{u^{*x_1^*}}{x_2^{*2}} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \\ & + \begin{bmatrix} \frac{1}{u^*} & x_1^* x_2^* + \ln(u^*) - 1 \\ x_1^* - \frac{x_1^*}{x_2^*} u^{*x_1^*-1} & -2 \frac{u^{*x_1^*}}{x_2^*} - x_1^* u^* + \frac{x_1^*}{x_2^*} u^{*x_1^*} + \frac{\ln(u^*) x_1^*}{x_2^*} u^{*x_1^*} \end{bmatrix} \begin{bmatrix} u \\ 1 \end{bmatrix} \end{aligned} \quad (4.28)$$

If we plug in the values $x_1^* = 0$, $x_2^* = 1$, and $u^* = 4$, we can see that the A and B matrices are constant-valued, and the input vector is expanded to include an additional element $\begin{bmatrix} u & 1 \end{bmatrix}^T$:

$$\begin{bmatrix} \dot{x}_1 \\ \dot{x}_2 \end{bmatrix} \approx \begin{bmatrix} -1 & 0 \\ 2.6137 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} + \begin{bmatrix} 0.25 & 0.3863 \\ 0 & -2 \end{bmatrix} \begin{bmatrix} u \\ 1 \end{bmatrix} \quad (4.29)$$

The original equation is now in its linearized form, and the example is complete. We can use the techniques from Chapter 1 to solve Eq. (4.29).

4.4.4 Linearization around Constant x^* and u^*

A common approach in linearizing nonlinear systems is to use constant values for x^* and u^* ; usually x^* is the equilibrium value of x . In this approach, the values of x^* and u^* never change, regardless of new or updated information about the state x or input u . The state and input matrices A and B never change. In Example 4.4.1, the A matrix would always be

$$A = \begin{bmatrix} -1 & 0 \\ 2.6137 & 1 \end{bmatrix}$$

and the B matrix would always be

$$B = \begin{bmatrix} 0.25 & 0.3863 \\ 0 & -2 \end{bmatrix}$$

4.4.5 Linearization around the Values $x^* = x_k$ and $u^* = u_k$

Another option is to linearize the nonlinear equation about the most recent known values $x^* = x_k$ and $u^* = u_k$ at each time t_k . This approach is more accurate, but significantly less computationally efficient because the A and B matrices (and therefore the A_d and B_d matrices) are updated on each iteration.

The following example compares the two linearization approaches (constant A and B matrices versus updated A and B matrices).

Linearize then Simulate a Nonlinear System

Example 4.4.2. Linearize and simulate a nonlinear system

Plot the state trajectories of the nonlinear system

$$\begin{bmatrix} \dot{x}_1 \\ \dot{x}_2 \end{bmatrix} = \begin{bmatrix} -x_1 x_2 + \ln(u) \\ x_1 u - \frac{u x_1}{x_2} \end{bmatrix}$$

with a constant input $u = 4$ starting from the initial values $x^* = [0, 1]^T$. Compare the linearization approach of Section 4.4.4 against the approach of Section 4.4.5. Compare both approaches with the Runge-Kutta solution. Use a time-step of 0.02. Plot the state trajectories until the second state x_2 crosses zero, at which time the term $\frac{u x_1}{x_2}$ becomes infinite.

Solution: Eq. (4.28) of Example 4.4.1 linearized the system around x^* and u^* . Using the methods of Section 4.4.4, we plugged in the constant values $x^* = [0 \ 1]$ and $u^* = 4$ to get Eq. (4.29). To apply the methods of this section, we will use the most recent values $x^* = x_k$ and $u^* = u_k = 4$ instead. As a result, Eq. (4.28) becomes

4.4 Linearizing Nonlinear Systems

$$\begin{bmatrix} \dot{x}_1 \\ \dot{x}_2 \end{bmatrix} \approx \begin{bmatrix} -x_{2,k} & -x_{1,k} \\ u_k - \frac{\ln(u_k)}{x_{2,k}} u_k^{x_{1,k}} & \frac{u_k^{x_{1,k}}}{x_{2,k}^2} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} + \begin{bmatrix} \frac{1}{u_k} & x_{1,k}x_{2,k} + \ln(u_k) - 1 \\ x_{1,k} - \frac{x_{1,k}}{x_{2,k}} u_k^{x_{1,k}-1} & -2\frac{u_k^{x_{1,k}}}{x_{2,k}} - x_{1,k}u_k + \frac{x_{1,k}}{x_{2,k}} u_k^{x_{1,k}} + \frac{\ln(u_k)x_{1,k}}{x_{2,k}} u_k^{x_{1,k}} \end{bmatrix} \begin{bmatrix} u \\ 1 \end{bmatrix} \quad (4.30)$$

We now implement the numerical solutions in MATLAB to produce the following graphs. The Runge-Kutta method matches the state-trajectories of the linearized equations about the updated $x^* = x_k$ and $u^* = u_k$ much better than the linearization around the constant values $x^* = [0 \ 1]$ and $u^* = 4$. The Runge-Kutta and updated linearization methods diverge only when x_2 crosses zero causing a divide-by-zero singularity.

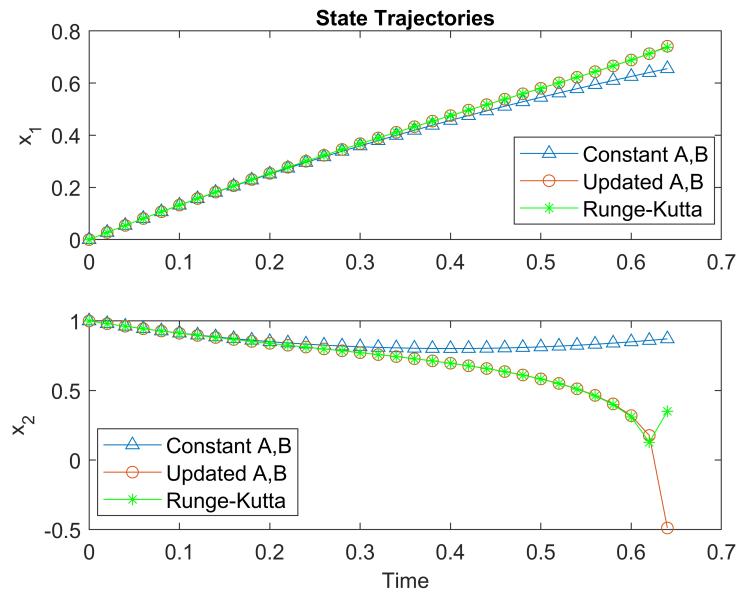


Figure 4.7 State trajectories

```

close all %close all open figures
clear all %clear all variables and functions
clc %clear the command window

% State-matrices for linearizing around constant x1=0, x2=1, u=4;
Ac = [-1, 0;
       2.6137, 1]; %State matrix
Bc = [0.25,0.3863;
      0, -2]; %Input matrix

%Choose a time-step (s) and create a time vector
dt = 0.02; %(s) time-step

```

```

t = 0:dt:0.64; %(s) time vector in steps of dt
N = length(t); %Number of array elements in the time vector

%calculate the Ad and Bd transition matrices for constant x* and u*
% Note: zeros(n,m) creates a matrix of zeros with n rows and m columns
Fdc = expm([Ac*dt, Bc*dt; zeros(2,2), zeros(2,2)]);
Adc = Fdc(1:2,1:2); %Ad is the upper left 2 by 2 submatrix of Fd
Bdc = Fdc(1:2,3:4); %Bd is the upper right 2 by 1 submatrix of Fd

%Set the initial conditions for constant x* and u*
xc = [0; 1]; %initial conditions for the state
xc1 = zeros(1,N); %initialize a vector to store x1
xc2 = zeros(1,N); %initialize a vector to store x2

%Set initial conditions for updated x* and u*
xk = [0; 1]; %initial conditions for the state
uk = 4; %initial condition for the input
xk1 = zeros(1,N); %initialize a vector to store x1
xk2 = zeros(1,N); %initialize a vector to store x2

%set initial conditions for the runge-kutta solution
xrk = [0;1];
xrk1 = zeros(1,N);
xrk2 = zeros(1,N);
for kk = 1:N
    %expand the input
    uk = 4;
    u = [uk; 1];

    %store the state trajectories for the updated x* and u*
    xk1(kk) = xk(1);
    xk2(kk) = xk(2);

    %Store the state trajectories for the constant
    xc1(kk) = xc(1); %x is the first output
    xc2(kk) = xc(2); %x_dot is the second output

    %store the runge-kutta state trajectories
    xrk1(kk) = xrk(1);
    xrk2(kk) = xrk(2);

    %Update the linearized matrices for updated x* and u*
    Ak = [-xk(2), -xk(1);
    uk-log(uk)*uk^xk(1)/xk(2), uk^xk(1)/xk(2)^2];
    Bk1 = 1/uk;
    Bk2 = xk(1)*xk(2)+log(uk)-1;
    Bk3 = xk(1)-xk(1)/xk(2)*uk^(xk(1)-1);
    Bk4 = -2*uk^xk(1)/xk(2)-xk(1)*uk+xk(1)*uk^xk(1)/xk(2)...
        +log(uk)*xk(1)/xk(2)*uk^xk(1);
    Bk = [Bk1, Bk2;
    Bk3, Bk4];

    %Solve one iteration of the state equation for updated x* and u*

```

4.4 Linearizing Nonlinear Systems

```
Fdk = expm([Ak*dt, Bk*dt; zeros(2,4)]);
Adk = Fdk(1:2, 1:2);
Bdk = Fdk(1:2, 3:4);
xk = Adk * xk + Bdk * u;

%Solve one iteration of the state equation for constant x* and u*
xc = Adc * xc + Bdc * u;

%solve the system using Runge-Kutta 4
k1 = [-xrk(1)*xrk(2)+log(uk);
        xrk(1)*uk-uk^xrk(1)/xrk(2)];
k2 = [-(xrk(1)+k1(1)*dt/2)*(xrk(2)+k1(2)*dt/2)+log(uk);
        (xrk(1)+k1(1)*dt/2)*uk-uk^(xrk(1)+k1(1)*dt/2)/(xrk(2)+k1(2)*dt/2)];
k3 = [-(xrk(1)+k2(1)*dt/2)*(xrk(2)+k2(2)*dt/2)+log(uk);
        (xrk(1)+k2(1)*dt/2)*uk-uk^(xrk(1)+k2(1)*dt/2)/(xrk(2)+k2(2)*dt/2)];
k4 = [-(xrk(1)+k3(1)*dt)*(xrk(2)+k3(2)*dt)+log(uk);
        (xrk(1)+k3(1)*dt)*uk-uk^(xrk(1)+k3(1)*dt)/(xrk(2)+k3(2)*dt)];
xrk = xrk + dt/6*(k1+2*k2+2*k3+k4);
end
%plot the state-trajectories
figure %create a new figure to draw a graph on
subplot(211) %create 2 plots on one figure, draw on the first plot
plot(t, xc1, '-^', t, xk1, '-o', t, xrk1,'g-*') %plot x1
legend('Constant A,B', 'Updated A,B', 'Runge-Kutta') %Create a legend
ylabel('x_1') %label the y-axis
title('State Trajectories') %Create the title
subplot(212) %draw on the second plot
plot(t, xc2, '-^', t, xk2, '-o', t, xrk2,'g-*') %plot x2
ylabel('x_2') %Label the y-axis
%Label the x-axis
xlabel('Time')
%Create a graph legend
legend('Constant A,B', 'Updated A,B', 'Runge-Kutta')
```


Chapter 5

Block Diagrams of Control Systems

Contents

5.1	Basic Block Diagrams	135
5.2	Feedforward Path, Feedback Path, and the Feedback Loop	137
5.2.1	Block Diagrams with Multiple Inputs and Outputs	139
5.2.2	Block Diagrams with Multiple Feedback Paths	142
5.3	Control Systems	144
5.3.1	Feedforward Controllers	145
5.3.2	Feedback Controllers	145
5.3.3	The Famous PID Controller	146
5.3.4	Integrator Anti-Windup	149
5.3.5	Modeling Nonlinear ODEs using Integrators	153
5.4	Block Diagrams of Discrete-Time Systems	154

Control system block diagrams can provide a helpful visual representation of dynamic systems, their controllers, inputs, and outputs. In addition, software packages such as Simulink® can solve dynamic systems in block diagram form. This chapter introduces block diagrams along with a variety of feedforward and feedback controllers. It also presents techniques for evaluating the stability and response time of Proportional-Integral-Derivative (PID) controllers.

Block diagrams consist of blocks connected by arrows. Blocks can be Laplace or Z-domain transfer functions, mathematical operations, state-space equations, subsystems, inputs, outputs, signal generators, etc. Arrows are input and output signals from the blocks which may be scalars or arrays. The next section discusses continuous-time block diagrams consisting of Laplace-domain transfer functions.

5.1 Basic Block Diagrams

Continuous-time control system block diagrams show mathematical relationships between inputs and outputs in the Laplace domain. A Laplace domain transfer function block is one of the most basic blocks. It shows the mathematical relationship between the input U and the output Y . In a transfer function block, the output Y is the transfer function $G_1(s)$ multiplied by the input U , or $Y = G_1(s)U$.

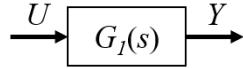


Figure 5.1 Transfer function block with input (U) and output (Y) signals.

Sometimes the output of one transfer function is the input to another. For example, $E = G_1 U$ and $Y = G_2 E$. An input in the time domain is multiplication in the Laplace domain so $Y = G_2 E$ can be written as $Y = G_2 G_1 U$. Both of the block diagrams in Figure 5.2 are equivalent.

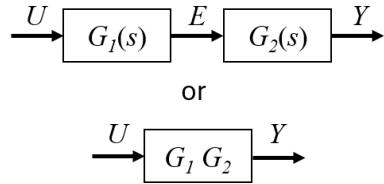


Figure 5.2 An input in the time domain is multiplication in the Laplace domain

Addition in the time domain is also addition in the Laplace domain. Subtraction in the time domain is also subtraction in the Laplace domain. Addition or subtraction blocks show these relationships in block diagram form. Figure 5.3 shows the block diagram for $Y = G_1 U - G_2 E$.

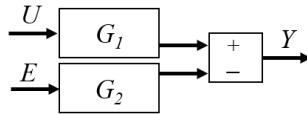


Figure 5.3 Subtraction in a block diagram

Besides subtraction blocks, there are blocks for every mathematical operation. The subtraction block is especially useful in feedback diagrams such as Figure 5.4.

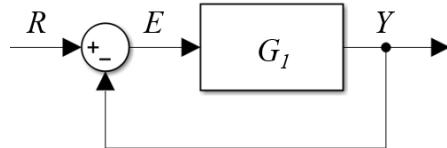


Figure 5.4 A basic feedback diagram

In Figure 5.4, the output Y is compared with the reference input R , and the difference is the error $E = R - Y$. The error E is fed back into the transfer function G_1 to produce the output $Y = G_1 E = G_1(R_Y)$. Determining the relationship between R and Y requires solving an algebraic loop. As shown in Example 5.1.1, the transfer function from R to Y for the basic feedback diagram of Figure 5.4 is

$$\frac{Y}{R} = \frac{G_1}{1 + G_1} \quad (5.1)$$

5.2 Feedforward Path, Feedback Path, and the Feedback Loop

Basic Feedback Controller Transfer Function

Example 5.1.1. Find the transfer function for the basic feedback controller

Find the transfer function for the basic feedback controller of Figure 5.4.

Solution: The goal is to find the transfer function from the input R to the output Y . The output of the subtraction block is

$$E = R - Y$$

The output Y of the transfer function block is

$$\begin{aligned} Y &= G_1 E \\ &= G_1(R - Y) \\ &= G_1 R - G_1 Y \end{aligned}$$

so

$$\begin{aligned} Y + G_1 Y &= G_1 R \\ (1 + G_1)Y &= G_1 R \\ Y &= \frac{G_1}{1 + G_1} R \\ \frac{Y}{R} &= \frac{G_1}{1 + G_1} \end{aligned}$$

With this simplified transfer function, the basic feedback diagram can be replaced by a single block as shown in Figure 5.5.

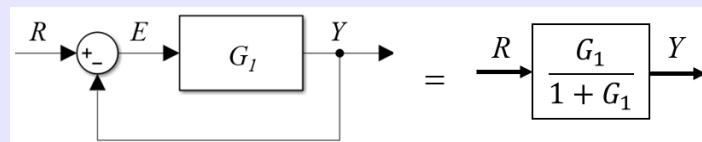


Figure 5.5 The basic feedback diagram can be replaced by a single block.

5.2 Feedforward Path, Feedback Path, and the Feedback Loop

Figure 5.6 shows a block diagram with one feedback path, one feedback loop, and two feedforward paths. Block diagrams can have multiple feedforward paths, feedback paths, and feedback loops. **Feedback paths** contain signals and blocks whose outputs are fed back into the block diagram at an earlier part of the path. **Feedforward paths** are all paths that are not parts of feedback paths. **Feedback loops** include both the feedback path and the part of the feedforward path that is affected by the feedback path. The output of a feedback loop depends on itself. Identifying these paths and loops can simplify the block diagram analysis and calculations. Simplifying rules are provided in Table 5.1. These rules will be used throughout this chapter and the remainder of this book.

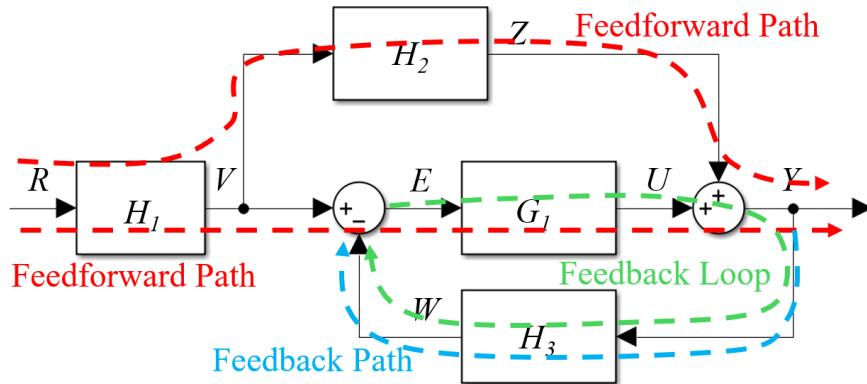


Figure 5.6 Block diagrams can have feedforward paths, feedback paths, and feedback loops.

Table 5.1 Rules to simplify the analysis of block diagrams with feedback loops. The examples refer to Figure 5.6.

Path	Rule	Example
Feedforward	Blocks in the same feedforward path are multiplied together in the transfer function numerator	$H_1 G_1 + H_1 H_2$...
Feedback	Blocks in the same feedback loop are multiplied together then added to one in the transfer function denominator	$\frac{...}{1+G_1 H_3}$
Feedback + Feedforward	The numerator and denominator are combined	$\frac{Y}{R} = \frac{H_1 G_1 + H_1 H_2}{1+G_1 H_3}$

In place of proving these rules, we will demonstrate how they are obtained by deriving them for the block diagram of Figure 5.6. This derivation is done in Example 5.2.1.

Deriving the Block Diagram Rules of Table 5.1 for Figure 5.6.

Example 5.2.1. Deriving the block diagram rules

Derive the block diagram rules of Table 5.1 for the block diagram of Figure 5.6 to show that the transfer function from R to Y is

$$\frac{Y}{R} = \frac{H_1 G_1 + H_1 H_2}{1+G_1 H_3}$$

Solution: This example will use basic block diagram analysis to obtain the same results found by the rules of Table 5.1. From the block diagram of Figure 5.6, the following signals are calculated as

5.2 Feedforward Path, Feedback Path, and the Feedback Loop

$$\begin{aligned}V &= H_1 R \\Z &= H_2 V \\E &= V - W \\U &= G_1 E \\Y &= Z + U \\W &= H_3 Y\end{aligned}$$

To derive the transfer function from R to Y , we must eliminate the intermediate variables V , Z , E , U , and W . We do so by substitution. For example, we substitute $Z = H_2 V$ into $Y = Z + U$ to get $Y = H_2 V + U$. We substitute $U = G_1 E$ into $Y = H_2 V + U$ to get $Y = H_2 V + G_1 E$, and so forth. The process of substitution continues until all intermediate variables are eliminated, resulting in the following:

$$\begin{aligned}Y &= H_2 H_1 R + G_1 H_1 R - G_1 H_3 Y \\Y + G_1 H_3 Y &= H_2 H_1 R + G_1 H_1 R \\Y(1 + G_1 H_3) &= (H_2 H_1 + G_1 H_1)R \\\frac{Y}{R} &= \frac{H_2 H_1 + G_1 H_1}{1 + G_1 H_3}\end{aligned}$$

Which can be rearranged by the commutative property of multiplication and addition to be

$$\frac{Y}{R} = \frac{H_1 G_1 + H_1 H_2}{1 + G_1 H_3}$$

This is the same transfer function obtained by applying the simplifying rules of Table 5.1.

5.2.1 Block Diagrams with Multiple Inputs and Outputs

Block diagrams can have multiple input and output signals. This section provides a couple examples to show how the rules of Table 5.1 apply to block diagrams with multiple inputs and outputs.

Block Diagram with Two Inputs

Example 5.2.2. A block diagram with two inputs

Calculate the transfer functions from U_1 and U_2 to Y for the following block diagram using the rules of Table 5.1.

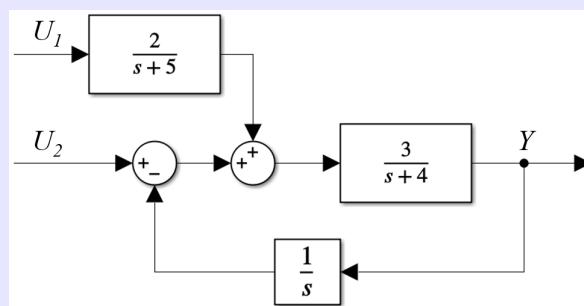


Figure 5.7 A block diagram with two inputs U_1 and U_2 and one output Y

Solution: From the input signal U_1 to the output Y , there are two transfer function blocks in the forward path:

$$G_1 = \frac{2}{s+5}$$

and

$$G_2 = \frac{3}{s+4}$$

From the input signal U_2 to the output Y there is only one transfer function, G_2 , in the feedforward path. In the feedback loop for both inputs, there is an integrator $G_3 = \frac{1}{s}$ and the transfer function G_2 . Using the rules from Table 5.1, the transfer function from U_1 to Y is therefore

$$\begin{aligned} Y_1 &= \frac{G_1 G_2}{1 + G_2 G_3} U_1 \\ &= \frac{\frac{2}{s+5} \frac{3}{s+4}}{1 + \frac{1}{s} \frac{3}{s+4}} U_1 \end{aligned}$$

which can be simplified algebraically to

$$\frac{Y_1}{U_1} = \frac{6s}{s^3 + 9s^2 + 23s + 15}$$

The output Y_1 is only part of the output Y . Similarly, the transfer function from U_2 to Y is

$$\begin{aligned} Y_2 &= \frac{G_2}{1 + G_2 G_3} U_2 \\ &= \frac{\frac{3}{s+4}}{1 + \frac{1}{s} \frac{3}{s+4}} U_2 \end{aligned}$$

5.2 Feedforward Path, Feedback Path, and the Feedback Loop

which simplifies to

$$\frac{Y_2}{U_2} = \frac{3s}{s^2 + 4s + 3}$$

The signal Y_2 is the other part of the output. Due to the add-block in the forward path, $Y = Y_1 + Y_2$:

$$Y = \frac{6s}{s^3 + 9s^2 + 23s + 15} U_1 + \frac{3s}{s^2 + 4s + 3} U_2$$

The next example demonstrates how to calculate the transfer functions for a block diagram with two outputs.

Block Diagram with Two Outputs

Example 5.2.3. A block diagram with two outputs

Use the rules of Table 5.1 to calculate the transfer function from the input U to each of the outputs Y_1 and Y_2 .

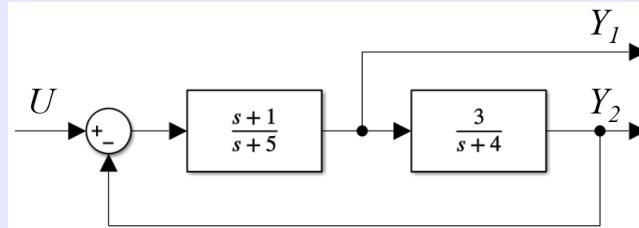


Figure 5.8 A block diagram with one input U and two outputs Y_1 and Y_2

Solution: The feedback loop for both outputs is the same. It contains both transfer functions

$$G_1 = \frac{s+1}{s+5}$$

and

$$G_2 = \frac{3}{s+4}$$

The feedforward path for the first output Y_1 only has the transfer function G_1 . The feedforward path for the second output Y_2 contains both transfer functions G_1 and G_2 . Therefore, using the rules of Table 5.1, the transfer function from U to Y_1 is

$$\begin{aligned} Y_1 &= \frac{G_1}{1 + G_1 G_2} \\ &= \frac{\frac{s+1}{s+5}}{1 + \frac{s+1}{s+5} \frac{3}{s+4}} \end{aligned}$$

which can be simplified to

$$\frac{Y_1}{U} = \frac{s^2 + 5s + 4}{s^2 + 12s + 23}$$

Similarly, the transfer function from U to Y_2 is

$$\begin{aligned} Y_2 &= \frac{G_1 G_2}{1 + G_1 G_2} \\ &= \frac{\frac{s+1}{s+5} \frac{3}{s+4}}{1 + \frac{s+1}{s+5} \frac{3}{s+4}} \end{aligned}$$

which can be simplified to

$$\frac{Y_2}{U} = \frac{3s + 3}{s^2 + 12s + 23}$$

Unlike systems with multiple inputs, the two resulting transfer functions are not combined into one but remain separate.

5.2.2 Block Diagrams with Multiple Feedback Paths

It is not uncommon to encounter control block diagrams with multiple feedback loops. Feedback loops can be nested within other loops and / or they can be in series. This section uses an example problem to show how the rules of Table 5.1 are used to sequentially solve block diagrams with multiple feedback paths.

For nested feedback loops, the block diagram analysis is simplified by solving the innermost loop first and replacing it with its simplified transfer function. Then the next innermost loop is replaced by its transfer function, the next innermost loop is solved, and so forth. This process is demonstrated in the following example.

Nested Feedback Loops

Example 5.2.4. A block diagram with nested feedback loops

Use the rules of Table 5.1 to find the transfer function from U to Y for the following block diagram:

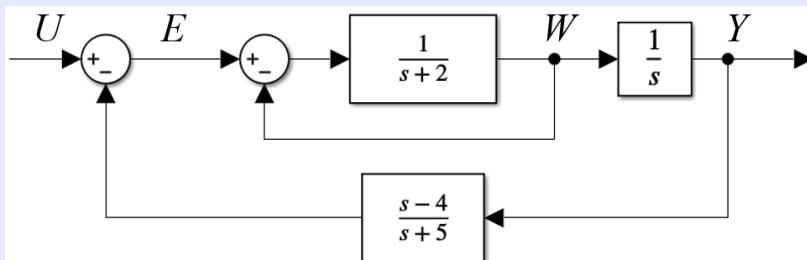


Figure 5.9 A block diagram with nested feedback loops

5.2 Feedforward Path, Feedback Path, and the Feedback Loop

Solution: The analysis begins by finding the transfer function for the inner loop from E to W :

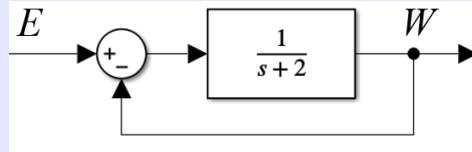


Figure 5.10 The innermost loop is the portion of the block diagram from E to W .

The feedforward path contains only the transfer function:

$$G_1 = \frac{1}{s+2}$$

The feedback loop also contains only the transfer function G_1 . Applying the rules of Table 5.1, the transfer function from E to W is

$$\begin{aligned} \frac{W}{E} &= \frac{G_1}{1 + G_1} \\ &= \frac{\frac{1}{s+2}}{1 + \frac{1}{s+2}} \end{aligned}$$

which can be simplified algebraically to

$$\frac{W}{E} = \frac{1}{s+3}$$

We can replace the innermost loop with the transfer function from E to W as shown below:

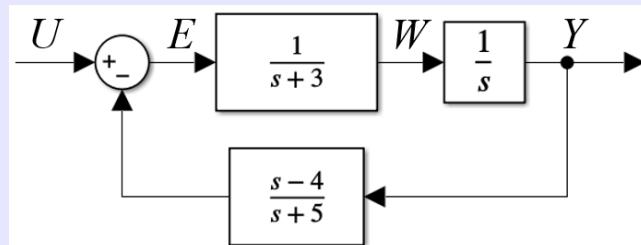


Figure 5.11 The innermost loop from E to W has been replaced by its transfer function.

With this substitution, only one feedback loop remains. The feedforward path includes two transfer functions:

$$G_2 = \frac{1}{s+3}$$

and

$$G_3 = \frac{1}{s}$$

In addition to G_2 and G_3 , the feedback loop also includes

$$G_4 = \frac{s-4}{s+5}$$

Applying the rules of Table 5.1, the transfer function from U to Y is

$$\begin{aligned} \frac{Y}{U} &= \frac{G_2 G_3}{1 + G_2 G_3 G_4} \\ &= \frac{\frac{1}{s+3} \frac{1}{s}}{1 + \frac{1}{s+3} \frac{1}{s} \frac{s-4}{s+5}} \end{aligned}$$

which can be simplified to

$$\frac{Y}{U} = \frac{s+5}{s^3 + 8s^2 + 16s - 4}$$

Block diagrams with feedback loops in series can be solved by replacing each feedback loop with its corresponding transfer function. The order is arbitrary. If block diagrams have both series and nested feedback loops, the innermost feedback loops should be solved first and replaced with their transfer functions.

5.3 Control Systems

Block diagrams offer a useful way to visually model and analyze control systems. Control systems are dynamic systems that have some type of control or regulation. Common types of control are open-loop control, which is also called feedforward control, feedback control, and combined feedforward / feedback control. Proportional-Integral-Derivative (PID) control is a type of feedback control. In each of these control systems in this section, we will adopt the following notation:

- Y : output signal of the dynamic system that is being controlled
- R : reference command signal (desired final value of Y)
- U : control command or input signal to the dynamic system
- E : error signal or difference between R and Y
- W : disturbance or uncontrolled input signal to the dynamic system

For this section, we will also adopt the following notation for the transfer functions in the control block diagrams:

- $G(s)$: Transfer function for the dynamic system (or “Plant”) that is being controlled

5.3 Control Systems

- $C(s)$: Transfer function for the feedback (and sometimes feedforward) controller
- $K(s)$: Transfer function for the feedforward controller

5.3.1 Feedforward Controllers

The first type of control system that this section presents is the feedforward or open-loop controller. It is shown in Figure 5.12.

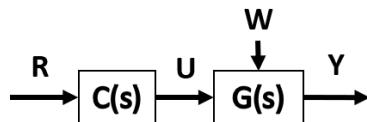


Figure 5.12 The feedforward controller

Like other controllers, often the goal of the feedforward controller is to cause the output Y to eventually match the reference command R , *i.e.*, $\lim_{t \rightarrow \infty} Y = R$ (or if R is a constant or step input $Y = \lim_{s \rightarrow 0} C(s)G(s)R$ by the final value theorem, see Section 2.6.1). To do so, the controller $C(s)$ is typically some type of inverse of the plant $G(s)$, such that the product is $C(s)G(s) = 1$. Because the controller has no information about the output Y or the error $E = R - Y$, it cannot correct for disturbances W . This is one major downside of feedforward or open-loop controllers. There are a number of benefits of feedforward controllers. They do not require sensors to measure the output Y . They cannot become unstable due to feedback problems, and they can have fast response times.

A propane stove is an example of a feedforward control system. The gas knob is the controller. Turning the knob to a set position sets the rate of fuel from the burners. There is no automatic feedback, such as a temperature measurement, that changes the setting of the knob to regulate the temperature of the food.

5.3.2 Feedback Controllers

Another type of control system is a feedback controller. One example is shown in Figure 5.13.

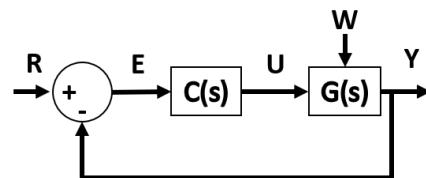


Figure 5.13 An example of a basic feedback controller

A key benefit of feedback systems is that they can be designed to cancel out the effect of disturbances W and enable the output Y to eventually track the reference R . The disturbance W is part of the feedback loop, and so the controller can be designed to eliminate the error E between the reference input R and the plant output Y despite disturbances W . An example of PI cruise control in the Example 5.3.1 will demonstrate this ability. Another advantage is that feedback controllers can stabilize unstable systems. For example, a feedback controller is required to balance an inverted pendulum. A feedforward controller could not balance it.

Feedback controllers also have disadvantages. They require sensors, which could be expensive. They can be difficult and time-consuming to tune and adjust. In challenging control problems, this can cost an engineer weeks or more of work. Another disadvantage is that they can potentially cause even stable dynamic systems to become unstable if they are not tuned properly. The loud ringing that occurs when a microphone and speaker are placed near each other is a well known example of feedback causing instability.

An electric oven is an example of a system with a feedback controller. A thermocouple inside the oven measures the temperature (Y) of the oven. This information is fed back to the oven controller. If there is a difference (E) between the set-point temperature (R) and the oven temperature (Y), the oven controller adjusts the amount of electricity (U) supplied to the oven's heating elements to regulate the temperature.

5.3.3 The Famous PID Controller

The PID (Proportional-Integral-Derivative) controller is a popular type of feedback controller. It does not require knowledge of the dynamics of the system that it is controlling. The proportional gain is k_P , the integral gain is k_I , and the derivative gain is k_D . Then the PID controller is just a feedback controller in which the controller transfer function is

$$C(s) = k_P + k_I \frac{1}{s} + k_D s \quad (5.2)$$

and the block diagram for a typical PID controller is shown in Figure 5.14.

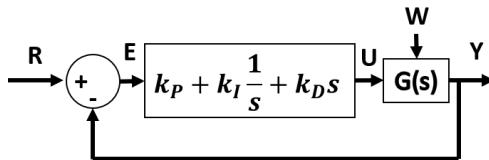


Figure 5.14 Block diagram of a PID controller

Numerical derivatives tend to be noisy signals. The derivative term s is therefore often replaced by a smoothed derivative term $\frac{s}{\tau_D s + 1}$, and Eq. (5.2) becomes

$$C(s) = k_P + k_I \frac{1}{s} + k_D \frac{s}{\tau_D s + 1} \quad (5.3)$$

The constant τ_D is a smoothing filter parameter that can be tuned by the designer. It is usually a small number, because $\lim_{\tau_D \rightarrow 0} \frac{s}{\tau_D s + 1} = s$.

Notice that the input to the PID controller is the error $E = R - Y$. As a result of the proportional term, if the error is large, the control command will be large. In the oven example, if the oven temperature is much less than the set-point temperature, the error is large, and the oven will supply a large amount of electricity to the heating elements. If the oven temperature is only slightly less than the set-point temperature, the error is small, and the amount of electricity supplied to the heating elements will be small. In proportional control, the electricity supplied to the heating elements is proportional to the difference between the set-point temperature and the actual oven temperature.

The integral part $k_I \frac{1}{s}$ of the PID controller causes the control signal to be proportional to the integral ($\int E dt$) of the error E . For example, if the error E is constant for a long time t , the integral of the error is

5.3 Control Systems

the product Et . Therefore, as time increases, the control command U increases. This integration property of the PID controller helps the controller eliminate steady-state errors such that $\lim_{t \rightarrow \infty} E = 0$.

The objective of the derivative part $k_D s$ of a PID controller is to dampen the effect of fast changes in the error and to decrease the likelihood of overshooting the set-point. For example, if a positive error is decreasing quickly in time, its derivative $\frac{dE}{dt}$ will be a large negative number. The derivative term in the PID controller will therefore also be a large negative number that is subtracted from the proportional and integral control commands. This will lessen their effects and cause the error to converge more slowly, which may help prevent overshooting the set-point.

Tuning PID controllers involves selecting values for the gains k_P , k_I , and k_D (and τ_D if Eq. (5.3) is used). These gains are often selected by trial and error. As will be discussed in Chapter 13, however, when a model of the system is available, there are better methods for selecting values for the gains. When tuning PID controllers by trial and error, it is often good to set k_I and k_D to zero initially and focus first on adjusting the value of k_P . The proportional gain k_P should be slowly increased from $k_P = 0$ until a desirable response time to a step input is achieved. Then the integral gain k_I can be slowly increased to eliminate steady-state error. Finally, k_D is adjusted to dampen oscillation and overshoot.

Proportional control is a variation of the PID controller where both integral k_I and derivative k_D gains are set to zero. In PI control, only the derivative gain k_D is zero. Other variations of the PID controller are possible. The following example shows how PI control can be used in a cruise control application for an automobile to cancel the effect of road grade disturbances.

PI Cruise Control of a Truck on a Hill

Example 5.3.1. PI cruise control of a truck on a hill

Show that a PI cruise control in an automobile can cancel the effects of an unknown road grade.

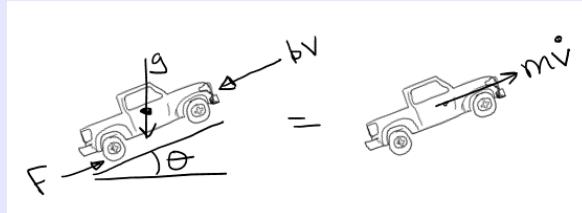


Figure 5.15 Dynamic force balance for the truck on a hill

For small deviations around the set-point speed v_d , we can linearize Eq. (1.17) to get

$$\dot{v} = -bv + F - mg \sin \theta$$

where the damping coefficient b includes effects from both the rolling resistance and air drag. In this equation, we can control the drivetrain force F , and so it is the control input $U = F$. We cannot control the unknown road grade θ . It is the disturbance (we will use $W = mg \sin \theta$ as the disturbance). The set-point speed v_d is the reference input $R = v_d$, and the actual vehicle speed v is the dynamic system output $Y = v$. The car is the plant, and its transfer function from $F - mg \sin \theta$ to v is found by taking the Laplace transform of the linearized equation.

$$\frac{v}{F - mg \sin \theta} = \frac{1/m}{s + b/m}$$

or in block diagram form:

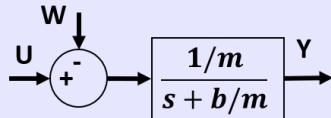


Figure 5.16 Block-diagram form of the truck dynamics

The transfer function for the PI controller is

$$C(s) = k_P + \frac{k_I}{s} \quad (5.4)$$

and the block diagram of the complete PI cruise control system is shown in Figure 5.17.

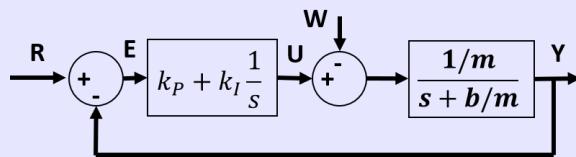


Figure 5.17 Block-diagram of the complete PI cruise control system

We can solve the block diagram of Figure 5.17 to get the transfer function from U and W to Y :

$$Y = \frac{\frac{k_p}{m}s + \frac{k_I}{m}}{s^2 + \left(\frac{b}{m} + \frac{k_p}{m}\right)s + \frac{k_I}{m}} R - \frac{\frac{1}{m}s}{s^2 + \left(\frac{b}{m} + \frac{k_p}{m}\right)s + \frac{k_I}{m}} W$$

We need to show that the PI controller can cancel the effects of the unknown road grade W and track the reference so that eventually $Y = R$. For these objectives to be satisfied, three requirements must be met.

1. The transfer functions from R and W to Y must be stable
2. The transfer function from R to Y must converge to one in the limit as $s \rightarrow 0$
3. The transfer function from W to Y must converge to zero in the limit as $s \rightarrow 0$

The denominator of both transfer functions is $s^2 + \left(\frac{b}{m} + \frac{k_p}{m}\right)s + \frac{k_I}{m}$. This is stable if and only if the coefficients $\left(\frac{b}{m} + \frac{k_p}{m}\right)$ and $\frac{k_I}{m}$ are greater than zero. We can choose the values of the gains k_P and k_I ; therefore, we can ensure that the transfer functions are stable.

For the remaining requirements, we need to analyze the transfer functions as $s \rightarrow 0$:

$$\lim_{s \rightarrow 0} \frac{Y}{R} = \frac{\frac{k_I}{m}}{\frac{k_I}{m}} = 1$$

$$\lim_{s \rightarrow 0} \frac{Y}{W} = \frac{0}{\frac{k_I}{m}} = 0$$

5.3 Control Systems

In the limit, the transfer function from R to Y is one, and therefore, the actual speed v will eventually converge to the set-point speed v_d . In the limit, the transfer function from W to Y is zero, and therefore, the effects of the unknown road grade W are canceled by the PI controller. A Simulink implementation in Figure 5.18 shows these results. In the simulation, $m = 1000 \text{ kg}$, $b = 10 \text{ kg/s}$, $g = 9.8 \text{ m/s}^2$, $\theta = 0.1 \text{ rad}$, $v_d = 25 \text{ m/s}$, $k_P = 300$, and $k_I = 30$. Simulink used a variable time-step solver.

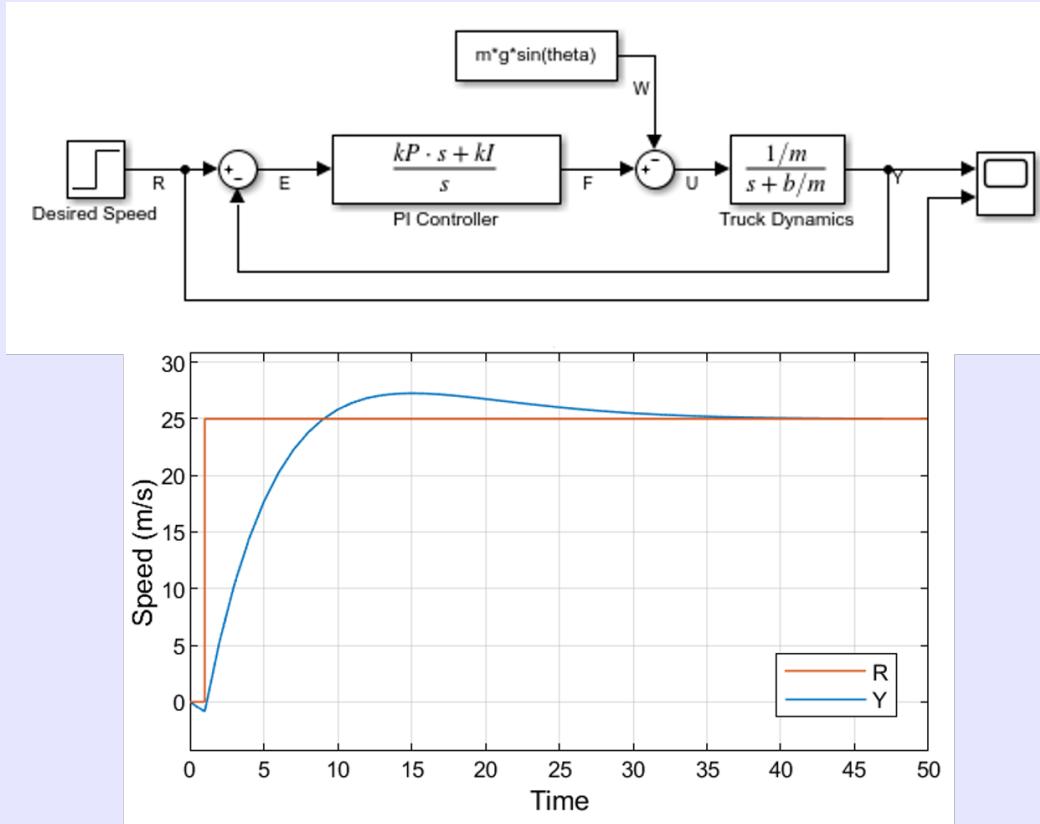


Figure 5.18 Simulink model and results of the PI cruise control

5.3.4 Integrator Anti-Windup

There are limits to the amount of control effort an actuator can supply. An engine cannot supply an infinite amount of torque to the wheels of a truck. The amount of electricity that can be supplied to the heating elements of an oven is limited, etc. When an actuator has reached its limits, it is **saturated**. Saturation limits affect the performance of controllers. Integral controllers are negatively affected by actuator saturation. During saturation, the integral term continues to accumulate error. This large accumulation is called **integrator windup**. Integrator windup causes the controller to overshoot the set-point target as this accumulated error is unwound. There are many techniques to minimize the effect of integrator windup. Two methods are clamping (conditional integration) and back-calculating.

The clamping method (or conditional integration) detects events when the controller is saturated.

Integration is paused during these events. A PID controller with saturation clamping is represented mathematically as follows:

$$\dot{I}(t) = E(t), \text{ where } \begin{cases} E(t) = E(t), & \text{if } U_{\min} < U(t) < U_{\max} \\ E(t) = 0, & \text{otherwise} \end{cases} \quad (5.5)$$

$$U(t) = k_P E(t) + k_I I(t) + K_D \frac{dE}{dt} \quad (5.6)$$

The block diagram of Figure 5.19 shows the back-calculating method. The gain k_B is a user-tuned back-calculating parameter. The back-calculating anti-windup method compares the control command signals before (U_1) and after (U_2) saturation. If they are different, the scaled difference is subtracted from the scaled error before integration.

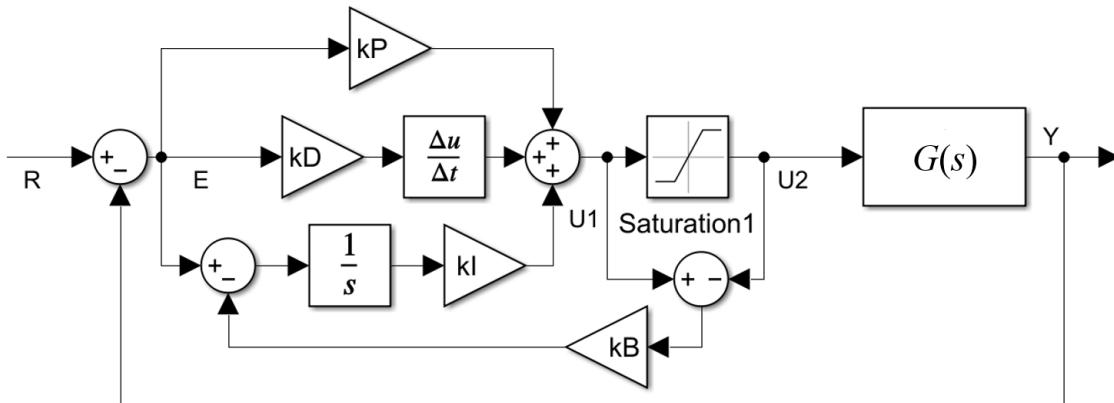


Figure 5.19 A PID controller with back-calculating integrator anti-windup

The following example demonstrates the effectiveness of the back-calculating technique to decrease the effect of integrator windup.

Integrator Anti-Windup

Example 5.3.2. PID control with integrator anti-windup

Repeat Example 5.3.1 with the following changes: (1) add a derivative controller with a gain of $k_D = 5$, (2) the engine is only capable of supplying forces in the range of zero to 3000 N, (3) add integrator anti-windup, and (4) perform the analysis in MATLAB using the forward Euler numerical approximation.

Solution: Figure 5.20 shows a block diagram of the PID cruise controller for the truck on a hill with back-calculating anti-windup.

5.3 Control Systems

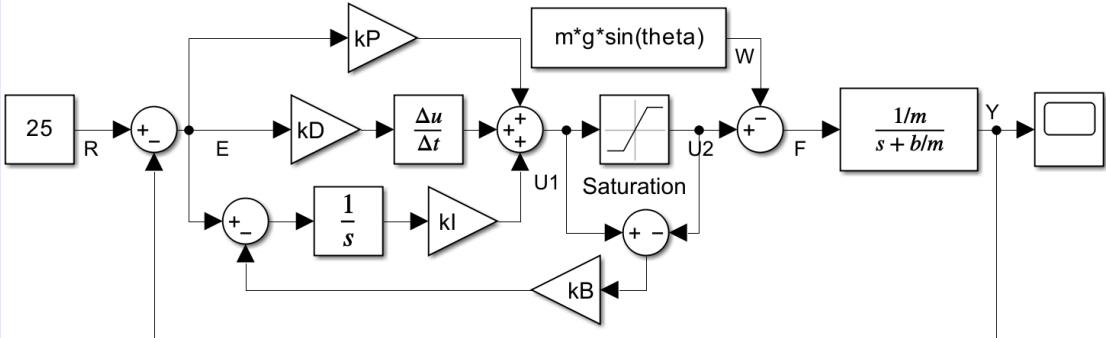


Figure 5.20 Simulink model of the PID cruise control with back-calculating anti-windup

To perform the analysis in MATLAB instead of Simulink, we must first convert the block diagram to discrete state-space using the forward Euler numerical approximation. The control command $U_{1,k}$ is

$$U_{1,k} = k_P E_k + k_I I_k + k_D \frac{E_k - E_{k-1}}{\Delta t}$$

and the saturated command $U_{2,k}$ is

$$U_{2,k} = \max(0, \min(U_{1,k}, 3000))$$

The integral I_k is calculated by the forward Euler approximation to be

$$I_{k+1} = I_k + \Delta t (E_k - k_B (U_{1,k} - U_{2,k}))$$

where $E_k = Y_k - R_k$. The forward Euler numerical approximation of the truck's transfer function $\frac{Y}{F} = \frac{1/m}{s+b/m}$ is

$$Y_{k+1} = \left(1 - \frac{b}{m} \Delta t\right) Y_k + \frac{1}{m} \Delta t F_k$$

The timestep is Δt , the truck force is $F_k = U_{2,k} - W_k$, and the subscript k indicates the value of the signal at the discrete time t_k . With these equations, the following MATLAB code implements the block diagram of Figure 5.20.

```

clear %clear all variables
close all %close all open figures
clc %clear the workspace

%Set the constants
m = 1000; %(kg) mass
b = 10; %(kg/s) drag coefficient
g = 9.8; %(m/s^2) gravitational acceleration

```

```

theta = 0.1; %(rad) road slope angle
R = 25; %(m/s) desired setpoint speed
kP = 300; %(-) Proportional gain
kI = 30; %(-) Integral gain
kD = 5; %(-) Derivative gain
kB = 1; %(-) Back-calculating gain
Umin = 0; %(N) minimum engine force
Umax = 3000; %(N) maximum engine force
dt = 0.001; %(s) integration timestep
t = 0:dt:50; %(s) time vector
N = length(t); %Number of timesteps

%Set the initial conditions
I = 0; %Initial integral value
Elast = 0; %Initial value of the previous timestep error
x = 0; %(m/s) Initial value of the truck speed

%Allocate memory to store the output
y = zeros(1,N);

%Run the simulation
for ii = 1:N
    %Store the output speed of the truck
    y(ii) = x;
    %Get the error
    E = (R-x);
    %Calculate the control command U1
    U1 = kP*E+kI*I+kD*(E-Elast)/dt;
    %Update the previous timestep error
    Elast = E;
    %Get the control command U2
    U2 = max(Umin, min(U1, Umax));
    %Update the integral
    I = I + dt*(E-kB*(U1-U2));
    %Calculate the truck forces
    F = U2 - m*g*sin(theta);
    %Update the truck state equation
    x = (1-b/m*dt)*x+1/m*dt*F;
end

%Graph the trajectory of the truck's speed
figure %Open a figure to draw on
plot(t, y) %Plot the truck's speed
grid on %Turn on the grid
xlabel('Time (s)') %Label the x-axis
ylabel('Truck Speed (m/s)') %Label the y-axis

```

The graph that results from the MATLAB simulation of the truck cruise control is shown below. Because of the limitations on the engine force and the back-calculating PID controller, the truck reaches the set-point speed of 25 m/s without overshooting it.

5.3 Control Systems

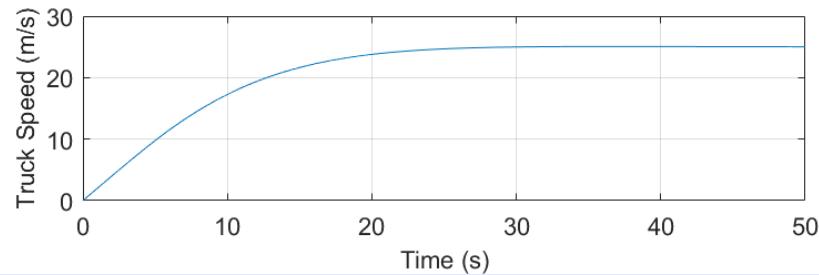


Figure 5.21 Truck speed that results from the PID cruise controller with back-calculating

5.3.5 Modeling Nonlinear ODEs using Integrators

An **integrator**, *i.e.*, the Laplace transfer function $\frac{1}{s}$, is useful for modeling linear or nonlinear dynamic systems. The output of an integrator is a state variable (*e.g.* x_i), and the input is a state-derivative (*e.g.* \dot{x}_i). Therefore, an integrator can be used to calculate the state in a state-space system. This can facilitate modeling linear or nonlinear systems, as demonstrated in the following example:

Block Diagram Modeling of Nonlinear Dynamic Systems

Example 5.3.3. Use integrators to model a nonlinear dynamic system

Create a Simulink model to solve the following nonlinear dynamic system:

$$\ddot{x}_1 + 2|\dot{x}_1|\dot{x}_1 - 0.5x_1x_2 = 5\cos(6t) \quad (5.7)$$

$$\dot{x}_2 + 4x_2 - x_1 = 0 \quad (5.8)$$

Solution: The following Simulink model of Figure 5.22 uses integrators. The input $ddx1$ to the top left integrator is the signal \ddot{x}_1 . The output $dx1$ is \dot{x}_1 , which is also the input to the top right integrator. The output $x1$ of the top right integrator is the state variable x_1 . The input $dx2$ to the bottom integrator is \dot{x}_2 . The output is the state variable x_2 . Notice how the input $ddx1$ is also equal to $-(2|\dot{x}_1|\dot{x}_1 - 0.5x_1x_2 + 5\cos(6t))$ which was determined by solving Eq. (5.7). The input $dx2$ is also equal to $-(4x_2 - x_1)$, which is found by solving Eq. (5.8) for \dot{x}_2 .

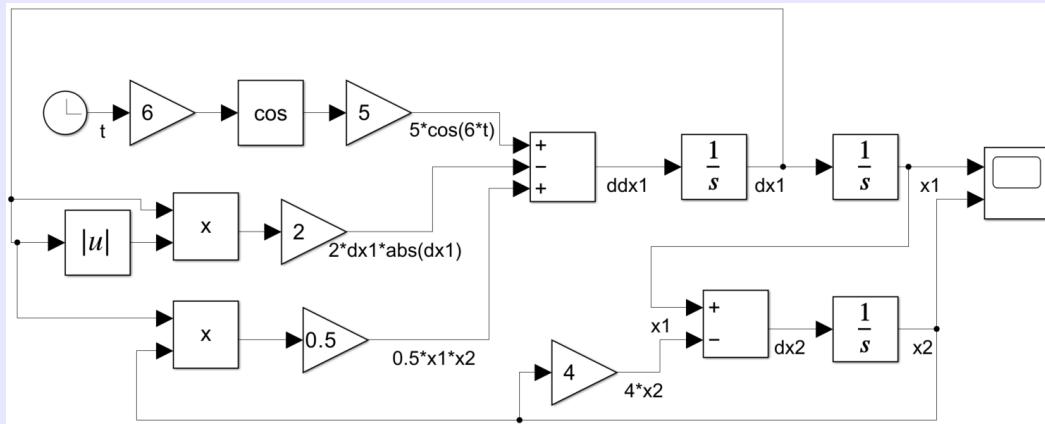


Figure 5.22 The solution to nonlinear modeling is to use integrators

5.4 Block Diagrams of Discrete-Time Systems

Block diagrams can model discrete-time systems. There are discrete state-space blocks, z-transfer function blocks, and many other discrete-time blocks. In continuous-time, the integrator aided in modeling linear and nonlinear systems. In discrete-time, the unit delay $\frac{1}{z}$ plays an equivalent role. This is demonstrated in the following example.

Block Diagram Modeling of Nonlinear Discrete-Time Systems

Example 5.4.1. Use unit delays to model a nonlinear discrete-time system

Create a Simulink model to solve the following nonlinear discrete-time system:

$$x_{1,k} + 2|x_{1,k-1}|x_{1,k-1} - 0.5x_{1,k-2}x_{2,k-1} = 5\cos(6t_k) \quad (5.9)$$

$$x_{2,k} + 4x_{2,k-1} - x_{1,k-2} = 0 \quad (5.10)$$

Solution: Although the above equations are for an unstable system, we can solve them using unit delays just like Example 5.3.3 solved its equations using integrators. Figure 5.23 shows the solution.

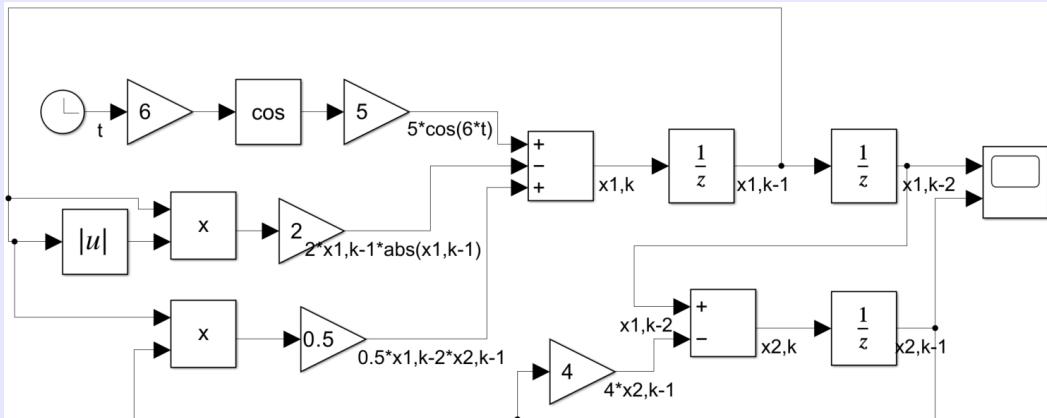


Figure 5.23 The solution to nonlinear discrete-time modeling is to use unit delays

Chapter 6

Modeling Dynamic Mechatronic Systems

Contents

6.1	Translational Dynamics and Vibrations	156
6.1.1	Modeling a Mass	157
6.1.2	Modeling a Spring	157
6.1.3	Modeling a Damper	158
6.1.4	d'Alembert's Principle	158
6.1.5	Modeling a Mass-Damper System	158
6.1.6	Modeling a Spring-Damper System	159
6.1.7	Modeling a Mass-Spring-Damper System	160
6.1.8	Frequency Response of a Mass-Spring-Damper System	163
6.1.9	Mass-Spring-Damper Systems with Multiple Masses	166
6.1.10	Mass-Spring-Damper Systems Modeled as a Longitudinal Wave Propagation	169
6.2	Rotational Dynamics and Vibrations	169
6.3	Modeling Dynamic Electrical Circuits	171
6.3.1	Basics of Capacitors	172
6.3.2	Basics of Inductors	174
6.3.3	Inductors store energy in the form of a magnetic field	174
6.3.4	Units of inductance and symbols for an inductor	175
6.4	Modeling Electrical Circuits with $\dot{\mathbf{x}} = \mathbf{Ax} + \mathbf{Bu}$	177
6.4.1	Modeling Capacitors	177
6.4.2	Modeling Inductors	178
6.4.3	Modeling Resistors	178
6.4.4	State-Space Modeling of Electrical Circuits	178
6.4.5	Modeling Electrical Circuits in the Laplace Domain	181
6.4.6	Equivalent Impedance of a Circuit	186
6.4.7	Electrical Circuits with Switches and Diodes	187
6.4.8	Switches	187
6.4.9	Diodes	187

6.4.10 Buck Converter: A Switch and Diode Circuit Example	189
6.5 Summary: Steps for Solving Electrical Circuits	193
6.6 Modeling Batteries using Equivalent Circuit Models	197
6.7 Modeling Electrical Circuits with Op Amps	199
6.7.1 Rules for Modeling Op Amps with Feedback	200
6.8 Modeling DC Motors	207
6.9 Modeling Dynamic Diffusion and Heat Transfer	210
6.9.1 Flux	211
6.9.2 Flux Driving Forces	211
6.9.3 Conservation	212
6.9.4 Conservation of Mass	212
6.9.5 Conservation of Energy	213
6.9.6 Conservation and Flux in Finite Volumes	213
6.9.7 The Heat Equation	215
6.9.8 The Finite Volume Method	216
6.9.9 Boundary Conditions	216
6.9.10 Neumann Boundary Conditions	217
6.9.11 Dirichlet Boundary Conditions	217
6.9.12 Diffusion Equation Solution by $\dot{x} = Ax + Bu$	217
6.9.13 Heat Equation Solution by $\dot{x} = Ax + Bu$	218
6.9.14 Example: Carburization	219
6.9.15 Problem Statement	220
6.9.16 Finite Volume Solution	220
6.9.17 MATLAB Code	222
6.9.18 Simulation Results	224
6.9.19 Analytical Solution	224

This chapter introduces modeling techniques for dynamic mechatronic systems. It presents some general modeling techniques for mechanical translations, rotations, and vibrations. It discusses modeling electrical circuits with capacitors, resistors, inductors, switches, operational amplifiers (op amps), batteries, and power supplies. It models electrical DC motors. The topics of this chapter are general for broad application. More specific modeling of fixed wing airplanes and drones is presented in Chapter 7.

6.1 Translational Dynamics and Vibrations

Many systems are made of mechanical components that store, dissipate, or supply energy. Machines, vehicles, and even buildings, bridges, and other structures can be analyzed using these basic components. These components appear in any system that experiences mechanical vibrations. A common example is a car suspension. In its simplest form, a car suspension can be modeled as a mass-spring-damper system, as can be seen in Figure 6.1. The body of the vehicle is the mass, the suspension and tire make up the spring and damper system. The road profile causes vertical forces that excite the suspension.

6.1 Translational Dynamics and Vibrations

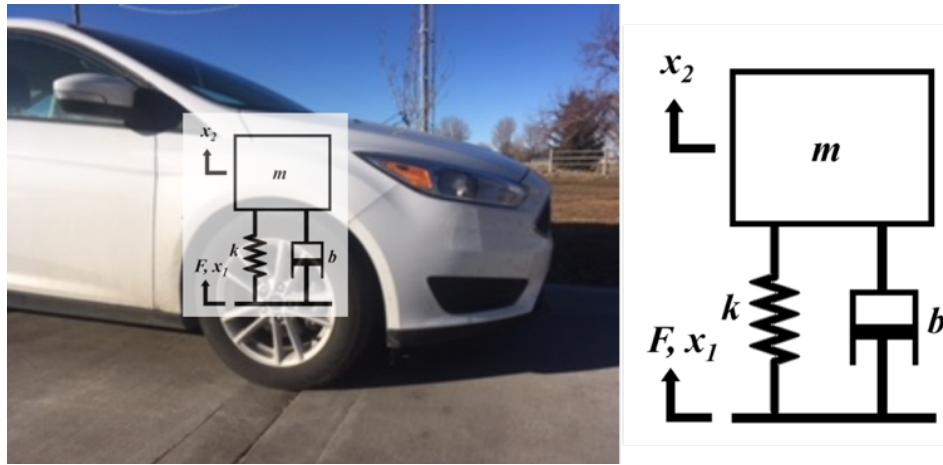


Figure 6.1 Suspension of a car modeled as a spring-mass-damper system.

A car suspension is one of the more obvious applications of the mass-spring-damper-force-velocity components. But these components appear in subtler ways in many applications. For example, these appear in biomechanical engineering as well. When studying how the skin transfers sound vibrations created by the voice-box, the skin stiffness is like a spring and damper system. The mass of the skin is the mass component; the air flow through the larynx produces the force that drives the sound waves. Try to think of how vibrations in a suspension bridge might be modeled with these simple components.

Most mechanical objects are combinations of mechanical elements. For example, the tire shown in Figure 6.1 has mass and inertia, but also dissipates energy to heat like a damper as it flexes and changes shape, and also has stiffness like a spring and will return to its shape. Thus a single object can be mathematically modeled as a combination of three mechanical elements.

6.1.1 Modeling a Mass

Mass elements are usually considered point masses and are assumed to not change shape or dissipate energy but instead store kinetic energy. Because mass has inertia and resists the change in acceleration, the force on a mass is the mass times acceleration:

$$F_m = m\ddot{x}$$

6.1.2 Modeling a Spring

Spring elements are assumed to be massless and store potential energy when they are compressed or stretched. Because springs resist a change in length, the force in a spring is the spring stiffness (k) times the change in length:

$$F_k = k\Delta x = k(x_1 - x_2)$$

If one side of the spring is fixed then the equation becomes:

$$F_k = kx$$

6.1.3 Modeling a Damper

Damper elements dissipate energy to heat and are also assumed to be massless. Dampers resist the change in velocity so the force in a damper is the damping constant (b) times the change in velocity:

$$F_b = b\Delta\dot{x} = b(\dot{x}_1 - \dot{x}_2)$$

In general, the damper force F_b is nonlinear; however, in this class, we will assume that it is proportional to the change in velocity and the proportionality constant is b . If one side of the damper is fixed then the equation becomes:

$$F_b = b\dot{x}$$

6.1.4 d'Alembert's Principle

When modeling electrical circuits, we often use Kirchhoff's voltage or loop law that states that the sum of voltage drops around a loop must sum to zero. When modeling mechanical systems, instead of using Newton's law:

$$\sum F = ma$$

these notes will use [d'Alembert's principle](#) and treat the acceleration term as a force such that the sum of forces will equal zero:

$$\sum F = 0$$

thus the acceleration term can instead be considered the reaction force due to the inertia or mass m .

6.1.5 Modeling a Mass-Damper System

Let's consider a simple mechanical system as shown in Figure 6.2. This is a mass-damper system.

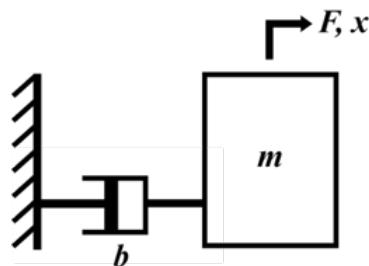


Figure 6.2 A mass-damper system

A boat on water is a mass-damper system. The water friction is the damper, the boat is the mass, the thrust of the propeller is the force, and the speed is the velocity. Another realization is a car. The air drag and friction is the damper, the mass is the car itself, the motor and drivetrain produce the force, and the speed of the car is the velocity.

If we were to draw a free body diagram of the mass, and assume the displacement x to be positive to the right, we can determine the directions of the reaction forces by pretending that we grab the mass and move it in the positive direction, or to the right. If we were to do so, the damper would resist the movement and cause a reaction force to the left; the inertia would also resist the movement and cause a

6.1 Translational Dynamics and Vibrations

reaction force to the left. The applied force is drawn pointing to the right because that was the direction shown in the original image above. Doing this would result in the free body diagram in Figure 6.3

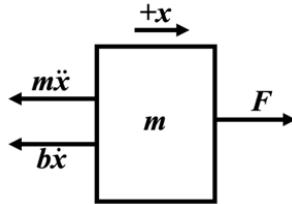


Figure 6.3 Free body diagram of a mass in a typical mass-damper system

A force balance on the mass produces the following equation:

$$F - b\dot{x} - m\ddot{x} = 0$$

and can be rearranged to get:

$$m\ddot{x} + b\dot{x} = F$$

The above equation has a second derivative from the inertial term, so it is second-order in position, but it is actually a first-order ODE in velocity because we can reduce the derivatives to:

$$m\dot{v} + bv = F$$

using the velocity instead of position.

This can further be arranged to the generic first-order form:

$$\tau\dot{x} + x = x_f$$

by dividing by the damping coefficient (b) to get:

$$\frac{m}{b}\dot{v} + v = \frac{F}{b}$$

indicating that the time constant of the mass-damper system is $\tau = \frac{m}{b}$

Knowing the time constant of a mass-damper system, we know that if the input force is a constant then it will take 3τ for the velocity to change 95% from v_0 to $v_f = \frac{F}{b}$. If the input force is sinusoidal then we would also know that the system would behave like a low-pass filter with a cutoff frequency of $\omega_c = \frac{1}{\tau} = \frac{b}{m}$.

6.1.6 Modeling a Spring-Damper System

Now we will model a mechanical system consisting only of a spring and damper, as shown in Figure 6.4.

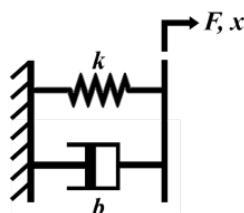


Figure 6.4 A spring-damper system

When drawing free body diagrams, we usually only draw them for masses. Because this system doesn't have a mass we will instead draw a very thin "massless" mass and assume the deflection x is positive to the right. If we were to grab the massless mass and displace it in the positive direction then the spring and damper would both resist the motion and cause a reaction force to the left, as shown in Figure 6.5.

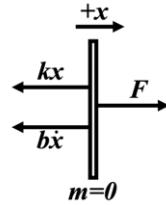


Figure 6.5 Free body diagram of a spring-damper system

A force balance on the spring-damper system results in the following equation:

$$F - kx - b\dot{x} = 0$$

and we can rearrange this equation to get:

$$b\dot{x} + kx = F$$

We can see that this system is a first-order equation. Placing the above equation into the generic first-order form we get:

$$\frac{b}{k}\dot{x} + x = \frac{F}{k}$$

indicating that the time constant of the mass-damper system is $\tau = \frac{k}{b}$.

Knowing the time constant of a spring-damper system, we know that if the input force is a constant then it will take 4τ for the position to change 98% from x_0 to $x_f = \frac{F}{k}$. If the input force is sinusoidal then we would also know that the system would behave like a low-pass filter with a cutoff frequency of $\omega_c = \frac{1}{\tau} = \frac{k}{b}$.

6.1.7 Modeling a Mass-Spring-Damper System

Now we will model a system composed of all three mechanical elements, or a mass-spring-damper system, as shown in Figure 6.6.

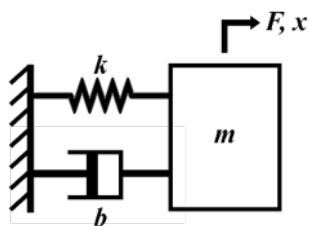


Figure 6.6 A mass-spring-damper system

6.1 Translational Dynamics and Vibrations

The mass-spring-damper system is an important building block for mechanical systems experiencing vibrations. It could represent the suspension system in an automobile, the vibration of human skin as it responds to sound waves, the deflection response of an atomic force microscope, the vibration of a bridge as vehicles drive over it, the swaying of a skyscraper due to wind, the vibrations experienced by a woodpecker's head, and many other applications.

Applying the same approach to drawing the free body diagram, we assume the displacement x is positive to the right and grab the mass and move it in that direction. Doing so stretches the spring and the damper causing reaction forces to the left. The inertia of the mass will also resist the motion and cause a reaction force to the left. The completed free body diagram is shown in Figure 6.7.

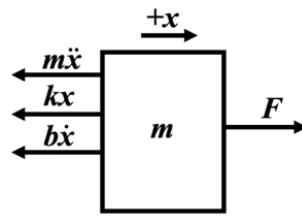


Figure 6.7 A free body diagram of the mass in a mass-spring-damper system

A force balance on the mass-spring-damper system results in the following equation:

$$F - kx - b\dot{x} - m\ddot{x} = 0$$

and we can rearrange this equation to get:

$$m\ddot{x} + b\dot{x} + kx = F$$

Notice that this system is a second-order equation, whereas the mass-damper and spring-damper systems were only first-order equations. The reason this system is second-order is because it has two energy-storing elements in it (the mass stores kinetic energy and the spring stores potential energy). Because dampers dissipate instead of storing energy the mass-damper and spring-damper systems only had one energy-storing element which resulted in first-order ODEs.

To solve the linear ODEs for a mass-spring-damper system, we will set up the problem to be solved using the state-space techniques. First, we need to set up the state-space equations. To do so, we define $x_1 = x$, $x_2 = \dot{x}$, and $u = F$. With these definitions,

$$\begin{bmatrix} \dot{x}_1 \\ \dot{x}_2 \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ -\frac{k}{m} & -\frac{b}{m} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} + \begin{bmatrix} 0 \\ \frac{1}{m} \end{bmatrix} u \quad (6.1)$$

This is in state-space form: $\dot{x} = Ax + Bu$. We learned how to solve these equations for any input u in previous chapters.

Modeling Mass-Spring-Damper Vibrations

Example 6.1.1. Modeling mass-spring-damper vibrations

Consider the mass-spring-damper system shown in Figure 6.6. The mass is $m = 3$ kg, the damping is $b = 2$ N s/m, and the spring stiffness is $k = 13$ N/m. The force F (N) is a chirp signal that sweeps

through a range of frequencies as the time changes from $t = 0$ to 10 s:

$$F = 4 \sin(2\pi(0.1t^2 + 0.001t))$$

Plot the displacement x (m) of the mass as a function of time.

Solution: Using Eq. (6.1) with the values $m = 3$ kg, $b = 2$ N s/m, and $k = 13$ N/m, the state-space equations are

$$\begin{bmatrix} \dot{x}_1 \\ \dot{x}_2 \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ -\frac{13}{3} & -\frac{2}{3} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} + \begin{bmatrix} 0 \\ \frac{1}{3} \end{bmatrix} F$$

$$y = \begin{bmatrix} 1 & 0 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$$

The code below simulates this system. The result is shown in Figure 6.8.

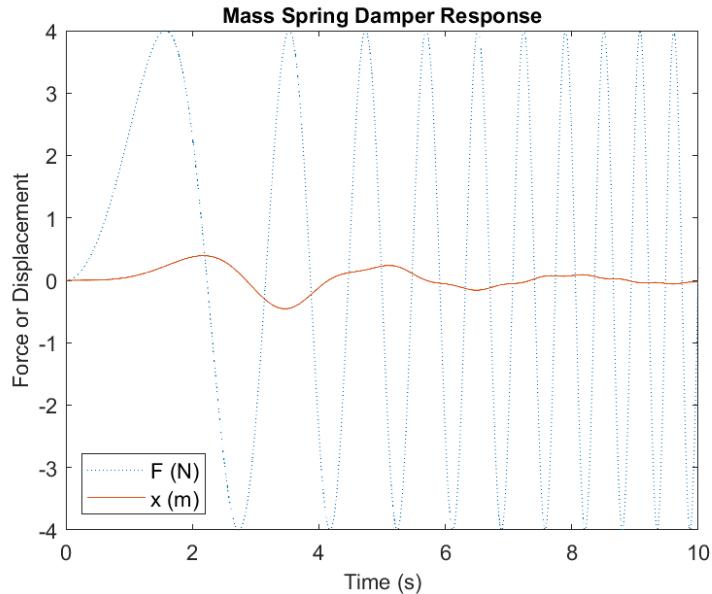


Figure 6.8 Response of the mass-spring-damper system to a chirp force

This is implemented in MATLAB as follows:

```
%Mass Spring Damper parameters
m = 3; %(kg)
b = 2; %(Nm/s)
k = 13; %(N/m)

%Simulation time
dt = 0.001; %(s) simulation timestep
```

6.1 Translational Dynamics and Vibrations

```
t = 0:dt:10; %(s) time vector
N = length(t);

%Chirp force input
F = 4 * sin(2*pi*(0.1*t.^2+0.001*t)); %(N)

%State-space equations
A = [0,1;-k/m,-b/m]; %State matrix
B = [0;1/3]; %Input matrix
Fd = expm([A*dt,B*dt;zeros(1,3)]);
Ad = Fd(1:2,1:2); %State-transition matrix
Bd = Fd(1:2,3); %Input-transition matrix
C = [1,0]; %Output matrix
D = 0; %Feedthrough matrix

%initial conditions
x = [0;0];

%Allocate memory to store the displacement
y = zeros(1,N);

%Run the simulation
for ii = 1:N
    y(ii)=C*x+D*F(ii);
    x = Ad*x+Bd*F(ii);
end

%Plot the results
figure,
plot(t, F, ':', t, y)
legend('F (N)', 'x (m)', 'Location', 'SouthWest')
ylabel('Force or Displacement')
title('Mass Spring Damper Response')
xlabel('Time (s)')
```

6.1.8 Frequency Response of a Mass-Spring-Damper System

You should be able to plot the frequency and time response of a Mass-Spring-Damper system. You should also be able to calculate the damping ratio ζ and natural frequency ω_n .

To derive the frequency response of a mass-spring-damper system, we will begin with the Laplace transfer function from the input force F (or u) to the displacement x . The transfer function is obtained from the state-space equations to be

$$\frac{X}{U} = \frac{1}{ms^2 + bs + k}$$

To determine the frequency response, we replace the Laplace variable s with $j\omega$:

$$\frac{X}{U} = \frac{1}{m(j\omega)^2 + b(j\omega) + k} = \frac{1}{k - mw^2 + j\omega b}$$

and find that

$$r_{TF} = \frac{1}{\sqrt{(k - m\omega^2)^2 + (\omega b)^2}}$$

$$\theta_{TF} = -\text{atan2}(\omega b, k - m\omega^2)$$

We can use these equations to study the frequency response of the mass-spring-damper system. To do so, we will fix the mass $m = 1$ kg and spring constant $k = 1$ N/m, and vary the damping coefficient b . We can use the following MATLAB code to plot the frequency response magnitude on a log-log graph:

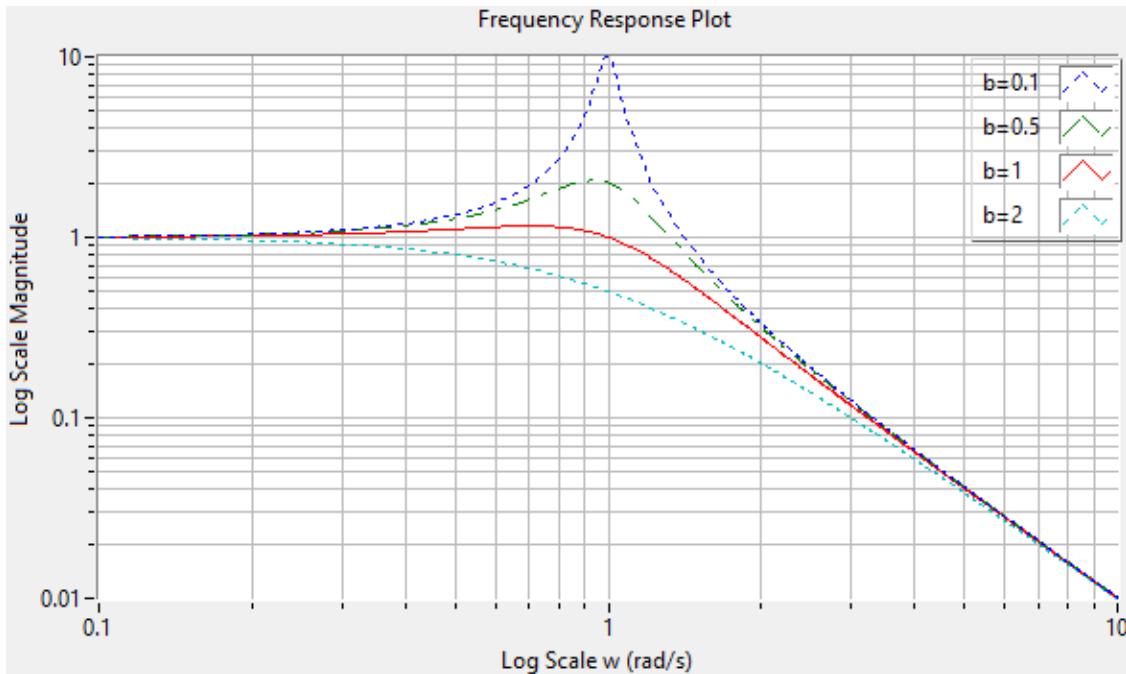


Figure 6.9 Frequency response of a mass-spring-damper system in log-scale

```

close all,
w = 0.1:0.01:10;
m = 1;
k = 1;
r_TF01 = 1./sqrt((k-m*w.^2).^2+(0.1*w).^2);
theta_TF01 = -atan2(0.1*w, k-m*w.^2) *180 / pi;
r_TF05 = 1./sqrt((k-m*w.^2).^2+(0.5*w).^2);
theta_TF05 = -atan2(0.5*w, k-m*w.^2) *180 / pi;
r_TF1 = 1./sqrt((k-m*w.^2).^2+(1*w).^2);
theta_TF1 = -atan2(1*w, k-m*w.^2) *180 / pi;
r_TF2 = 1./sqrt((k-m*w.^2).^2+(2*w).^2);
theta_TF2 = -atan2(2*w, k-m*w.^2) *180 / pi;

figure,
loglog(w, r_TF01,'--',w, r_TF05,'-.',w,r_TF1,'-',w,r_TF2,':')
ylim([0.01,10])

```

6.1 Translational Dynamics and Vibrations

```
title('Frequency Response Plot')
ylabel('Log Scale Magnitude')
grid on
xlabel('Log Scale w (rad/s)')
legend('b=0.1','b=0.5','b=1','b=2')
```

Notice what happens at the 1 rad/s radial frequency as the damping coefficient is decreased to values below $b = 1 \text{ kg/s}$. With less damping, the amplitude increases until it eventually reaches a resonance peak at the resonance frequency of 1 rad/s. This means that it requires less force at a frequency of 1 rad/s to oscillate the mass at a fixed displacement than at any other frequency. The frequency of 1 rad/s is called the **natural frequency** of the system. It is the frequency at which the system would oscillate if it had no damping and had been released from an initially stretched spring displacement. The natural frequency ω_n of a mass-spring-damper system is calculated by

$$\omega_n = \sqrt{\frac{k}{m}}$$

Notice that when $b = 1 \text{ kg/s}$, the amplitude peak is not only lower, but is shifted to a slightly lower frequency than the natural frequency of 1 rad/s. Therefore, damping not only influences the amplitude of the peak, but the frequency at which the peak occurs. The **damped natural frequency** is

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

where ζ is the **damping ratio**, which, for the mass-spring-damper system is

$$\zeta = \frac{b}{2\sqrt{km}}$$

If the damping ratio is $\zeta > 1$, then the mass spring damper system is called **overdamped**. If the ratio is $\zeta = 1$, the system is **critically damped**, and if $\zeta < 1$, the system is **underdamped**.

If the system is underdamped, the system will have a frequency at which it has a peak. This frequency is called the **resonance frequency** ω_R . The resonance frequency occurs at

$$\omega_R = \omega_n \sqrt{1 - 2\zeta^2}$$

Note the differences between the resonance frequency and the damped natural frequency ω_d . The resonance frequency is found by taking the derivative of r_{TF} with respect to ω and solving for the value of ω that causes it to go to zero.

At the resonance frequency, the resonance peak has an amplitude of

$$\text{Peak Magnitude} = \frac{1}{b\omega_n \sqrt{1 - \zeta^2}} = \frac{1}{2m\zeta\omega_n^2 \sqrt{1 - \zeta^2}} = \frac{1}{2k\zeta \sqrt{1 - \zeta^2}}$$

The following graph shows the time response of the system under each of the three damping conditions:

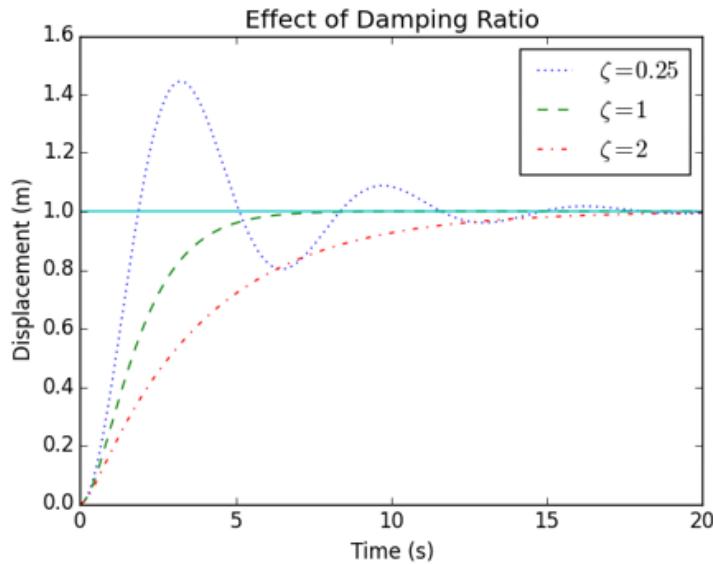


Figure 6.10 Time response of a mass-spring-damper system in overdamped, underdamped, and critically damped scenarios

The underdamped system ($\zeta = 0.25$) overshoots, then oscillates around the final steady-state value. The critically damped system ($\zeta = 1$) rises quickly to the final steady-state value without overshooting or oscillating. Although the overdamped system ($\zeta = 2$) takes a longer time to reach the steady-state value, it also does not overshoot or oscillate. If Figure 6.10 represented a car suspension, which damping ratio would be best? What if it was a door closing system? How about a guitar string? Can you think of systems that fit the three damping conditions?

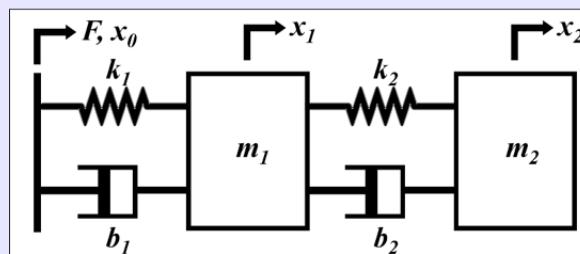
6.1.9 Mass-Spring-Damper Systems with Multiple Masses

Until now, we have modeled systems having only one mass. Modeling systems with multiple masses follows the same process, but we draw a free-body-diagram for each mass in the system. The procedure is illustrated in the following example.

Modeling Systems with Multiple Masses

Example 6.1.2. Modeling dynamic translational systems with multiple masses

Consider the mass-spring-damper system with two masses in Figure 6.11. It is driven by a force F applied at the moving base on the far left of the system. Model this system in state-space.



6.1 Translational Dynamics and Vibrations

Figure 6.11 Mechanical system with two masses, springs, and dampers. The input is the applied force F at the base x_0 .

Solution: In this system, we would need to draw at least two free body diagrams because of the two masses. However, if we would like to model the force input at the base, then we will need to draw a third free body diagram at the base.

Base: Starting at the base, if we draw a massless mass and displace it in the positive x_0 direction we would get the free body diagram in Figure 6.12.

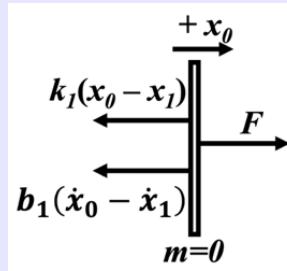


Figure 6.12 Free body diagram for the fixed wall in Figure 6.11

Modeling the free-body diagram results in the equation:

$$F = b_1(\dot{x}_0 - \dot{x}_1) + k_1(x_0 - x_1) = 0 \quad (6.2)$$

which can be arranged to get:

$$\dot{x}_0 = \frac{-k_1}{b_1}x_0 + \frac{k_1}{b_1}x_1 + \dot{x}_1 + \frac{1}{b_1}F \quad (6.3)$$

Mass 1: Now we change our focus to mass m_1 . To draw a free-body-diagram, we will assume a positive x_1 direction to the right, and imagine that we displace it in that direction. Doing so would cause all the springs and dampers to apply a reaction force in the opposite direction, as can be seen in Figure 6.13.

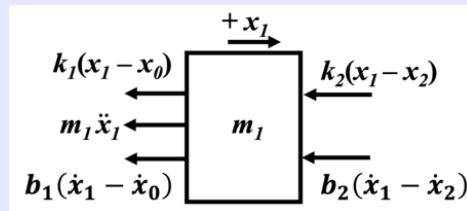


Figure 6.13 Free body diagram for mass 1 in Figure 6.11

Modeling the free-body diagram results in the equation:

$$-m_1\ddot{x}_1 + [b_1(\dot{x}_0 - \dot{x}_1) + k_1(x_0 - x_1)] - b_2(\dot{x}_1 - \dot{x}_2) - k_2(x_1 - x_2) = 0 \quad (6.4)$$

Recognizing that we can replace the term in square brackets with the force F from Eq. (6.2), we can simplify Eq. (6.4) to:

$$\ddot{x}_1 = -\frac{k_2}{m_1}x_1 - \frac{b_2}{m_1}\dot{x}_1 + \frac{k_2}{m_1}x_2 + \frac{b_2}{m_1}\dot{x}_2 + \frac{1}{m_1}F \quad (6.5)$$

The above equation is quite busy, which can make it difficult to determine if it is correct. One method of checking the equation is to use the Sign Consistency Rule. All of the state variables and their derivatives with the same subscript (e.g., x_1 , \dot{x}_1 , \ddot{x}_1) should have the same sign (\pm) if they are placed on the same side of the equation.

Mass 2: Now, looking at mass m_2 , we again assume a positive direction to the right for the displacement x_2 and imagine that we displace it in that direction. This results in the following reaction forces:

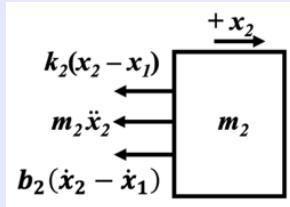


Figure 6.14 Free-body-diagram showing reaction forces for mass 2 in Figure 6.11

From the free-body-diagram, we get the equation:

$$-m_2\ddot{x}_2 - b_2(\dot{x}_2 - \dot{x}_1) - k_2(x_2 - x_1) = 0$$

which simplifies to:

$$\ddot{x}_2 = \frac{k_2}{m_2}x_1 + \frac{b_2}{m_2}\dot{x}_1 - \frac{k_2}{m_2}x_2 - \frac{b_2}{m_2}\dot{x}_2 \quad (6.6)$$

Eq. (6.3), (6.5), and (6.6) provide three state-equations for the five states x_0 , x_1 , \dot{x}_1 , x_2 , and \dot{x}_2 . The input is the force F . Since there are five states, they require five state-equations. Since \dot{x}_1 and \dot{x}_2 are both state-variables and state-derivatives, the additional two equations are trivial:

$$\dot{x}_1 = \dot{x}_1 \quad (6.7)$$

$$\dot{x}_2 = \dot{x}_2 \quad (6.8)$$

Combining these equations with Eq. (6.3), (6.5), and (6.6), the state equations in matrix form are

$$\begin{bmatrix} \dot{x}_0 \\ \dot{x}_1 \\ \ddot{x}_1 \\ \dot{x}_2 \\ \ddot{x}_2 \end{bmatrix} = \begin{bmatrix} \frac{-k_1}{b_1} & \frac{k_1}{b_1} & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & \frac{-k_2}{m_1} & \frac{-b_2}{m_1} & \frac{k_2}{m_1} & \frac{b_2}{m_1} \\ 0 & 0 & 0 & 0 & 1 \\ 0 & \frac{k_2}{m_2} & \frac{b_2}{m_2} & \frac{-k_2}{m_2} & \frac{-b_2}{m_2} \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ \dot{x}_1 \\ x_2 \\ \dot{x}_2 \end{bmatrix} + \begin{bmatrix} \frac{1}{b_1} \\ 0 \\ \frac{1}{m_1} \\ 0 \\ 0 \end{bmatrix} F \quad (6.9)$$

The first row of Eq. (6.9) is Eq. (6.3). The second row is Eq. (6.7). The third row is Eq. (6.5). The fourth row is Eq. (6.8), and the fifth row is Eq. (6.6).

6.2 Rotational Dynamics and Vibrations

6.1.10 Mass-Spring-Damper Systems Modeled as a Longitudinal Wave Propagation

If you have ever played with a Slinky®, you have likely visualized longitudinal wave propagation. The wave propagates from one side of the Slinky to the other by expanding and contracting as shown in Figure 6.15.

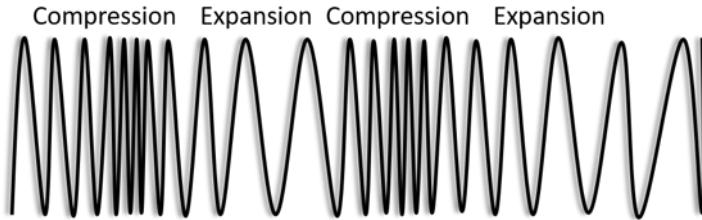


Figure 6.15 Using a slinky to illustrate longitudinal wave propagation.

Waves propagate through other media in a similar way. Sound waves are expansions and contractions in air, water, solids, or other substances. Even waves in the ocean, before the crests begin breaking, can be modeled as the expansion and compression of water.

Mass-spring-damper systems with multiple masses can be used to model longitudinal wave propagation. For example, the Slinky can be thought of as an infinite series of masses, springs, and dampers. For practical purposes, we can model it using a finite series of masses, springs, and dampers as in Figure 6.16. The approximation improves with more mass-spring-damper elements.

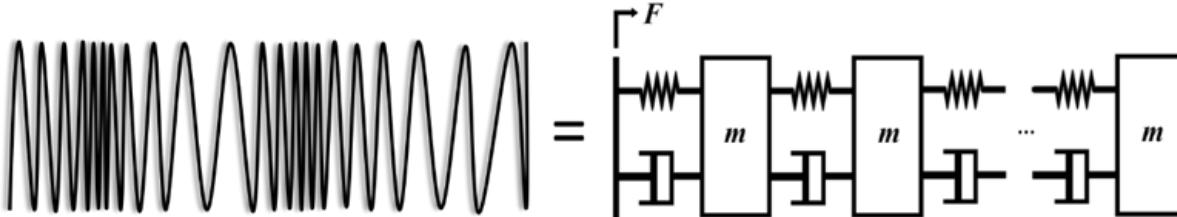


Figure 6.16 Longitudinal wave propagation as an infinite series of masses, springs, and dampers.

We used one-dimensional mass-spring-damper systems to model longitudinal wave propagation. Transverse waves can also be modeled using two-dimensional mass-spring-damper systems.

6.2 Rotational Dynamics and Vibrations

All the translational dynamics principles of Section 6.1 are relevant to rotational dynamics and vibrations. A rotational system is equivalent to a translational system if mass m is replaced by moment of inertia J , translational damping b is replaced by rotational damping b , and translational spring stiffness k is replaced by rotational stiffness k . This is illustrated in the following example, in which a two-inertia system is connected by a shaft. The shaft has torsional stiffness and damping.

Rotational Dynamics

Example 6.2.1. State-space modeling of rotational dynamics

Find the state-space model of the rotational vibrations of the two-inertia system in the following figure:

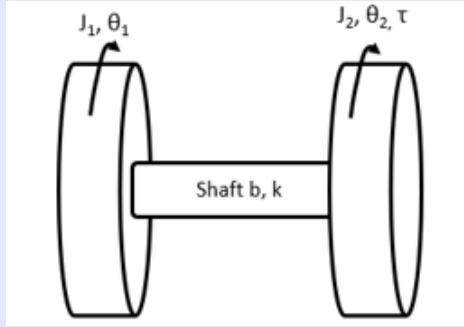


Figure 6.17 A two-inertia rotational system

The two inertias J_1 and J_2 (kg m^2) are connected by a shaft having torsional stiffness k (N m/rad) and damping b (N m s/rad). A positive torque τ (N m) acts on the second inertia.

Solution: Deriving the state-equations begins by applying d'Alembert's principle to the first inertia J_1 . If J_1 is rotated in the positive direction θ_1 , the shaft stiffness and damping would oppose the rotation. The inertia of J_1 would also oppose any accelerations. A free-body-diagram of J_1 is shown below:

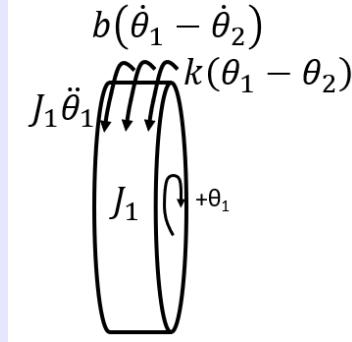


Figure 6.18 Free-body-diagram of inertia J_1

From the free-body-diagram, the torque-balance is

$$-J_1\ddot{\theta}_1 - b(\dot{\theta}_1 - \dot{\theta}_2) - k(\theta_1 - \theta_2) = 0$$

from which we solve for $\ddot{\theta}_1$:

$$\ddot{\theta}_1 = -\frac{k}{J_1}\theta_1 - \frac{b}{J_1}\dot{\theta}_1 + \frac{k}{J_1}\theta_2 + \frac{b}{J_1}\dot{\theta}_2 \quad (6.10)$$

Next, we create a free-body-diagram of the second inertia J_2 :

6.3 Modeling Dynamic Electrical Circuits

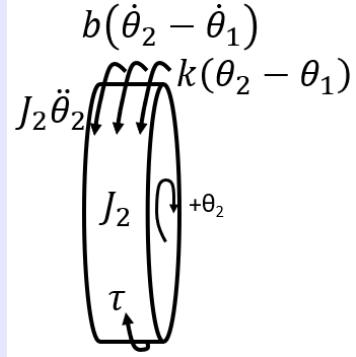


Figure 6.19 Free-body-diagram of inertia J_2

From the free-body-diagram, the torque-balance is

$$-J_2\ddot{\theta}_2 - b(\dot{\theta}_2 - \dot{\theta}_1) - k(\theta_2 - \theta_1) + \tau = 0$$

from which we solve for $\ddot{\theta}_2$:

$$\ddot{\theta}_2 = \frac{k}{J_2}\theta_1 + \frac{b}{J_2}\dot{\theta}_1 - \frac{k}{J_2}\theta_2 - \frac{b}{J_2}\dot{\theta}_2 + \frac{1}{J_2}\tau \quad (6.11)$$

There are four state-variables $\theta_1, \dot{\theta}_1, \theta_2, \dot{\theta}_2$, and one input torque τ . Eqs. (6.10) and (6.11) provide two of the required four state-equations. Because $\dot{\theta}_1$ and $\dot{\theta}_2$ are both state derivatives and variables, the other two equations are trivial:

$$\dot{\theta}_1 = \dot{\theta}_1 \quad (6.12)$$

$$\dot{\theta}_2 = \dot{\theta}_2 \quad (6.13)$$

Combining Eqs. (6.12) and (6.13) with Eqs. (6.10) and (6.11) into matrix form results in the desired state-equation:

$$\begin{bmatrix} \dot{\theta}_1 \\ \ddot{\theta}_1 \\ \dot{\theta}_2 \\ \ddot{\theta}_2 \end{bmatrix} = \begin{bmatrix} 0 & 1 & 0 & 0 \\ \frac{-k}{J_1} & \frac{-b}{J_1} & \frac{k}{J_1} & \frac{b}{J_1} \\ 0 & 0 & 0 & 1 \\ \frac{k}{J_2} & \frac{b}{J_2} & \frac{-k}{J_2} & \frac{-b}{J_2} \end{bmatrix} \begin{bmatrix} \theta_1 \\ \dot{\theta}_1 \\ \theta_2 \\ \dot{\theta}_2 \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ 0 \\ \frac{1}{J_2} \end{bmatrix} \tau \quad (6.14)$$

The first row of Eq. (6.14) is Eq. (6.12). The second row is Eq. (6.10). The third row is Eq. (6.13), and the fourth row is Eq. (6.11).

6.3 Modeling Dynamic Electrical Circuits

Electrical circuits with energy storage components, such as batteries, capacitors, and inductors, are dynamic systems. The voltage, current, energy, and power in the circuit change over time. This section

creates mathematical models of dynamic electrical circuits.

6.3.1 Basics of Capacitors

Capacitors are basically two conductive surfaces separated by an insulating material such as air or ceramic. When a capacitor is subjected to a DC voltage source, electric charges are attracted to opposite sides of the capacitor and an electric field is formed as can be seen in Figure 6.20.

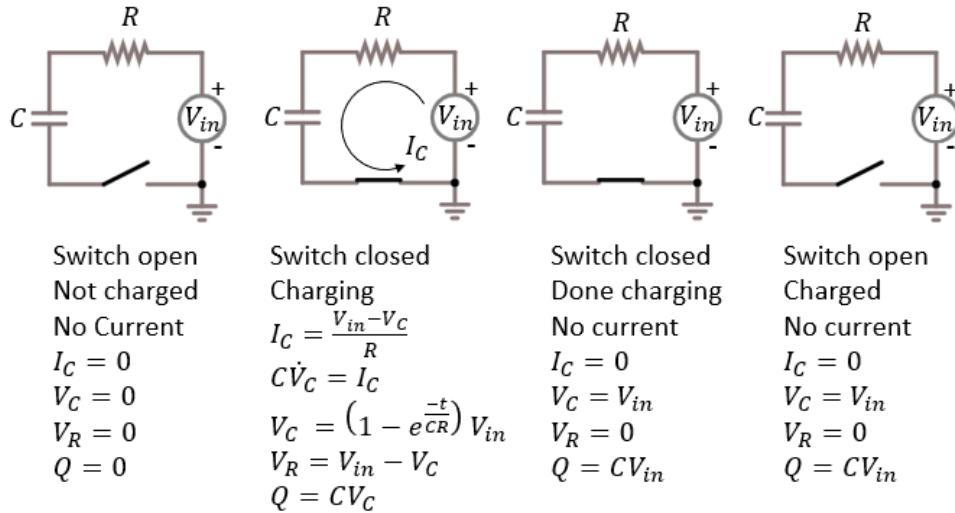


Figure 6.20 Capacitor charged states and equations

If one coulomb of charge is stored on the plates of a capacitor, and the voltage across the plates is one volt, then the capacitor is said to have a capacitance of one farad, thus a capacitor has units of farads (F), where one farad is equal to one coulomb of charge per volt, or

$$1 F = \frac{1 C}{1 V}$$

where C is charge, or 1 coulomb. Another useful unit conversion is $1 F = 1 s/\Omega$. The symbol Ω (ohm) is used for resistance. A capacitor is characterized by a symbol C, thus:

$$C = \frac{Q}{V}$$

where C is the capacitance, Q is the charge stored on the plate of the capacitor, and V is the voltage across the plates. (I know, it's confusing having multiple Cs). Because one farad is actually a very large capacitance, we typically use capacitors in the range of micro- (μF), nano- (nF), and pico-farads (pF). Capacitors in circuit diagrams are symbolized by two parallel plates as can be seen in Figure 6.21.

As shown in Figure 6.21, some capacitors have a plus sign by them in the circuit diagram. This is to show that the capacitor is polarized and is made of a material that requires one side to be connected to positive voltage. If a polarized capacitor is accidentally reversed, then it could cause the material in the capacitor to break down and even explode. Typically, polarized capacitors have a dark casing with one white strip on the leg or terminal that indicates the negative terminal of the capacitor.

Capacitors have a large variety of uses. Some are listed below:

6.3 Modeling Dynamic Electrical Circuits

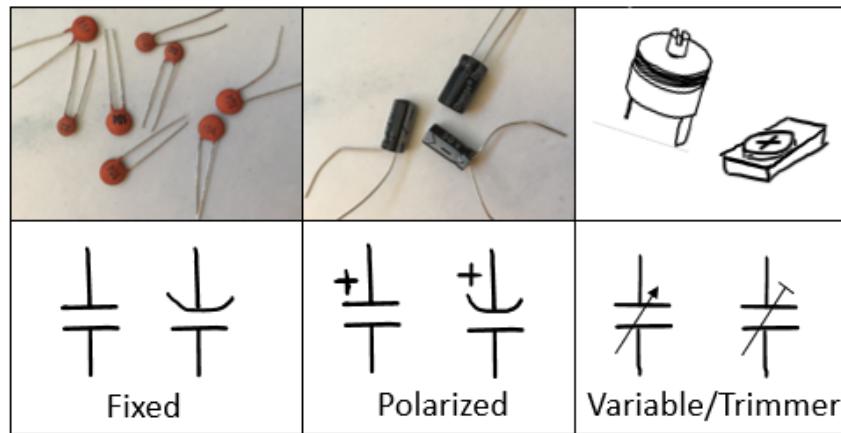


Figure 6.21 Capacitors and symbols

Energy storage:

- Capacitors can be used to replace batteries in automobiles and other applications. Supercapacitors can handle large charging and discharging rates so they can be used to start vehicles or charge quickly at a charging station for electric vehicles.
- Because supercapacitors can handle a large discharge of current, they are used in AED devices when an electric shock is needed for someone in cardiac arrest.

DC blocking:

- Because current will only pass through a capacitor if a voltage is changing, a capacitor can be used to block DC current in a circuit, or remove DC current from a sinusoidal voltage with a DC offset.

Filtering:

- Capacitors are used to filter high or low frequencies from AC waveforms, for example in audio applications where the high frequency audio is filtered out of a signal before being sent to a subwoofer used to play the bass.

Smoothing:

- Capacitors can be used to remove the ripple, or smooth the fluctuating voltage from a signal. This is useful when AC voltage is rectified to DC voltage. This can be seen in Figure 6.22.

Sensors:

- When the geometry of a capacitor changes, its capacitance changes. Capacitance also changes when the dielectric or material near the capacitor changes. Thus, capacitors can be used as sensors. For example, stud finders detect the change in capacitance near a stud in the wall.

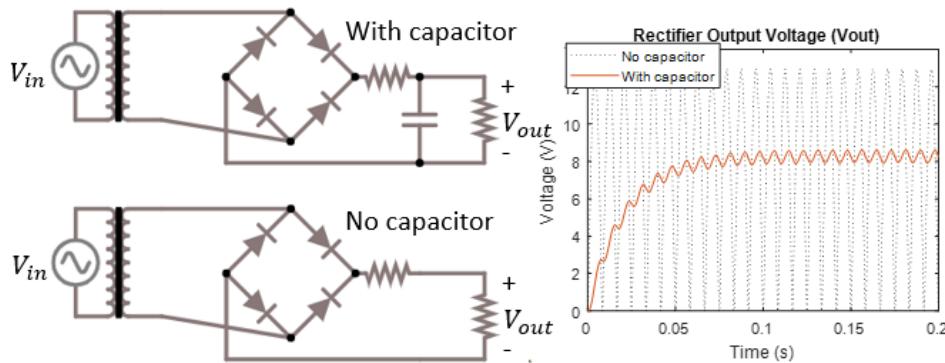


Figure 6.22 Capacitors are used in AC rectifier circuits

6.3.2 Basics of Inductors

Inductors are another type of energy storage component in electrical circuits. When an electron moves, it generates a magnetic field around it. When a lot of electrons are moving in a larger current, the strength of the magnetic field increases. The magnetic field around current-carrying wire flows in a fashion that follows a right-hand rule, as shown in Figure 6.23, where if the right thumb is pointed in the direction of the current then right-hand fingers will curl in the direction of magnetic flux around the wire.

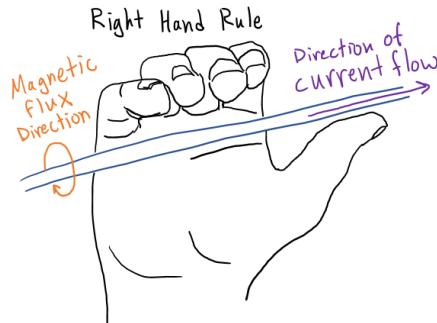


Figure 6.23 The right-hand rule relates current flow and magnetic field directions

If the wire is bent into a loop then the fields merge in the middle and flow in the same direction as can be seen in Figure 6.24. If the wire is wrapped several more times, then the strength of the magnetic field is increased, as shown in Figure 6.25. The magnetic field now acts just like a permanent magnet, but because its magnetic field is induced by the flow of electricity it is called an electromagnet.

6.3.3 Inductors store energy in the form of a magnetic field

An inductor is basically an electromagnet where wires are coiled together to make a stronger magnetic field. As long as current is flowing then the magnetic field will be present around the wires. The magnetic field takes energy to form and gets its energy from the flow of current. Because energy is drawn from the current to make the magnetic field, an inductor acts like mechanical mass or inertia in the sense that it takes time to get the current to flow through it, just like it takes time to build up the speed of a mass.

6.3 Modeling Dynamic Electrical Circuits

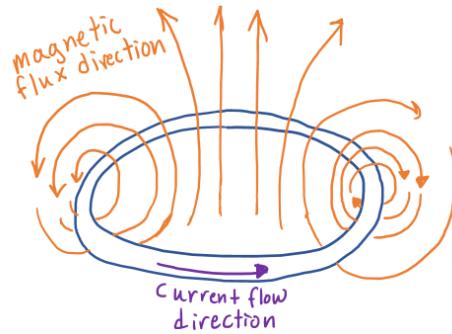


Figure 6.24 The shape of a magnetic field around a loop of wire.

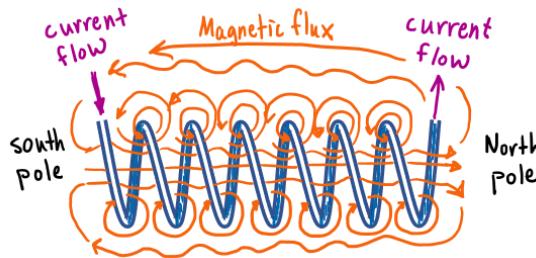


Figure 6.25 An inductor and electromagnet

The energy built up in the magnetic field can also drive a flow of current as the magnetic field collapses. For example, if the current through the inductor is shut off then as the field collapses it will continue to push current through the wire. Thus, an inductor acts like inertia where it tries to keep current moving at the same speed, preventing the current from instantly speeding up or slowing down.

6.3.4 Units of inductance and symbols for an inductor

The unit of measure for an inductor is the henry (H). One henry (H) is equivalent to one volt being induced by a change in current of one ampere (A) in one second (s):

$$1 \text{ H} = \frac{1 \text{ V}}{1 \text{ A/s}}$$

Another useful unit conversion is $1 \text{ H} = 1 \Omega \text{ s}$.

An inductor is characterized in equations by the capital letter L . Inductors in circuits are symbolized by a coil of wire. Inductors with an iron core have parallel lines next to the symbol of a coil of wire. These can be seen in Figure 6.26.

Inductors have a variety of uses. Some are listed below.

Filtering:

- Just like capacitors, inductors can be used to filter out high or low frequencies from AC waveforms.

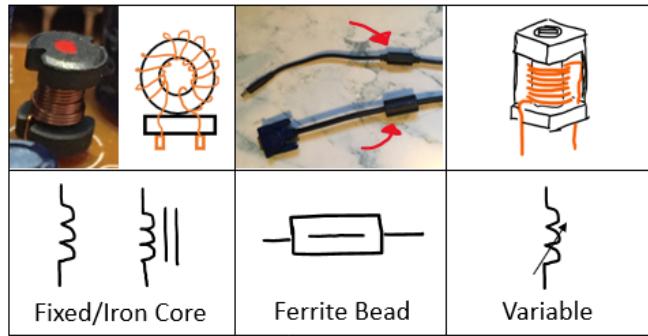


Figure 6.26 Various inductors and symbols



Figure 6.27 Inductors are used to prevent damage from electrical surges

Smoothing:

- Inductors can be used to smooth the ripple in a voltage signal or power supply. For example, most laptop power cords have an inductor placed in the cord to help smooth out the ripple in current from the adapter that converts the AC to DC power. Such as in Figure 6.27.

Sensors:

- Inductors can be used as sensors, to detect when a change in inductance occurs by changing the geometry of the inductor, or the permeability of the space around the inductor. For example, induction sensors are used to detect when a car is present at a stoplight, or gate.

Transformers:

- As described above, transformers use two inductors of different inductance in order to exchange between current and voltage in AC power.

Power transfer:

- Similar to a transformer, inductors can be used to transfer power between devices, for example:
 - Electric toothbrush charger, which can charge a toothbrush through the sealed plastic housing.

6.4 Modeling Electrical Circuits with $\dot{x} = Ax + Bu$

- Cell phone chargers, which can charge your phone while resting on it.
- RFID tags, which do not have their own power, are powered when it comes near an RFID reader. These can be put in devices, even pets, to track or identify them.
- Ignition coils in automobiles use inductors to jump up the 12 volts from a battery to much larger voltages (like 50,000 V) to spark a spark plug.
- Inductive stove tops use induction to induce a current in a pot or pan to get it hot, instead of using flames or heating elements which get other objects hot.

6.4 Modeling Electrical Circuits with $\dot{x} = Ax + Bu$

This section models electrical circuits with energy storage components such as capacitors and inductors. Table 6.1 summarizes various common circuit elements. It lists the element symbol, the time-domain equation, the Laplace-domain equation, and the series and parallel equivalences.

Table 6.1 Common Circuit Elements

Element		Time Eqn	Laplace	Series	Parallel
Resistor	R	$V = IR$	$V = IR$	$R_1 + R_2 + \dots$	$\frac{1}{R_1 + \frac{1}{R_2} + \dots}$
Capacitor	C	$C \frac{dV}{dt} = I$	$V = \frac{1}{sC} I$	$\frac{1}{C_1 + \frac{1}{C_2} + \dots}$	$C_1 + C_2 + \dots$
Inductor	L	$L \frac{dI}{dt} = V$	$V = sLI$	$L_1 + L_2 + \dots$	$\frac{1}{L_1 + \frac{1}{L_2} + \dots}$
Voltage	V	V	V	$V_1 + V_2 + \dots$	$\max(V_1, V_2, \dots)$
Current	I	I	I	$\max(I_1, I_2, \dots)$	$I_1 + I_2 + \dots$

6.4.1 Modeling Capacitors

Flowing electrons leave one plate of a capacitor and build up on the opposite plate. This causes a difference in electrical charge across the capacitor. The difference in the electrical charge Q (in coulombs) that builds up between the plates is proportional to the voltage drop across the plate:

$$Q = CV_C$$

where V_C is the voltage drop across the capacitor and C is the capacitance. The electrical current in the capacitor circuit is equal to the change in the charge of the capacitor over time:

$$I = \frac{dQ}{dt}$$

Combining the last two equations produces a relationship between the electrical current and the capacitor voltage drop:

$$I = C \frac{dV_C}{dt}$$

In the Laplace domain, this equation is

$$I(s) = sCV_C(s)$$

or

$$V_C(s) = I(s) \frac{1}{sC}$$

where $I(s)$ and $V_C(s)$ are the Laplace transforms of I and V_C respectively, and s is the Laplace variable. This equation looks a lot like ohm's law for a resistor, $V_R = IR$, except R is replaced by $\frac{1}{sC}$. Because it is in the Laplace domain instead of the time-domain, the term $\frac{1}{sC}$ is called "impedance" (Z_C) instead of "resistance" (R). The impedance of a capacitor is therefore

$$Z_C = \frac{1}{sC}$$

Impedance has the same units as resistance: Ω . The Laplace variable s has units of rad/s and C has units of farads (s/Ω).

6.4.2 Modeling Inductors

The voltage drop V_L across an inductor is proportional to the change in current over time:

$$V_L = L \frac{dI}{dt}$$

As a result, if current flows steadily through the inductor, there is no voltage drop across it. The constant L is the inductance. It has units of Henries ($\Omega \cdot s$). The equation in Laplace form looks a lot like Ohm's law

$$V_L(s) = sLI(s)$$

where

$$Z_L = sL$$

is the impedance of the inductor and has units of ohms.

6.4.3 Modeling Resistors

Ohm's law for a resistor is the same in the time and Laplace domains:

$$V_R(s) = RI(s)$$

Since the equation is in the Laplace domain, the impedance is the resistance

$$Z_R = R$$

and has units of ohms.

6.4.4 State-Space Modeling of Electrical Circuits

It may be important to ensure that the physical meaning of the state variables in the state-space equations $\dot{x} = Ax + Bu$ are known. This is possible by using a time-domain analysis. The Laplace-domain analyses explained in Section 6.4.5 cannot always ensure that the state variables have a known physical meaning. To ensure that the states have physical meaning, we can define the state variables to be the voltage drop across each capacitor and the current in each inductor. This approach is demonstrated in the following example. Another viable approach not discussed in this book is to use bond-graphs or linear graphs.

6.4 Modeling Electrical Circuits with $\dot{x} = Ax + Bu$

State-Space Modeling of an Electrical Circuit

Example 6.4.1. Modeling an RLC (Resistor-Inductor-Capacitor) circuit with state-space

Find the current through the resistor and the voltage of Node A in the circuit of Figure 6.28 with $R = 1 \text{ k}\Omega$, $L = 0.5 \text{ H}$, and $C = 5 \text{ mF}$. The input voltage is $V_{in} = 5 \text{ V}$ and the initial conditions for the inductor and capacitor are both zero. Select state variables that are physically meaningful, specifically, the current through the inductor and the voltage drop across the capacitor.

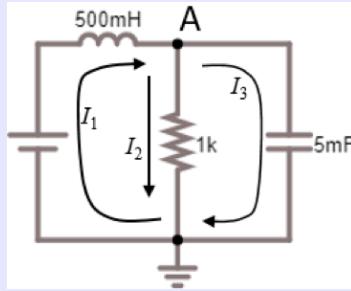


Figure 6.28 A Resistor-Inductor-Capacitor (RLC) circuit

Solution: To ensure that the state variables are physically meaningful, we will choose the states to be the current through the inductor and the voltage drop across the capacitor. To do so, we note that

$$\dot{I}_1 = \frac{1}{L} V_L$$

where the voltage drop V_L across the inductor is

$$V_L = V_{in} - V_A$$

Since $I_L = I_1$ is the current through the inductor and $V_C = V_A$ is the voltage drop across the capacitor, we can combine the two equations above into one state equation:

$$\dot{I}_L = \frac{1}{L} (V_{in} - V_C)$$

We also note that by the capacitor equation

$$\dot{V}_C = \frac{1}{C} I_3$$

where by Kirchhoff's current law

$$I_3 = I_1 - I_2 = I_L - \frac{V_A - 0}{R} = I_L - \frac{1}{R} V_C$$

We can combine the two equations above into a single state equation.

$$\dot{V}_C = \frac{1}{C} \left(I_L - \frac{1}{R} V_C \right)$$

Finally, we can combine the two state equations into a state-space form:

$$\begin{bmatrix} \dot{I}_L \\ \dot{V}_C \end{bmatrix} = \begin{bmatrix} 0 & -\frac{1}{L} \\ \frac{1}{C} & -\frac{1}{CR} \end{bmatrix} \begin{bmatrix} I_L \\ V_C \end{bmatrix} + \begin{bmatrix} \frac{1}{L} \\ 0 \end{bmatrix} V_{in}$$

In this form, the states have the desired physical meaning. The first state is the current in the inductor. The second state is the voltage drop across the capacitor.

The output equation is

$$\begin{bmatrix} V_A \\ I_R \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 0 & \frac{1}{R} \end{bmatrix} \begin{bmatrix} I_L \\ V_C \end{bmatrix}$$

```
C = 0.005; %Farads
L = 0.5; %Henry
R = 1000; %Ohms
Vin = 5; %V
dt = 0.001;
A = [0,-1/L;1/C,-1/(C*R)];
B = [1/L;0];
CV = [0,1];
CI = [0,1/R];
Ad = expm(A*dt);
Bd = A^(-1)*(Ad-eye(2))*B;
t = [0:dt:20];
N = length(t);
V_A = zeros(1,N);
I_R = zeros(1,N);
x = [0;0];
for ii = 1:N
    V_A(ii) = CV*x;
    I_R(ii) = CI*x;
    x = Ad*x+Bd*Vin;
end
figure,
subplot(211)
plot(t,V_A,'k');
ylabel('V_A (V)');
grid on
subplot(212)
plot(t,I_R,'k');
ylabel('I_R (A)');
xlabel('Time (s)');
grid on
```

Compare the resulting plots of 6.29 with Figures 6.30 and 6.32 above.

6.4 Modeling Electrical Circuits with $\dot{x} = Ax + Bu$

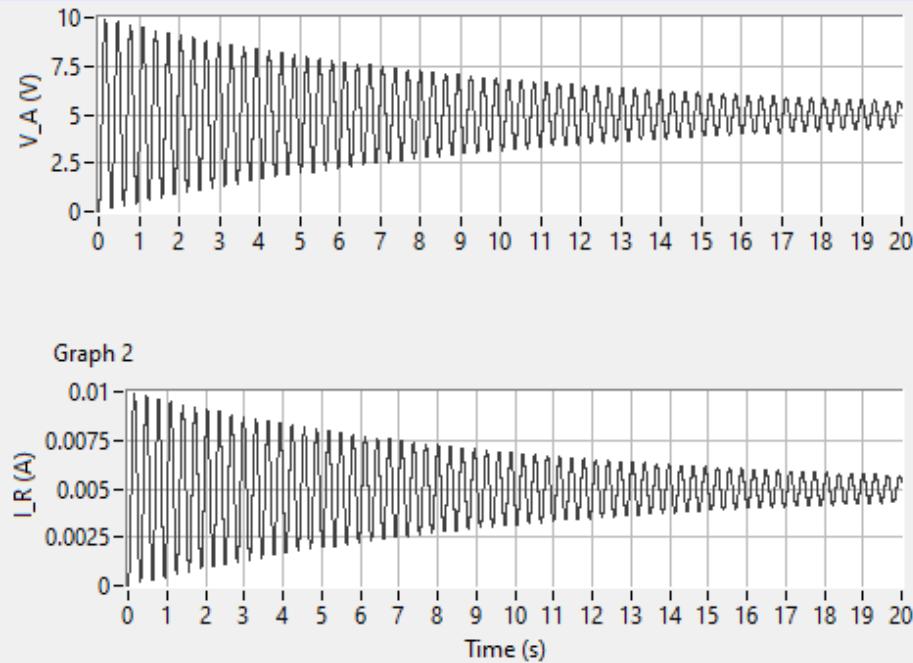


Figure 6.29 RLC circuit response

6.4.5 Modeling Electrical Circuits in the Laplace Domain

Evaluating circuits with Laplace impedances in the Laplace domain is equivalent to evaluating circuits with resistors in the time domain. We can use the same sets of tools that we used for resistor circuits: mesh analysis, nodal analysis, Kirchhoff's voltage and current laws, ohm's law. The result, however, for circuits with Laplace impedances is an ODE.

Warning! When evaluating circuits in the Laplace-domain, the physical meaning of the state variables is lost. Therefore, if initial conditions are known and the transient response is important, you should use the time-domain analysis described earlier. If a circuit has switches or diodes, you should always use the time-domain analysis, because the effects of transients after switching are important. You should never use the Laplace domain analyses in these cases.

The Laplace domain analysis is demonstrated in the following two examples:

Laplace Domain Analysis of an RLC Circuit

Example 6.4.2. Nodal analysis of an RLC circuit in the Laplace domain

Use nodal analysis to find the voltage at the node A in the circuit in Figure 6.28 with $R = 1\text{k}\Omega$, $L = 0.5 \text{ H}$, and $C = 5 \text{ mF}$. The input voltage is $V_{in} = 5 \text{ V}$ and the initial conditions for the inductor and capacitor are both zero.

Solution: The circuit has two nodes, Node A and the ground node. We can use Kirchhoff's current

law at either one of these nodes to find that

$$I_1 = I_2 + I_3$$

We use ohm's law to replace each of the three currents with the voltage drop across the impedance element divided by its impedance:

$$\frac{V_{in} - V_A}{Z_L} = \frac{V_A - 0}{Z_R} + \frac{V_A - 0}{Z_C}$$

Plugging in the impedances $Z_C = \frac{1}{sC}$, $Z_L = sL$, and $Z_R = R$, we get

$$\frac{V_{in} - V_A}{sL} = \frac{V_A}{R} + sCV_A$$

Now we rearrange the equation to solve for the transfer function $\frac{V_A}{V_{in}}$

$$\begin{aligned} \frac{V_A}{V_{in}} &= \frac{R}{s^2CLR + sL + R} \\ &= \frac{\frac{1}{CL}}{s^2 + s\frac{1}{CR} + \frac{1}{CL}} \end{aligned}$$

We can take the inverse Laplace transform to convert this equation to an ODE

$$\ddot{V}_A = -\frac{1}{CR}\dot{V}_A - \frac{1}{CL}V_A + \frac{1}{CL}V_{in}$$

If we chose the states to be

$$x = \begin{bmatrix} V_A \\ \dot{V}_A \end{bmatrix}$$

Then the state-space equations are

$$\begin{bmatrix} \dot{V}_A \\ \ddot{V}_A \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ -\frac{1}{CL} & -\frac{1}{CR} \end{bmatrix} \begin{bmatrix} V_A \\ \dot{V}_A \end{bmatrix} + \begin{bmatrix} 0 \\ \frac{1}{CL} \end{bmatrix} V_{in}$$

Both initial conditions are zero. If we plug in values for V_{in} , C , L , and R , we get

$$\begin{bmatrix} \dot{V}_A \\ \ddot{V}_A \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ -400 & -0.2 \end{bmatrix} \begin{bmatrix} V_A \\ \dot{V}_A \end{bmatrix} + \begin{bmatrix} 0 \\ 400 \end{bmatrix} 5$$

The output is V_A , so the output equation ($y = Cx + Du$) is rather trivial:

$$V_A = \begin{bmatrix} 1 & 0 \end{bmatrix} \begin{bmatrix} V_A \\ \dot{V}_A \end{bmatrix} + 0V_{in}$$

6.4 Modeling Electrical Circuits with $\dot{x} = Ax + Bu$

The discrete-time solution using a time-step of $\Delta t = 0.001\text{s}$ is

$$\begin{bmatrix} V_{A,k+1} \\ \dot{V}_{A,k+1} \end{bmatrix} = \begin{bmatrix} 0.9998 & 0.001 \\ -0.3999 & 0.9996 \end{bmatrix} \begin{bmatrix} V_{A,k} \\ \dot{V}_{A,k} \end{bmatrix} + \begin{bmatrix} 0.0002 \\ 0.3999 \end{bmatrix} \quad 5$$

$$V_{A,k} = \begin{bmatrix} 1 & 0 \end{bmatrix} \begin{bmatrix} V_{A,k} \\ \dot{V}_{A,k} \end{bmatrix} + 0V_{in,k}$$

Simulating the circuit for 20 seconds results in the plot shown in Figure 6.30.

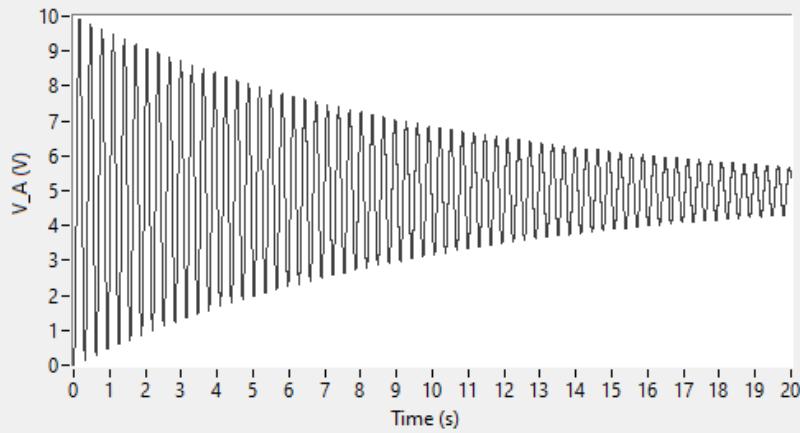


Figure 6.30 RLC circuit response

The circuit is simulated with the following MATLAB code.

```
C = 0.005; %Farads
L = 0.5; %Henry
R = 1000; %Ohms
Vin = 5; %V
dt = 0.001;
A = [0,1;-1/(C*L),-1/(C*R)];
B = [0;1/(C*L)];
C = [1,0];
D = 0;
Ad = expm(A*dt);
Bd = A^(-1)*(Ad-eye(2))*B;
t = [0:dt:20];
N = length(t);
V_A = zeros(1,N);
x = [0;0];
for ii = 1:N
    V_A(ii) = C*x;
    x = Ad*x+Bd*Vin;
end
plot(t,V_A,'k');
xlabel('Time (s)');
ylabel('V_A (V)');
```

The following example solves the same circuit but uses mesh instead of nodal analysis.

Laplace Domain Mesh Analysis of an RLC Circuit

Example 6.4.3. Mesh analysis of an RLC circuit in the Laplace domain

Use mesh analysis to find the current through the resistor in the following circuit with $R = 1 \text{ k}\Omega$, $L = 0.5 \text{ H}$, and $C = 5 \text{ mF}$. The input voltage is $V_{in} = 5 \text{ V}$ and the initial conditions for the inductor and capacitor are both zero.

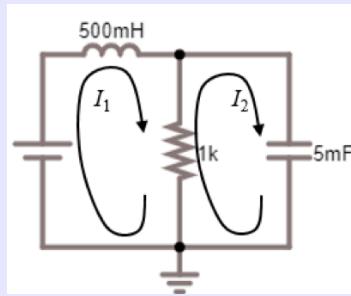


Figure 6.31 An RLC circuit with two meshes labeled

Solution: Using the two mesh currents in the Laplace domain with Kirchhoff's voltage law results in the following set of equations:

$$\begin{aligned} V_{in} - I_1 Z_L - Z_R(I_1 - I_2) &= 0 \\ -Z_R(I_2 - I_1) - I_2 Z_C &= 0 \end{aligned}$$

where the impedances are $Z_C = \frac{1}{sC}$, $Z_L = sL$, and $Z_R = R$. If we plug in these values, we can solve for the currents to get

$$\begin{aligned} I_1 &= \frac{sCR + 1}{s^2 LRC + sL + R} V_{in} \\ I_2 &= \frac{sCR}{s^2 LRC + sL + R} V_{in} \end{aligned}$$

The current through the resistor (from top to bottom) is the difference between the two currents I_1 and I_2 :

$$I_R = I_1 - I_2$$

or:

$$\begin{aligned} I_R &= \frac{sCR + 1}{s^2 LRC + sL + R} V_{in} - \frac{sCR}{s^2 LRC + sL + R} V_{in} \\ &= \frac{1}{s^2 LRC + sL + R} V_{in} \end{aligned}$$

We can apply the inverse Laplace to get the differential equation:

$$LRC\ddot{I}_R + L\dot{I}_R + RI_R = V_{in}$$

6.4 Modeling Electrical Circuits with $\dot{x} = Ax + Bu$

If we choose the state vector to be

$$x = \begin{bmatrix} I_R \\ \dot{I}_R \end{bmatrix}$$

Then the state-space and output equations are

$$\begin{bmatrix} \dot{I}_R \\ \ddot{I}_R \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ -\frac{1}{CL} & -\frac{1}{RC} \end{bmatrix} \begin{bmatrix} I_R \\ \dot{I}_R \end{bmatrix} + \begin{bmatrix} 0 \\ \frac{1}{LCR} \end{bmatrix} V_{in}$$

$$I_R = \begin{bmatrix} 1 & 0 \end{bmatrix} \begin{bmatrix} I_R \\ \dot{I}_R \end{bmatrix} + [0] V_{in}$$

We can use MathScript to calculate the discrete-time solution and plot the response:

```
C = 0.005; %Farads
L = 0.5; %Henry
R = 1000; %Ohms
Vin = 5; %V
dt = 0.001;
A = [0,1;-1/(C*L),-1/(C*R)];
B = [0;1/(C*L*R)];
C = [1,0];
D = 0;
Ad = expm(A*dt);
Bd = A^(-1)*(Ad-eye(2))*B;
t = [0:dt:20];
N = length(t);
I_R = zeros(1,N);
x = [0;0];
for ii = 1:N
    I_R(ii) = C*x;
    x = Ad*x+Bd*Vin;
end
plot(t,I_R,'k');
xlabel('Time (s)');
ylabel('I_R (A)');
```

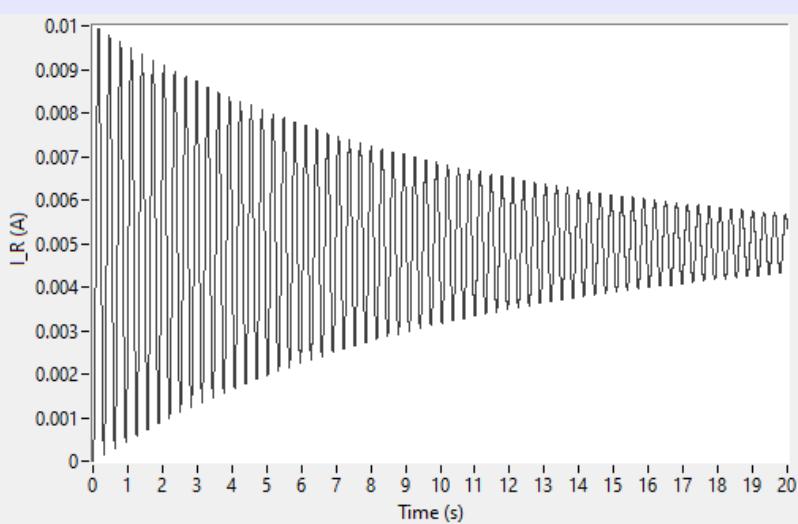


Figure 6.32 RLC circuit response

6.4.6 Equivalent Impedance of a Circuit

Finding the equivalent impedance of a circuit with resistors, capacitors, and inductors in the Laplace domain is exactly like solving for the equivalent resistance of a circuit with resistors in the time domain. This is demonstrated in the following example.

Finding Equivalent Impedance

Example 6.4.4. Finding equivalent impedance

Find the equivalent impedance of the circuit in Figure 6.33 with $R = 1\text{k}\Omega$, $L=0.5 \text{ H}$, and $C=5\text{mF}$.

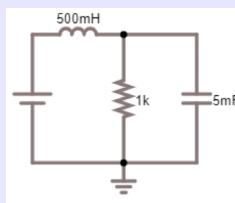


Figure 6.33 An RLC circuit

Solution: The resistor and capacitor are in parallel. Like parallel resistors, their impedances add as follows:

$$Z_{RC} = \frac{1}{\frac{1}{Z_C} + \frac{1}{Z_R}}$$

where Z_{RC} is the equivalent impedance of the parallel RC branch of the circuit, $Z_C = 1/sC$ is the impedance of the capacitor, and $Z_R = R$ is the impedance of the resistor. The inductor is in series

6.4 Modeling Electrical Circuits with $\dot{x} = Ax + Bu$

with the RC parallel circuit. Like series resistors, their impedances add as follows:

$$\begin{aligned}Z_{eq} &= Z_L + Z_{RC} \\&= Z_L + \frac{1}{\frac{1}{Z_C} + \frac{1}{Z_R}}\end{aligned}$$

where Z_{eq} is the equivalent impedance of the circuit and $Z_L = sL$ is the impedance of the inductor. Putting in Laplace impedances for Z_L , Z_C , and Z_R produces

$$Z_{eq} = sL + \frac{1}{sC + \frac{1}{R}} = sL + \frac{R}{sCR + 1}$$

With values, the equivalent impedance is

$$Z_{eq} = 0.5s + \frac{1000}{5s + 1}$$

and has units of Ω .

6.4.7 Electrical Circuits with Switches and Diodes

Mathematical models of electrical circuits can become more complicated when the circuits have switches and / or diodes. One reason is because the switch or diode can open a branch in the circuit, entirely changing the possible paths for current to flow. Modeling circuits with switches or diodes often requires removing or adding branches to the circuit, completely changing it. The mathematical models need to correctly account for this change while simultaneously retaining the physical meaning of each state variable and being consistent with positive sign notation.

6.4.8 Switches

For example, consider the electrical circuit in Figure 6.34. When the switch is open, the circuit is modeled with a single circuit. When the switch is closed, the circuit is modeled as two separate circuits having a common electrical ground.

What if the ground was in a different location? In that case, Figure 6.35 shows that the two circuits no longer share a common ground, but rather share a common voltage, say, V_a . This point of common voltage is how the two models are related. The voltage V_a must always be the same for both parts of the circuit when the switch is closed.

6.4.9 Diodes

Like switches, diodes can cause branches in circuits to be open or closed. Diodes automatically open the circuit when they become reverse-biased. Diodes only allow current to flow in one direction, and reverse-bias means that the current is trying to flow the wrong way through the diode. Forward-bias means that the current flow is in the correct direction. The symbol for a diode looks like an arrow pointing in the forward-biased direction.

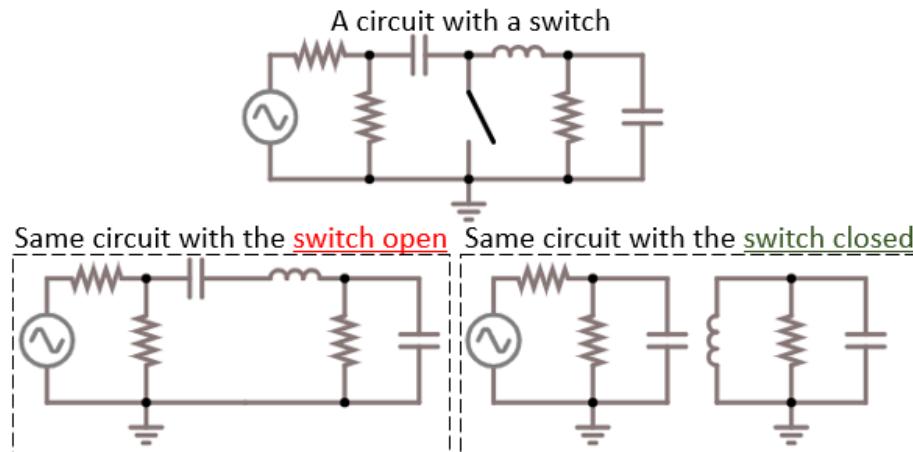


Figure 6.34 A circuit with a switch

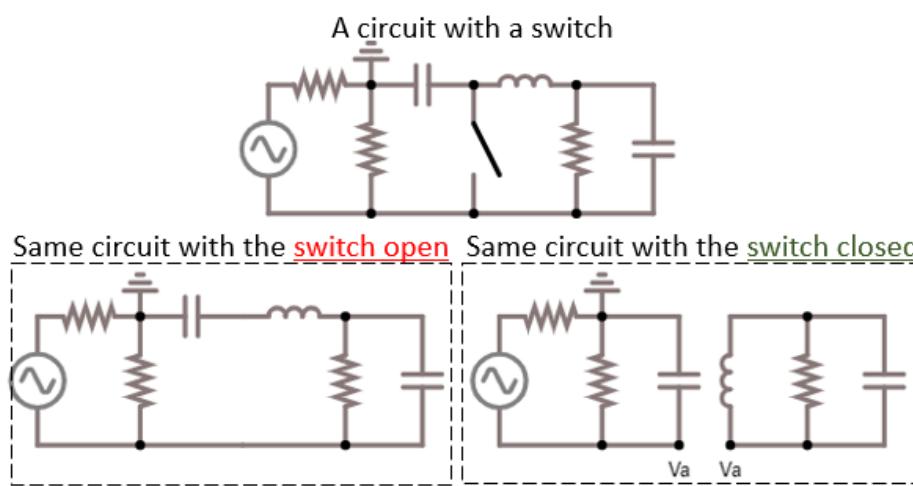


Figure 6.35 The same circuit as Figure 6.34, but with a different ground location. The switch open circuit and the switch closed circuit do not share a common ground. Rather, they now share the common voltage V_a.

6.4 Modeling Electrical Circuits with $\dot{x} = Ax + Bu$

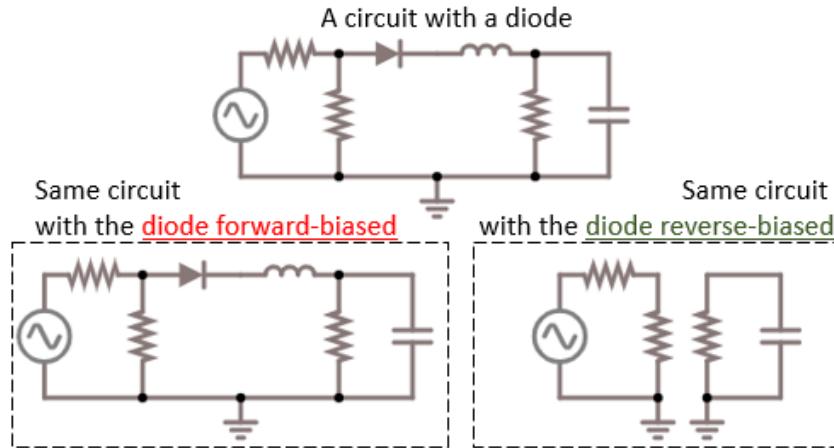


Figure 6.36 A circuit with a diode

Consider the circuit of Figure 6.36. When the diode is forward-biased, the circuit is modeled with a single circuit. When the diode is reverse-biased, the circuit is modeled as two separate circuits having a common ground voltage.

6.4.10 Buck Converter: A Switch and Diode Circuit Example

Switches and diodes often are used in power electronics applications. They can be used to boost the output of a circuit to a higher voltage – with the trade-off of a lower electrical current. The circuit that increases the output voltage is called a boost converter. Switches and diodes can also be used to decrease the output of a circuit to a lower voltage in a buck converter. The following example shows how to use state-space equations to model the dynamic behavior of a buck converter.

Buck Converter Circuit

Example 6.4.5. Buck converter circuit

Derive the dynamic equations for the buck converter of Figure 6.37. Use the sign notation shown in the figure. The input voltage is V_{in} . The diode forward bias voltage is V_D . The inductance is L . The capacitance is C , and the resistance is R . Consider all possible switch and diode conditions.

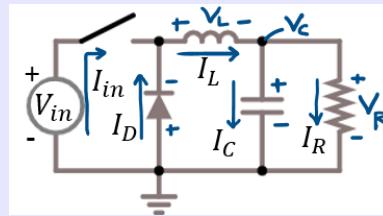


Figure 6.37 A buck converter with labeled sign notation

Solution: The buck converter has a switch and a diode. The switch has two possible conditions: open and closed. The diode also has two possible conditions: forward and reverse biased. We

therefore must consider four different circuit models and derive the equations for each. The first model is for the switch closed and diode forward-biased condition. The resulting circuit is shown in Figure 6.38.

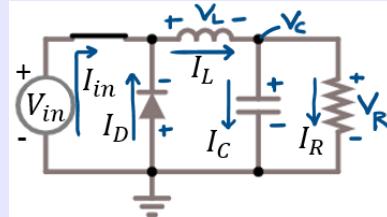


Figure 6.38 Switch closed and diode forward-biased

We choose the state variables to be the voltage drop across the capacitor V_C and the current in the inductor I_L . We use the following steps to derive the state equations.

$$\begin{aligned} C \dot{V}_C &= I_C \\ &= I_L - I_R \\ &= I_L - \frac{V_R}{R} \\ &= I_L - \frac{V_C}{R} \end{aligned} \quad \begin{aligned} L \dot{I}_L &= V_L \\ &= \max(V_{in}, -V_D) - V_C \end{aligned}$$

Figure 6.39 Derivation for the switch closed and diode forward-biased condition

The state equations are therefore

$$\begin{bmatrix} \dot{V}_C \\ \dot{I}_L \end{bmatrix} = \begin{bmatrix} \frac{-1}{CR} & \frac{1}{C} \\ \frac{-1}{L} & 0 \end{bmatrix} \begin{bmatrix} V_C \\ I_L \end{bmatrix} + \begin{bmatrix} 0 \\ \frac{1}{L} \end{bmatrix} \max(V_{in}, -V_D) \quad (6.15)$$

The second circuit model is for the switch closed and diode reverse-biased condition. The resulting circuit is shown in Figure 6.40.

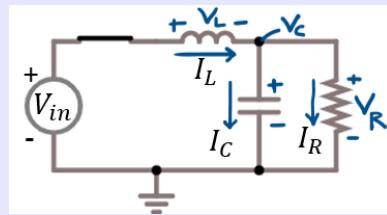


Figure 6.40 Switch closed and diode reverse-biased

We must use the same state variables in the same order as before to be consistent. The derivation is shown below:

6.4 Modeling Electrical Circuits with $\dot{x} = Ax + Bu$

$$\begin{aligned} C\dot{V}_C &= I_C \\ &= I_L - I_R \\ &= I_L - \frac{V_R}{R} \\ &= I_L - \frac{V_C}{R} \end{aligned}$$

$$\begin{aligned} L\dot{I}_L &= V_L \\ &= V_{in} - V_C \end{aligned}$$

Figure 6.41 Derivation for the switch closed and diode reverse-biased condition

The resulting state equations are

$$\begin{bmatrix} \dot{V}_C \\ \dot{I}_L \end{bmatrix} = \begin{bmatrix} \frac{-1}{CR} & \frac{1}{C} \\ \frac{-1}{L} & 0 \end{bmatrix} \begin{bmatrix} V_C \\ I_L \end{bmatrix} + \begin{bmatrix} 0 \\ \frac{1}{L} \end{bmatrix} V_{in} \quad (6.16)$$

The third circuit model is for the switch open and diode forward-biased condition. The resulting circuit is shown in Figure 6.42.

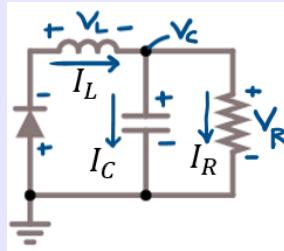


Figure 6.42 Switch open and diode forward-biased

Keeping the same state variables and sign notation, the derivation becomes:

$$\begin{aligned} C\dot{V}_C &= I_C \\ &= I_L - I_R \\ &= I_L - \frac{V_R}{R} \\ &= I_L - \frac{V_C}{R} \end{aligned} \quad \begin{aligned} L\dot{I}_L &= V_L \\ &= -V_D - V_C \end{aligned}$$

Figure 6.43 Derivation for the switch open and diode forward-biased condition

The state equations for the buck converter with the switch open and the diode forward-biased are

$$\begin{bmatrix} \dot{V}_C \\ \dot{I}_L \end{bmatrix} = \begin{bmatrix} \frac{-1}{CR} & \frac{1}{C} \\ \frac{-1}{L} & 0 \end{bmatrix} \begin{bmatrix} V_C \\ I_L \end{bmatrix} + \begin{bmatrix} 0 \\ \frac{-1}{L} \end{bmatrix} V_D \quad (6.17)$$

The fourth and final circuit model is for the switch open and diode reverse-biased condition. The resulting circuit is shown in Figure 6.44.

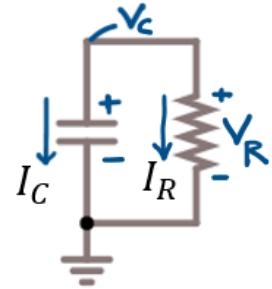


Figure 6.44 Switch open and diode reverse-biased

The derivation becomes:

$$\begin{aligned} C \dot{V}_C &= I_C \\ &= -I_R \\ &= -\frac{V_R}{R} \\ &= -\frac{V_C}{R} \end{aligned} \quad I_L = 0$$

Figure 6.45 Derivation for the switch-open and diode reverse-biased condition

The state equations are

$$\dot{V}_C = \frac{-1}{RC} V_C \quad (6.18)$$

$$I_L = 0 \quad (6.19)$$

The circuit models and corresponding state equations for the four switch and diode conditions are summarized in the figure below:

6.5 Summary: Steps for Solving Electrical Circuits

<p>Switch closed and diode forward biased</p> $\begin{bmatrix} \dot{V}_C \\ \dot{I}_L \end{bmatrix} = \begin{bmatrix} -\frac{1}{CR} & \frac{1}{C} \\ \frac{-1}{L} & 0 \end{bmatrix} \begin{bmatrix} V_C \\ I_L \end{bmatrix} + \begin{bmatrix} 0 \\ \frac{1}{L} \end{bmatrix} \max(V_{in}, -V_D)$	<p>Switch open and diode forward biased</p> $\begin{bmatrix} \dot{V}_C \\ \dot{I}_L \end{bmatrix} = \begin{bmatrix} -\frac{1}{CR} & \frac{1}{C} \\ \frac{-1}{L} & 0 \end{bmatrix} \begin{bmatrix} V_C \\ I_L \end{bmatrix} + \begin{bmatrix} 0 \\ \frac{1}{L} \end{bmatrix} V_D$
<p>Switch closed and diode reverse biased</p> $\begin{bmatrix} \dot{V}_C \\ \dot{I}_L \end{bmatrix} = \begin{bmatrix} -\frac{1}{CR} & \frac{1}{C} \\ \frac{-1}{L} & 0 \end{bmatrix} \begin{bmatrix} V_C \\ I_L \end{bmatrix} + \begin{bmatrix} 0 \\ \frac{1}{L} \end{bmatrix} V_{in}$	<p>Switch open and diode reverse biased</p> $\dot{V}_C = \frac{-1}{RC} V_C$ $I_L = 0$

Figure 6.46 Summary of the circuit models and state equations for the four different switch and diode combinations for the buck converter of Figure 6.37

6.5 Summary: Steps for Solving Electrical Circuits

This section recommends steps for solving electrical circuits using $\dot{x} = Ax + Bu$. It applies these steps to solve a circuit with a diode and a switch.

1. Reduce the Circuit to its Simplest Equivalent Circuit
 - (a) Reduce the circuit using equivalent resistances, capacitances, inductances, and sources
2. Assign Labels and Signs
 - (a) Assign a current to each branch of the circuit and establish the positive current flow direction
 - (b) Label the voltage drop sign convention on all circuit elements to agree with the direction of current flow
3. Identify Input (u) and State Variables (x) for $\dot{x} = Ax + Bu$

- (a) Sources and states are the independent variables
 - (b) Sources are the input variables (u)
 - (c) Capacitor voltages are state variables (x)
 - (d) Inductor currents are state variables (x)
4. Solve the Circuit
- (a) Write the state-equations for each capacitor and inductor
 - (b) Use Kirchhoff's laws and Ohm's law to eliminate dependent variables (the dependent variables are the unknown voltages and currents that are not state or input variables)

The following example uses these steps to solve the circuit shown in Figure 6.47.

State-Space Modeling of a Circuit with a Switch and a Diode

Example 6.5.1. State-space modeling of a circuit with a switch and a diode

Calculate the voltage drop across the 50Ω resistor in Figure 6.47 as a function of time. The switch is closed for the first 5 minutes, then it is open for 5 minutes. The input voltage is $V_{in} = 10 \cos(4\pi t)$. The capacitors and inductors are initially uncharged.

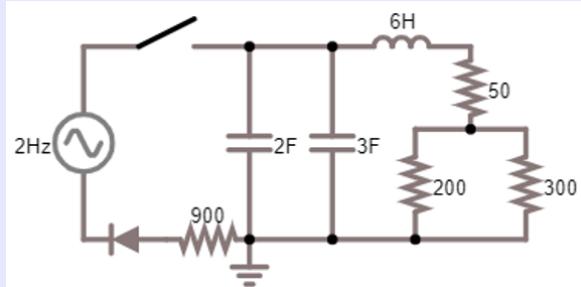


Figure 6.47 An electrical circuit with a switch and a diode

Solution: We will use the steps described above to solve this electrical circuit. The first step is to reduce the circuit to its simplest equivalent form. This is shown in Figure 6.48.

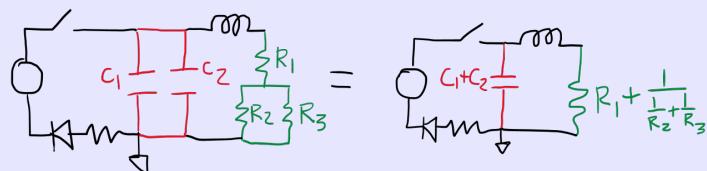


Figure 6.48 Simplest equivalent circuit

The next step is to assign labels and signs to the branch currents and component voltages. This is shown in Figure 6.49.

6.5 Summary: Steps for Solving Electrical Circuits

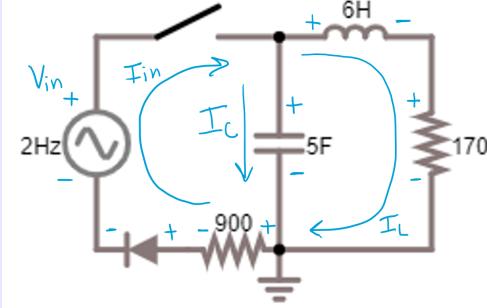


Figure 6.49 Simplest Equivalent Circuit with Labels and Signs

It is important to note that the output is the voltage drop across the 50Ω resistor in Figure 6.47. According to the new labels and sign convention of Figure 6.49, the output is $V_{50} = 50I_L$.

The next step is to identify the input and state variables. The input variables are the source voltage V_{in} and the diode voltage V_D . The state variables are the capacitor voltage V_C and the inductor current I_L .

The final step is to solve the circuit. We begin by writing the equations for each capacitor and inductor:

$$\dot{V}_C = \frac{1}{C} I_C$$

$$\dot{I}_L = \frac{1}{L} V_L$$

Now we use Kirchhoff's laws and Ohm's law to eliminate dependent variables. Using Kirchhoff's voltage law around the right-hand loop eliminates the dependent variable V_L , because $V_L = V_C - 170I_L$. Both V_C and I_L are state variables, and so we can write the second state equation as

$$\dot{I}_L = \frac{1}{L} (V_C - 170I_L)$$

Using Kirchhoff's current law at the node above the capacitor, the capacitor current is $I_C = I_{in} - I_L$. I_L is a state variable, but I_{in} is a dependent variable that we must eliminate. To eliminate it, because the circuit has a switch and a diode, we must consider 4 different possibilities. The first possibility is switch-closed and diode forward biased. In that case, using Kirchhoff's voltage law around the left-hand loop combined with Ohm's law for the 900Ω resistor gives

$$I_{in} = \frac{V_{in} - V_D - V_C}{900}$$

The second possibility is switch closed, but diode reverse biased. In that case

$$I_{in} = 0$$

The third possibility is switch opened and diode forward biased. The fourth possibility is switch open and diode reverse biased. Both result in the same equation for the current:

$$I_{in} = 0$$

As a result of the four possibilities, the dependent variable I_{in} is

$$I_{in} = \begin{cases} \frac{V_{in} - V_D - V_C}{900} & \text{Switch Closed and } V_{in} > V_C + V_D \\ 0 & \text{Otherwise} \end{cases}$$

Plugging this result back into the state equations gives

$$\begin{bmatrix} \dot{V}_C \\ \dot{I}_L \end{bmatrix} = \begin{bmatrix} \frac{-1}{900C} & \frac{-1}{C} \\ \frac{1}{L} & \frac{-170}{L} \end{bmatrix} \begin{bmatrix} V_C \\ I_L \end{bmatrix} + \begin{bmatrix} \frac{1}{900C} & \frac{-1}{900C} \\ 0 & 0 \end{bmatrix} \begin{bmatrix} V_{in} \\ V_D \end{bmatrix}$$

if the switch is closed and $V_{in} > V_C + V_D$. Otherwise, it gives

$$\begin{bmatrix} \dot{V}_C \\ \dot{I}_L \end{bmatrix} = \begin{bmatrix} 0 & \frac{-1}{C} \\ \frac{1}{L} & \frac{-170}{L} \end{bmatrix} \begin{bmatrix} V_C \\ I_L \end{bmatrix} + \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} V_{in} \\ V_D \end{bmatrix}$$

We need to determine the initial conditions for the state variables V_C and I_L . Initially, the capacitors in Figure 6.47 were uncharged. They were in parallel, so they must have the same voltage of 0 V. The inductor was also initially uncharged, and therefore $I_L = 0$ initially. The following MATLAB code implements the solution and produces the graph of Figure 6.50.

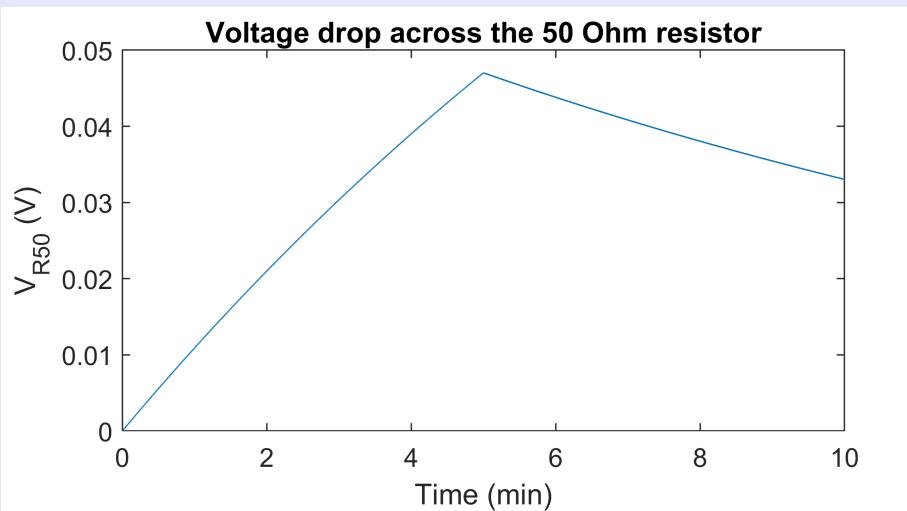


Figure 6.50 Circuit response

6.6 Modeling Batteries using Equivalent Circuit Models

```
close all %close all open figures
clear %clear all variables
clc %clear the Command Window
C = 5; R1 = 900; R2 = 170; L = 6; VD = 0.6; %Set the parameters
dt = 0.001; %(s) time step
t = 0:dt:10*60; %(s) Time vector from 0 to 10 minutes
N = length(t); %Length of the time vector
Vin = 10*cos(4*pi*t); %(V) Voltage source input

%Switch closed and V_in > V_C + V_D
A1 = [-1/C/R1,-1/C;1/L,-R2/L] %A matrix in dx=Ax+Bu
B1 = [1/C/R1,-1/C/R1;0,0] %B matrix in dx=Ax+Bu
Fd1 = expm([A1*dt, B1*dt;zeros(2,4)]);
Ad1 = Fd1(1:2,1:2);
Bd1 = Fd1(1:2,3:4);
%otherwise
A2 = [0,-1/C;1/L,-R2/L] %A matrix in dx=Ax+Bu
B2 = [0,0;0,0];
Fd2 = expm([A2*dt, B2*dt;zeros(2,4)]);
Ad2 = Fd2(1:2,1:2);
Bd2 = Fd2(1:2,3:4);

V_R50 = zeros(1,N); %(V) An array to store the voltage across R50
x = [0;0]; % initial condition
for ii = 1:N
    Vc = x(1); %(V) The first state is the capacitor voltage
    IL = x(2); %(A) The second state is the inductor current
    u = [Vin(ii);VD]; %(V) input for dx=Ax+Bu
    if t(ii)<=5*60 && Vin(ii)-Vc-VD>0 %Switch closed and current>0
        Ad = Ad1; %Use the appropriate discrete Ad matrix
        Bd = Bd1; %Use the appropriate discrete Bd matrix
    else %Switch is open, or diode current <= 0
        Ad = Ad2; %Use the appropriate discrete Ad matrix
        Bd = Bd2; %Use the appropriate discrete Bd matrix
    end
    V_R50(ii) = 50 * IL; %(V)Voltage drop across 50 Ohm resistor
    x = Ad*x+Bd*u; %iterative solution to dx=Ax+Bu
end
figure, %open a new figure
plot(t/60,V_R50) %Plot the 50 Ohm resistor voltage vs. time
xlabel('Time (min)') %(min) x-axis label
ylabel('V_R_5_0 (V)') %(V) y-axis label
title('Voltage drop across the 50 Ohm resistor')
```

6.6 Modeling Batteries using Equivalent Circuit Models

Batteries are energy storage devices that may have a dynamic response. As the battery charges and discharges, the current and voltage change with time. This is especially true in automotive and aerospace applications where the battery may be used to provide propulsion energy. There are many different types

of battery models. Some of the most complex models use finite element software packages running multiple CPUs (central processing units) to track ion migration, electrochemical reactions, temperature distributions, charge distributions, diffusion, side reactions, and many other phenomena. On the extreme opposite end is the sometimes overly-simple model of treating the battery as an ideal voltage source. Computers that monitor battery properties in real-time often use models with a complexity somewhere between these extremes. These models can help determine the state-of-charge (SOC) of the battery, help diagnose battery faults that could lead to premature failures, or control the voltage and electrical currents of the battery. These medium complexity models are called control-oriented models. One physics-based example is provided in the Batteries Book in the Textbook module in I-Learn.

One of the most popular types of control-oriented battery models is the Equivalent Circuit Model (ECM). Equivalent circuit models treat the battery as a series of resistor-capacitor components connected to an OCV source that is a function of the battery's SOC. The dynamic response of batteries can be approximated by an Equivalent Circuit Model (ECM) such as the one in Figure 6.51.

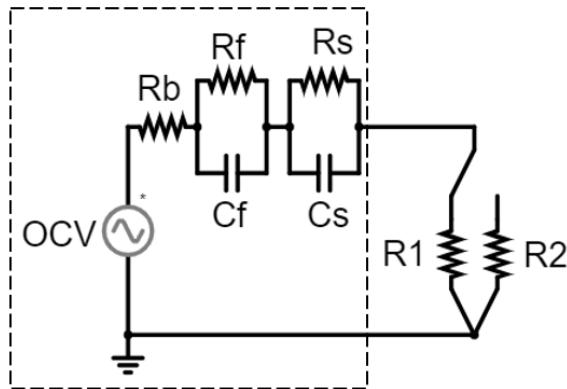


Figure 6.51 Equivalent Circuit Model (ECM) of a battery connected to a resistive load by a switch

The model of the battery includes only the circuitry inside the dashed box. The resistors R_1 and R_2 and switch outside the dashed box are not part of the battery model but are part of an external circuit. The resistance R_b is an internal battery resistance. Physically, it could be partly caused by the battery's resistance to ion migration through the membrane separating the anode and cathode.

The circuit formed by resistance R_f and capacitance C_f models the fast dynamics of the battery. The fast dynamics of a lithium ion battery, for example, typically have a time-constant of 10 seconds or less. Fast dynamics could be caused by electrochemical reaction kinetics and ion adsorption at the surfaces of the battery electrodes.

The circuit formed by resistance R_s and capacitance C_s models the slow dynamics of the battery. Physically, slow dynamics could be caused by ionic or atomic diffusion. The time constant for these dynamics could be around hundreds of seconds.

The Open Circuit Voltage (OCV) is the voltage of the battery after the switch has been open for long enough that all transient effects have worn off. For equivalent circuit models, it is typically determined experimentally. The Open Circuit Voltage (OCV) of the battery is a function of the battery's state-of-charge (SOC). The OCV curve is often a nonlinear function of the SOC. Actual data for a lithium ion battery is shown in Figure 6.52.

The battery current I is related to the SOC S by the following equation:

6.7 Modeling Electrical Circuits with Op Amps

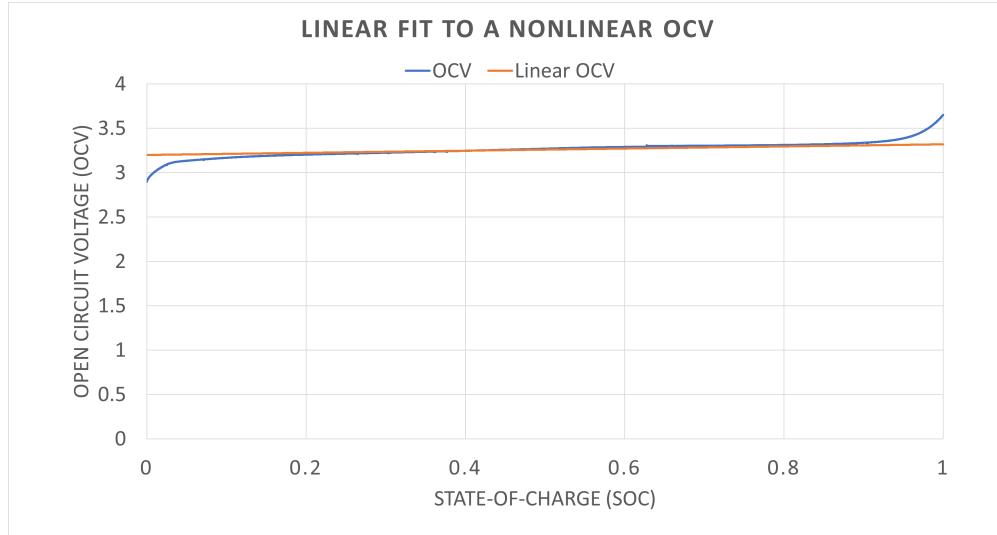


Figure 6.52 Nonlinear and linearized OCV curves as a function of SOC

$$\dot{S} = \frac{-1}{Q} I \quad (6.20)$$

where Q (coulombs) is the maximum capacity of the battery, *i.e.*, the total charge when the SOC is $S = 1$. The state equations for the capacitor voltage drops V_s and V_f can be found using a circuit analysis of the ECM circuit. A circuit analysis shows that the state-equations can be written as follows:

$$\begin{bmatrix} \dot{S} \\ \dot{V}_s \\ \dot{V}_f \end{bmatrix} = \begin{bmatrix} \frac{-(OCV - V_s - V_f)}{Q(R_b + R)} \\ \frac{OCV - V_s - V_f}{C_s(R_b + R)} - \frac{V_s}{C_s R_s} \\ \frac{OCV - V_s - V_f}{C_f(R_b + R)} - \frac{V_f}{C_f R_f} \end{bmatrix} \quad (6.21)$$

If the battery voltage V is the output, the output equation is

$$V = \frac{OCV - V_s - V_f}{R_b + R} R \quad (6.22)$$

Eqs. (6.21) and (6.22) are the nonlinear state-space equations for the battery model. The methods of Chapter 4.3 can be used to solve them.

6.7 Modeling Electrical Circuits with Op Amps

An operational amplifier, simply referred to as an op amp, is typically used to amplify a voltage signal to a higher voltage, or one with more power. For example, a pickup on an electric guitar can sense the vibrations in the guitar string and produce a voltage signal. However, that voltage signal is weak, and it is not powerful enough to drive a speaker. This is why a guitar is plugged into an amp, or amplifier, which increases the signal power enough to drive a speaker and produce sound. Op amps are used to amplify a voltage signal from a sensor, for example a temperature or pressure sensor, to be recorded or used in a

feedback loop. Op amps can play an important role in signal conditioning, which means that a voltage signal is conditioned for better use or recording.

An op amp is a combination of transistors, resistors, and capacitors, but in a circuit diagram, it is simply represented by the symbol in Figure 6.53.

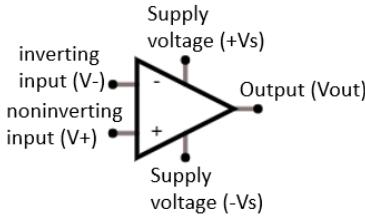


Figure 6.53 The symbol for an op amp in an electrical circuit diagram

The electrical power from the supply voltage amplifies or otherwise conditions the signals at the inverting and non-inverting inputs of the op amp. Therefore, the output voltage and current of the op amp are determined by both the supply and signal inputs. If the supply voltages are not connected, the op amp is not powered and therefore does not work; many circuit diagrams draw the op amp without explicitly showing the supply inputs. When the op amp is drawn without the supply inputs, it is assumed that the op amp is supplied with sufficient power.

Many electrical circuits connect the output of the op amp to the inverting input using electrical components, such as a resistor, capacitor, inductor, or combination of components. When electrical components connect the op amp output to the inverting input, the op amp is said to have **feedback**. Without feedback, the output voltage of the op amp becomes saturated by the limits of the supply voltage $+V_s$ or $-V_s$:

$$V_{\text{out}} = \begin{cases} +V_s & \text{if } V_+ > V_- \\ -V_s & \text{otherwise} \end{cases} \quad (6.23)$$

The rules for modeling an op amp with feedback are presented in the next section.

6.7.1 Rules for Modeling Op Amps with Feedback

When an op amp has feedback, the following two rules can be used to model it:

Table 6.2 Modeling Rules for Op Amps with Feedback

Rule	Description
$I_+ = 0$	No current goes in or out of the non-inverting or inverting inputs
$I_- = 0$	The inverting and non-inverting input voltages are equal
$V_+ = V_-$	The inverting and non-inverting input voltages are equal

The modeling rules of Table 6.2 are only approximations. For the first rule, actually, a negligible amount of current enters the op amp. Inspecting the internal circuitry of an op amp reveals that the inverting and non-inverting inputs are connected to the bases of two transistors. The electrical current entering the base of a transistor is very small compared to the available current that can flow through the

6.7 Modeling Electrical Circuits with Op Amps

collector and emitter. For the second rule, negative feedback is used to control the inputs such that the non-inverting input voltage is equal to the inverting input voltage. There is a time delay associated with this control. For the applications considered in this book, the time delay is short enough that its affect is negligible. The following example illustrates how the two rules are used to model an electrical circuit with an op amp.

Buffer Op Amp Circuit

Example 6.7.1. Use the Op Amp Rules to Model a Buffer Op Amp Circuit

Use the two rules of Table 6.2 to determine the output voltage and current of the following op amp circuit if $V_{in} = 4.6$ V.

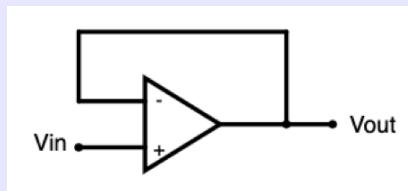


Figure 6.54 An buffer op amp circuit

Solution: The non-inverting input is connected to V_{in} , so $V_+ = V_{in} = 4.6$ V. From the second rule $V_+ = V_-$, we know that $V_- = 4.6$ V. Since V_{out} is wired directly to V_- , it must have the same voltage: $V_{out} = 4.6$ V.

The output current I_{out} is less obvious. From the op amp rules, we know that $I_- = 0$ A, and the output is directly wired to the inverting input, but *that does not mean the output current is zero. The output current in an op amp circuit is provided by the supply current plus the signal current.* Therefore, the output current depends on the circuit that is connected to the output of the op amp. Since the circuit of Figure 6.54 is an open circuit, the output current is $I_{out} = 0$ A, but if it was connected to a $10\ \Omega$ resistor, the output current would be $I_{out} = \frac{4.6\text{ V}}{10\ \Omega} = 0.46$ A.

There is no electrical circuit connected to the output of the op amp of Figure 6.54 that could affect the input voltage V_{in} and current I_{in} . In this sense, the op amp circuit is a buffer that prevents the output from affecting the input.

The buffer op amp circuit of Example 6.7.1 demonstrates the op amp modeling rules of Table 6.2, but it does not demonstrate how to model connected circuitry. The following example shows how to model an op amp circuit with resistors.

Subtractor Op Amp Circuit

Example 6.7.2. Model and solve a subtractor op amp circuit

If the values of the resistors are $R_1 = 10\ \Omega$, $R_2 = 20\ \Omega$ and $R_3 = 30\ \Omega$, and the input voltages are $V_1 = 1$ V and $V_2 = 2$ V, what are the values of V_{out} and I_{out} for the subtractor op amp circuit of Figure 6.55.

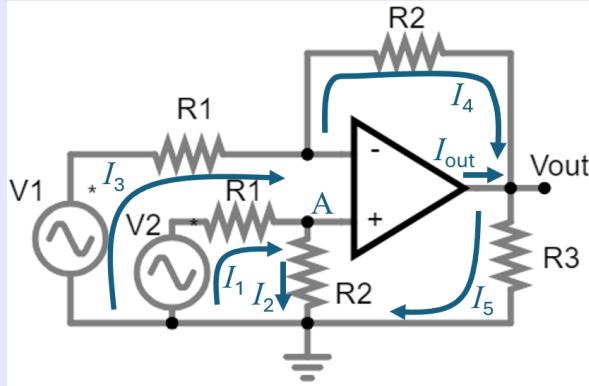


Figure 6.55 A subtractor op amp circuit

Solution: Before we can use the rule $V_- = V_+$, we must determine V_+ . The current I_1 in Figure 6.55 is the current through resistor R_1 , and it can be determined using Ohm's law:

$$I_1 = \frac{V_2 - V_+}{R_1}$$

Since no current enters the op amp at the non-inverting input, $I_+ = 0$, and therefore $I_2 = I_1$. It is the current through R_2 and can be determined using Ohm's law:

$$I_2 = \frac{V_+}{R_2}$$

Setting $I_1 = I_2$ requires that

$$\begin{aligned} \frac{V_2 - V_+}{R_1} &= \frac{V_+}{R_2} \\ V_+ &= V_2 \frac{R_2}{R_1 + R_2} \\ &= 2 \text{ V} \frac{20}{10 + 20} \\ &= \frac{4}{3} \text{ V} \end{aligned}$$

The current I_3 can be determined by Ohm's law to be

$$\begin{aligned} I_3 &= \frac{V_1 - V_-}{R_1} \\ &= \frac{V_1 - V_+}{R_1} \\ &= \frac{1 - \frac{4}{3}}{10} \\ &= \frac{-1}{30} \text{ A} \end{aligned}$$

6.7 Modeling Electrical Circuits with Op Amps

No current enters the op amp at the inverting input, $I_- = 0$, and therefore $I_4 = I_3 = \frac{-1}{30}$ A. Ohm's law states that

$$I_4 = \frac{V_- - V_{\text{out}}}{R2}$$

and so

$$\begin{aligned} V_{\text{out}} &= V_- - I_4 R2 \\ &= \frac{4}{3} + \frac{1}{30} 20 \\ &= 2 \text{ V} \end{aligned}$$

The current I_5 can be determined by Ohm's law to be

$$\begin{aligned} I_5 &= \frac{V_{\text{out}}}{R3} \\ &= \frac{2}{30} \text{ A} \end{aligned}$$

From Kirchhoff's current law, The current I_{out} is

$$\begin{aligned} I_{\text{out}} &= -I_4 + I_5 \\ &= \frac{1}{30} + \frac{2}{30} \\ &= 0.1 \text{ A} \end{aligned}$$

Signal conditioning and filtering is an important application of op amp circuits. Filtering can remove unwanted noise or bias from a signal, as discussed in Chapter 9. Filters often require energy storage components such as capacitors and inductors. The following example derives the equations for an active analog band-stop filter.

Circuit for an Active Analog Band Stop Filter

Example 6.7.3. Derive the equations for an active analog band stop filter

Band stop filters are introduced in Section 9.1.8. They have the ability to remove unwanted noise with a certain frequency range from a signal. Figure 6.56 shows a circuit that implements a twin-T analog band stop filter. Derive the equations for the filter transfer function.

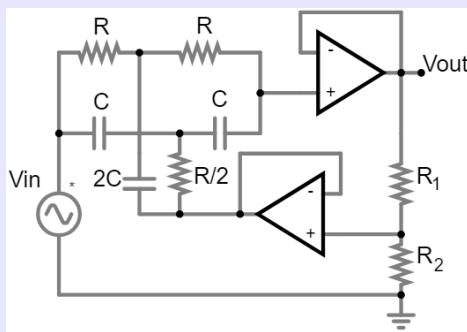


Figure 6.56 Circuit for an active band stop filter

Solution: We will analyze the circuit in the Laplace domain. To do so, we first replace the circuit components with their Laplace impedance as shown in Figure 6.57. The impedance Z is $Z = \frac{1}{sC}$.

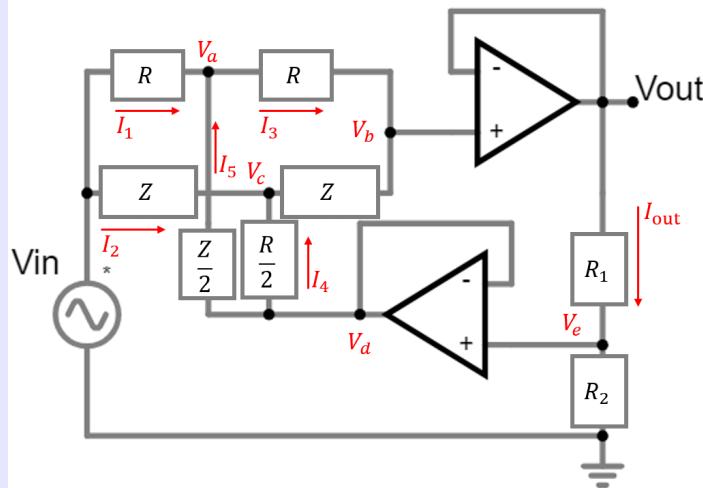


Figure 6.57 An active band stop filter showing Laplace impedances

The equations for this circuit are derived using Kirchhoff's current law and Ohm's law. The equations for the five electrical currents are calculated using Ohm's law:

$$I_1 = \frac{V_{in} - V_a}{R}$$

$$I_2 = \frac{V_{in} - V_c}{Z}$$

$$I_3 = \frac{V_a - V_c}{R + Z}$$

$$I_4 = \frac{2V_d - 2V_c}{R}$$

$$I_5 = \frac{2V_d - 2V_a}{Z}$$

6.7 Modeling Electrical Circuits with Op Amps

Kirchhoff's current law is applied at nodes a and c :

$$\begin{aligned} I_1 - I_3 + I_5 &= 0 \\ I_2 + I_3 + I_4 &= 0 \end{aligned}$$

The voltage divider law determines the voltage V_e to be

$$V_e = V_{\text{out}} \frac{R_2}{R_1 + R_2}$$

Ohm's law relates the voltage V_b to V_a and I_3 :

$$V_a - V_b = I_3 R$$

The rules for op amps also require the following:

$$\begin{aligned} V_d &= V_e \\ V_b &= V_{\text{out}} \end{aligned}$$

We can simultaneously solve all the above equations by creating a matrix solution:

$$\left[\begin{array}{ccccccc|c|c} 0 & \frac{-1}{R} & 0 & 0 & -1 & 0 & 0 & 0 & 0 \\ 0 & 0 & \frac{-1}{Z} & 0 & 0 & -1 & 0 & 0 & 0 \\ 0 & \frac{1}{R+Z} & \frac{-1}{R+Z} & 0 & 0 & 0 & -1 & 0 & 0 \\ 0 & 0 & \frac{-2}{R} & \frac{2}{R} & 0 & 0 & 0 & -1 & 0 \\ 0 & \frac{-2}{Z} & 0 & \frac{2}{Z} & 0 & 0 & 0 & 0 & -1 \\ 0 & 0 & 0 & 0 & 1 & 0 & -1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 \\ \frac{R_2}{R_1+R_2} & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 \\ -1 & 1 & 0 & 0 & 0 & 0 & -R & 0 & 0 \end{array} \right] \begin{bmatrix} V_{\text{out}} \\ V_a \\ V_c \\ V_d \\ I_1 \\ I_2 \\ I_3 \\ I_4 \\ I_5 \end{bmatrix} = \begin{bmatrix} \frac{-1}{R} \\ \frac{-1}{Z} \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} V_{\text{in}} \quad (6.24)$$

If we let f be the fraction $\frac{R_2}{R_1+R_2}$, the matrix equation can be solved in MATLAB using its Symbolic Math Toolbox:

```
syms R f Z

A = [0,-1/R,0,0,-1,0,0,0,0;...
      0,0,-1/Z,0,0,-1,0,0,0;...
      0,1/(R+Z),-1/(R+Z),0,0,0,-1,0,0;...
      0,0,-2/R,2/R,0,0,0,-1,0;...
      0,-2/Z,0,2/Z,0,0,0,0,-1;...
      0,0,0,0,1,0,-1,0,1;...
      0,0,0,0,0,1,1,1,0;...
      f,0,0,-1,0,0,0,0,0;...
      -1,1,0,0,0,0,-R,0,0]
```

```
B = [-1/R ; -1/Z ; 0 ; 0 ; 0 ; 0 ; 0 ; 0 ; 0]
[1,zeros(1,8)]*(A\B) %Vout / Vin
```

The output to the MATLAB command window is

```
(R^2 + Z^2)/(4*R*Z + R^2 + Z^2 - 4*R*Z*f)
```

Making the substitution $Z = \frac{1}{sC}$ and using $f = \frac{R_2}{R_1+R_2}$ results in the following transfer function:

$$\frac{V_{\text{out}}}{V_{\text{in}}} = \frac{C^2 R^2 s^2 + 1}{C^2 R^2 s^2 + 4RC(1-f)s + 1} \quad (6.25)$$

This transfer function is an equation for a band-stop filter as described in Section 9.1.8. The rejected frequency is $\omega_0 = \frac{1}{RC}$ and the width of the stopped frequency band is $\omega_Q = \frac{4(1-f)}{RC}$. The width of the stopped frequency band can be adjusted by changing the values of R_1 and R_2 .

The following example demonstrates another application of op amps for signal conditioning. It derives the equations for the active low-pass filter.

Active Low Pass Filter Op Amp Circuit

Example 6.7.4. Derive the transfer function for a low pass filter op amp circuit
Determine the output of the op amp circuit in Figure 6.58.

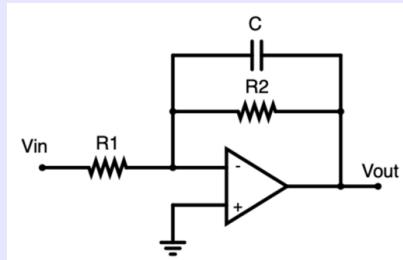


Figure 6.58 Op amp low-pass filter circuit

Solution: We can start by combining the capacitor C and resistor R_2 into an equivalent impedance:

$$\begin{aligned}\frac{1}{Z_{eq}} &= \frac{1}{1/(Cs)} + \frac{1}{R_2} \\ Z_{eq} &= \frac{R_2}{R_2 Cs + 1}\end{aligned}$$

Now we can model the simplified op amp circuit using the assumed currents in Figure 6.59.

6.8 Modeling DC Motors

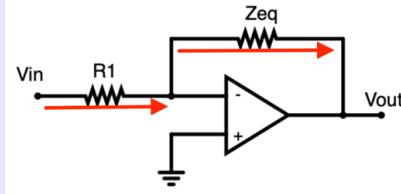


Figure 6.59 Simplified circuit of Figure 6.58.

The nodal equation is

$$\frac{V_{in} - 0}{R_1} = \frac{0 - V_{out}}{Z_{eq}}$$

and the output voltage is:

$$\begin{aligned} V_{out} &= -\frac{Z_{eq}}{R_1} V_{in} \\ &= -\frac{R_2/R_1}{R_2 C s + 1} V_{in} \end{aligned}$$

From this solution, we can solve for the closed-loop gain, or the ratio of the output voltage to input voltage, to get:

$$TF(s) = \frac{V_{out}}{V_{in}} = -\frac{R_2}{R_1} \frac{1}{R_2 C s + 1} = -\frac{R_2}{R_1} \frac{1}{\tau s + 1}$$

which is the transfer function of a low-pass filter with a time constant $\tau = R_2 C$. This can also be rearranged to get

$$TF(s) = -\frac{R_2}{R_1} \frac{\omega_c}{s + \omega_c}$$

which is also the transfer function of a low-pass filter with a cutoff frequency of

$$\omega_c = \frac{1}{R_2 C}$$

This is an analog method of applying a low-pass filter using op amps.

6.8 Modeling DC Motors

An electrical motor is a mechatronic system. Mechatronic systems often include both mechanical and electrical parts. For example, a laptop computer is a mechatronic system. It has keys that convert mechanical motion to electrical signals. It has a screen that converts electrical signals to light and optical signals. Some laptops have spinning hard drives or CD drives that use a motor to convert electrical signals to rotational motion. Touch screens and touch pads convert swipes and taps to electrical signals. The speakers on a laptop convert electrical vibrations to mechanical vibrations, which are converted to fluid vibrations of the air that become sound waves. A camera on the computer converts light to electrical signals. Microphones convert sound waves to mechanical vibrations and finally to electrical signals. A DC

motor converts electrical power to rotational power.



Figure 6.60 A brushless DC motor with visible armature windings

Motors and generators convert rotational power to electrical power and vice-versa. A motor consists of coils of wires wrapped multiple times around an iron or other magnetically conductive core. These coils act like inductors. The wire coils are usually made of long strands of wire. Long strands of wire have some resistance. Permanent magnet DC motors have a motor constant k_T (Nm/A or Vs/rad) that relates electrical current to motor torque. The constant is the ratio of motor torque T (Nm) over armature current I (A) (or back electromotive force (ε) in volts over angular velocity ω (rad/s)) and is given by:

$$k_T = \frac{T}{I} = \frac{\varepsilon}{\omega} = BLnd \quad (6.26)$$

where B (N/m/A) is the magnetic flux density, L (m) is the length of the armature wire in the magnetic field, n (unitless) is the number of wire wraps, and d (m) is the effective diameter of the armature coil. The rotor has rotational inertia J (kg m^2) and frictional damping b (Nms/rad). Schematically, the model of the motor can be seen in Figure 6.61.

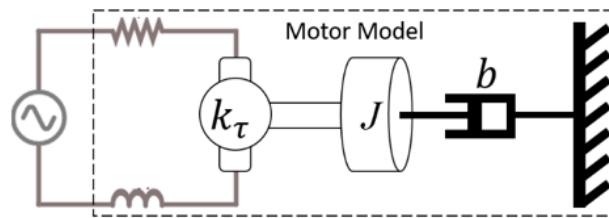


Figure 6.61 The motor model consists of electrical resistance and inductance, a motor constant, rotational inertia, and rotational friction.

If the armature resistance is R , and the armature inductance is L , doing a Kirchhoff's voltage loop around the electrical circuit will give:

$$V_{in} - RI - \varepsilon - L\dot{I} = 0 \quad (6.27)$$

6.8 Modeling DC Motors

The electromotive force ε is the voltage loss by converting the electrical energy to rotational energy. It can be related to angular velocity by $\varepsilon = k_T \omega$. Then Eq. (6.27) becomes:

$$V_{in} - RI - k_T \omega - L \dot{I} = 0$$

Thus, the state-equation for the current I is related to the battery or input motor voltage V_{in} by

$$\dot{I} = -\frac{R}{L}I - \frac{k_T}{L}\omega + \frac{1}{L}V_{in} \quad (6.28)$$

The inertia J of the rotor also experiences rotational dynamics (see Section 6.2). The rotational state equation of the rotor is

$$\dot{\omega} = \frac{1}{J}(T - b\omega) \quad (6.29)$$

where b (Nms/rad) is the frictional damping coefficient, and T is the motor torque which can be replaced by $T = k_T I$. Then Eq. (6.29) becomes

$$\dot{\omega} = \frac{k_T}{J}I - \frac{b}{J}\omega \quad (6.30)$$

Combining Eqs. (6.28) and (6.30) results in the following DC motor state equation:

$$\begin{bmatrix} \dot{I} \\ \dot{\omega} \end{bmatrix} = \begin{bmatrix} -\frac{R}{L} & -\frac{k_T}{L} \\ \frac{k_T}{J} & -\frac{b}{J} \end{bmatrix} \begin{bmatrix} I \\ \omega \end{bmatrix} + \begin{bmatrix} \frac{1}{L} \\ 0 \end{bmatrix} V_{in} \quad (6.31)$$

Most often, the rotor is attached to a load, and the inertia and damping of the load is combined with the inertia and damping of the rotor into one total inertial component. An example of a battery powered car will demonstrate this. Sometimes, the inertia and damping of the load dominates, and the armature inductance, rotor inertia, rotor frictional damping, and electrical resistance are relatively negligible.

Wheels convert rotational motion to translational motion. The translational velocity \dot{x} (m/s) is related to the rotational velocity ω (rad/s) by the wheel radius r_w (m):

$$\dot{x} = r_w \omega$$

Sometimes, if a transmission is involved, the gear ratio is lumped in with the wheel radius. In that case, ω is the angular velocity of the motor, \dot{x} is the velocity of the translational system (e.g. vehicle), and r_w is the total gear ratio of the drivetrain, including the wheel radius. The translational force is related to the torque by the equation

$$F = \frac{T}{r_w} \quad (6.32)$$

Modeling a Battery Powered Car

Example 6.8.1. Modeling a battery powered car

A battery powered car is an example of a mechatronic system that combines electrical, rotational, and translational dynamics. Use a battery powered vehicle to demonstrate multiple forms of energy combined into a single mathematical model. Derive a transfer function model of a battery powered car.

Solution: The following electrical car example combines electrical, rotational, and translational

dynamic systems. The equations of motion for the car are derived from Newton's second law:

$$F - b\dot{x} - m\ddot{x} = 0 \quad (6.33)$$

The force F propelling the car forward is equal to the motor torque divided by the total gear ratio of the drivetrain, including the wheel radius. This assumes that the motor inertia can be lumped in with the vehicle mass (inertia) and the motor damping can be lumped in with the air drag and other frictional drag forces. The torque is related to the current through the motor constant k_T as derived in Eq. (6.26). Therefore, Equation 6.33 can be written as a function of current I instead of force F :

$$\frac{k_T}{r_w} I - b\dot{x} - m\ddot{x} = 0 \quad (6.34)$$

Eqs. (6.34) and (6.27) make up two state equations that govern the behavior of the motorized car. We can rewrite these equations as

$$\begin{aligned}\dot{I} &= -\frac{R}{L}I - \frac{k_T}{r_w L} \dot{x} + \frac{1}{L} V_{in} \\ \ddot{x} &= \frac{k_T}{m r_w} I - \frac{b}{m} \dot{x}\end{aligned}$$

Defining state variables $x_1 = I$ and $x_2 = \dot{x}$, these equations can be written in state-equation form:

$$\begin{bmatrix} \dot{x}_1 \\ \dot{x}_2 \end{bmatrix} = \begin{bmatrix} -\frac{R}{L} & -\frac{k_T}{r_w L} \\ \frac{k_T}{m r_w} & -\frac{b}{m} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} + \begin{bmatrix} \frac{1}{L} \\ 0 \end{bmatrix} V_{in} \quad (6.35)$$

The output y is the velocity $\dot{x} = x_2$ of the car, so the output equation is

$$y = \begin{bmatrix} 0 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \quad (6.36)$$

These equations show that the battery powered car, which combined electrical, rotational, and translational components, can be modeled with a single set of equations. To show this more clearly, we can derive a transfer function from the battery supply voltage to the velocity. Section 2.2.2 found that $\frac{Y}{U} = C(sI - A)^{-1}B + D$. Therefore, the transfer function from the input voltage to velocity is

$$\frac{Y}{V_{in}} = \frac{\frac{k_T}{Lmr_w}}{s^2 + \left(\frac{R}{L} + \frac{b}{m}\right)s + \frac{Rb}{Lm} + \frac{k_T^2}{r_w^2 m L}} \quad (6.37)$$

6.9 Modeling Dynamic Diffusion and Heat Transfer

This section describes numerical solutions to the diffusion equation, which, in one form is also the heat equation. To understand the diffusion equation, we must first understand the concepts of flux and the laws of conservation.

6.9 Modeling Dynamic Diffusion and Heat Transfer

6.9.1 Flux

Flux is the movement of something through an area over a certain amount of time. For example, if 35 g of water pass through a 5 m^2 opening in a channel in exactly one second, the flux of water through the opening is $\frac{35 \text{ g}}{(5 \text{ m}^2)(1 \text{ s})} = 7 \text{ g}/(\text{m}^2\text{s})$. In this example, the flux is “mass flux” because mass is passing through the area. Other types of flux are also possible. For example, if we divide the mass flux in the example above by the molar mass of water, $M = 18.015 \text{ g/mol}$, then the “molar flux” of water is $\frac{7 \text{ g}/(\text{m}^2\text{s})}{18.015 \text{ g/mol}} = 0.3886 \text{ mol}/(\text{m}^2\text{s})$.

“Thermal flux” or “heat flux” is the transfer of heat energy through an area over a certain amount of time. Heat flux has units of $\text{J}/(\text{m}^2\text{s})$. If we multiply flux by the area, we get flow rate. Multiplying mass flux by the area results in mass flow rate. Multiplying heat flux by the area results in heat transfer rate. Mathematical symbols for flux often depend on the type of flux. Mass flux is sometimes denoted by the symbol J . The symbol N often denotes molar flux, and q is used for heat flux. Using the symbol $A (\text{m}^2)$ for area, Table 6.3 describes the mathematical relationship between flux and flow rate.

Table 6.3 Relationship Between Flux and Flow Rate Through an Area $A (\text{m}^2)$

Flux Type	Symbol	Flow Rate	Description
Mass Flux ($\text{g}/(\text{m}^2\text{s})$)	J	$\dot{m} = JA (\text{g/s})$	\dot{m} : Mass Flow Rate
Molar Flux ($\text{mol}/(\text{m}^2\text{s})$)	N	$\dot{n} = NA (\text{mol/s})$	\dot{n} : Molar Flow Rate
Heat Flux ($\text{J}/(\text{m}^2\text{s})$)	q	$\dot{Q} = qA (\text{J/s})$	\dot{Q} : Heat Transfer Rate

6.9.2 Flux Driving Forces

Various factors can be the driving forces for flux. Two important driving forces are gradients and convection. For the sake of simplicity this paper only considers gradient-driven flux.

Flux Caused by Gradients

Gradients describe a variation in space. For example, a temperature gradient in a material means that the temperature is different at different locations in the material. You have likely experienced flux due to a temperature gradient. If one part of a metal object is hotter than another, you have probably noticed that the heat tends to flow from the hotter to the colder parts of the metal. This heat flux is caused by a temperature gradient. The heat flux continues until the temperature is uniformly distributed and the temperature gradients are gone.

Conduction: Gradient-Based Heat Flux

The heat flux described in the example in the previous paragraph is heat conduction. Conduction is heat flux driven by a temperature gradient. One-dimensional heat conduction in the x-direction is described by the equation

$$q = -k \frac{dT}{dx} \quad (6.38)$$

where q ($\text{J}/(\text{m}^2\text{s})$) is the heat flux, the thermal conductivity k ($\text{J}/(\text{m s K})$) is a material property, T (K) is the temperature, and x (m) is the independent spatial variable.

Fick's 1st Law of Diffusion: Gradient-Based Mass Flux

You have likely experienced concentration gradient-based diffusion. On a calm spring day with no wind, you may have noticed that as you approach a flowering lilac plant that the strength of its aroma increases. The reason you can smell the lilac flowers when you are still a long distance away from the lilac plant is because of concentration-driven diffusion. The fact that the smell grows stronger the nearer you get to the plant indicates the gradient that is driving the diffusion of the scent.

If you have taken a course in materials science, you have likely studied Fick's laws of diffusion. Fick's 1st law states that binary diffusion flux is due to a concentration gradient. One dimensional binary diffusion in the x -direction is calculated by

$$N = -D \frac{dc}{dx} \quad (6.39)$$

In Fick's 1st law, N ($\text{mol}/(\text{m}^2\text{s})$) is the molar flux, and the diffusion coefficient D (m^2/s) is a property of the two substances, namely the diffusing substance and the host substance. c (mol/m^3) is the concentration of the diffusing substance. It is the number of moles per volume of the diffusing substance. It is related to the mass m (g) and volume V by the following equation

$$c = \frac{m}{VM} \quad (6.40)$$

where c (mol/m^3) is the molar concentration, V (m^3) is the volume, and M (g/mol) is the molar mass of the diffusing substance.

6.9.3 Conservation

Matter cannot be created or destroyed. It is conserved. Energy is not created or destroyed either. It is also conserved.

As a result of the laws of conservation, the change in the amount of mass contained in a volume is equal to the mass that enters the volume minus the mass that exits the volume. The change in the energy of a substance is equal to the energy transferred to it minus the energy transferred from it.

6.9.4 Conservation of Mass

In its rate form, the conservation of mass equation is

$$\frac{dm}{dt} = \dot{m}_{in} - \dot{m}_{out} \quad (6.41)$$

where m (g) is the mass inside a volume, t (s) is the independent time variable, \dot{m}_{in} (g/s) is the mass flow rate into the volume, and \dot{m}_{out} (g/s) is the mass flow rate out of the volume.

The conservation of mass equation can be written in terms of molar flux using the relationship from Table 6.3.

$$\frac{dm}{dt} = MA_{in}N_{in} - MA_{out}N_{out} \quad (6.42)$$

6.9 Modeling Dynamic Diffusion and Heat Transfer

where m (g) is the mass inside a volume, t (s) is the independent time variable, N_{in} (mol/(m²s)) is the molar flux into the volume through the area A_{in} , and N_{out} (mol/(m²s)) is the molar flux out of the volume through the area A_{out} . The molar mass M (g/mol) is a constant material property.

6.9.5 Conservation of Energy

In its rate form, the conservation of thermal energy of a closed system is

$$\frac{dU}{dt} = \dot{Q}_{in} - \dot{Q}_{out} \quad (6.43)$$

where U (J) is the internal energy of the substance, t (s) is the independent time variable, \dot{Q}_{in} (J/s or W) is the heat transfer rate to the substance, and \dot{Q}_{out} (J/s or W) is the rate at which heat is transferred from the substance.

The internal energy U is a function of absolute temperature T (K) of the substance, its mass m (kg), and its constant volume specific heat coefficient c_v (kJ/(kgK)). A differential change dU in the internal energy is equal to a differential change in the product of the mass m , specific heat c_v , and the temperature T ,

$$dU = d(mc_v T) \quad (6.44)$$

and if the mass m and specific heat c_v are constant:

$$dU = mc_v dT \quad (6.45)$$

Then Eq. (6.43) can be written as

$$mc_v \frac{dT}{dt} = \dot{Q}_{in} - \dot{Q}_{out} \quad (6.46)$$

or in terms of thermal flux (see Table 6.3):

$$mc_v \frac{dT}{dt} = q_{in} A_{in} - q_{out} A_{out} \quad (6.47)$$

6.9.6 Conservation and Flux in Finite Volumes

Eqs. (6.42) and (6.47) are the conservation laws in terms of flux. Gradient driven flux was described by Eqs. (6.39) and (6.38). Combining the conservation laws with the principles that describe flux results in partial differential equations. This section describes how. First, we start with the equations for molar flux and conservation of mass.

Consider again Fick's first law of diffusion Eq. (6.39). We will apply it to the series of finite volumes shown in Figure 6.62.

The concentration gradient $\frac{dc}{dx}$ can be approximated in terms of the concentrations of the finite volumes. If the concentration of volume i is c_i , then the concentration gradient between volumes $i - 1$ and i is approximately

$$\frac{dc}{dx} \approx \frac{c_i - c_{i-1}}{\Delta x} \quad (6.48)$$

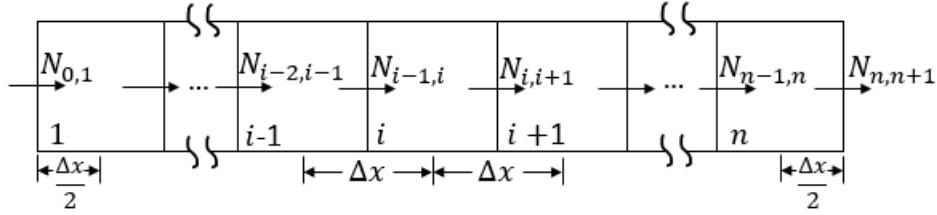


Figure 6.62 A series of finite volumes containing mass

This approximation becomes exact in the limit as the number of volumes approaches infinity, and the volume length Δx approaches zero.

Each volume has a length of Δx in the x-direction, and a cross-sectional area of A . Therefore, each volume has a value of

$$V = A\Delta x \quad (6.49)$$

Using the definition of concentration in Eq. (6.40), the approximation Eq. (6.48) becomes

$$\frac{dc}{dx} \approx \frac{m_i - m_{i-1}}{MA\Delta x^2} \quad (6.50)$$

and therefore the flux from volume $i - 1$ to volume i is (using Eq. (6.39))

$$N_{i-1,i} \approx -D \frac{m_i - m_{i-1}}{MA\Delta x^2} \quad (6.51)$$

The flux from volume i to $i + 1$ is similarly

$$N_{i,i+1} \approx -D \frac{m_{i+1} - m_i}{MA\Delta x^2} \quad (6.52)$$

Now, by applying the conservation of mass Eq. (6.42) to volume i

$$\frac{dm_i}{dt} \approx MA \left(-D \frac{m_i - m_{i-1}}{MA\Delta x^2} + D \frac{m_{i+1} - m_i}{MA\Delta x^2} \right) \quad (6.53)$$

which simplifies to

$$\frac{dm_i}{dt} \approx D \frac{m_{i-1} - 2m_i + m_{i+1}}{\Delta x^2} \quad (6.54)$$

In the limit as the number of volumes approaches infinity and $\Delta x \rightarrow 0$, and noting that mass is a function of both time and space:

$$\frac{\partial m}{\partial t} = D \frac{\partial^2 m}{\partial x^2} \quad (6.55)$$

or in terms of concentration instead of mass

$$\frac{\partial c}{\partial t} = D \frac{\partial^2 c}{\partial x^2} \quad (6.56)$$

6.9 Modeling Dynamic Diffusion and Heat Transfer

Eq. (6.56) is one form of Fick's 2nd law of diffusion. It is a partial differential equation that results from the combination of flux and the conservation of mass. Sometimes the diffusion coefficient D is not constant, and a more general form of Fick's second law is

$$\frac{\partial c}{\partial t} = -\frac{\partial N}{\partial x} = \frac{\partial}{\partial x} \left(D \frac{\partial c}{\partial x} \right) \quad (6.57)$$

Eq. (6.57) is the one-dimensional diffusion equation in the x-direction.

6.9.7 The Heat Equation

A similar analysis using a finite series of volumes can be used to derive the heat equation. The analysis replaces molar flux with heat flux in the finite volumes (see Figure 6.63).

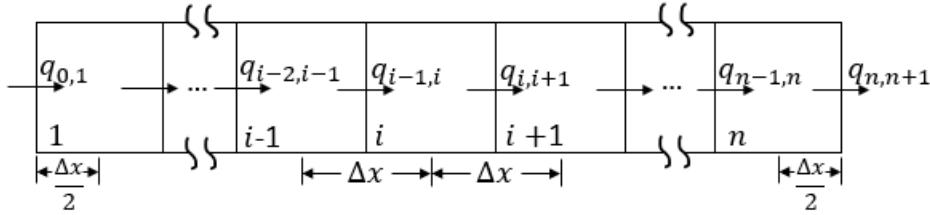


Figure 6.63 A series of finite volumes with heat flux between them

The temperature gradient between neighboring volumes $i - 1$ and i can be approximated by

$$\frac{dT}{dx} \approx \frac{T_i - T_{i-1}}{\Delta x} \quad (6.58)$$

and the heat flux approximation is (using Eq. (6.38))

$$q_{i-1,i} \approx -k \frac{T_i - T_{i-1}}{\Delta x} \quad (6.59)$$

Using these equations in the conservation of energy equation Eq. (6.47) for volume i results in

$$mc_v \frac{dT_i}{dt} \approx Ak \frac{T_{i-1} - 2T_i + T_{i+1}}{\Delta x} \quad (6.60)$$

We assume that the mass m of each volume is the same, and that the density ρ (kg/m^3) of each volume is also uniform. Each volume has a value of $V = A\Delta x$. The mass can be written in terms of the density ρ , area A , and volume length Δx .

$$m = \rho V = \rho A \Delta x \quad (6.61)$$

Substituting Eq. (6.61) into Eq. (6.60) and dividing both sides by $A\Delta x$ produces the finite volume approximation of the heat equation in one-dimension.

$$\rho c_v \frac{dT_i}{dt} \approx k \frac{T_{i-1} - 2T_i + T_{i+1}}{\Delta x^2} \quad (6.62)$$

In the limit as $\Delta x \rightarrow 0$ and the number of volumes approaches infinity, Eq. (6.62) becomes

$$\rho c_v \frac{\partial T}{\partial t} = k \frac{\partial^2 T}{\partial x^2} \quad (6.63)$$

Eq. (6.63) is the one-dimensional heat equation in the x-direction.

The following sections describe a finite-volume numerical solution in the form $\dot{x} = Ax + Bu$ to the diffusion equation Eq. (6.56) and the heat equation Eq. (6.63).

6.9.8 The Finite Volume Method

The finite volume method for numerically solving the diffusion equation Eq. (6.56) was actually already derived in Section 6.9. It was part of the derivation of the diffusion equation itself. It used the series of finite volumes shown first in Figure 6.62 and repeated here in Figure 6.64.

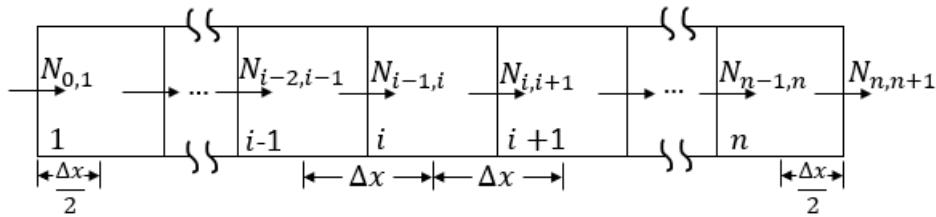


Figure 6.64 A series of finite volumes containing mass

Eq. (6.54) is the finite volume solution to the diffusion equation. It is repeated here:

$$\boxed{\frac{dm_i}{dt} \approx D \frac{m_{i-1} - 2m_i + m_{i+1}}{\Delta x^2}} \quad (6.64)$$

Following the derivation, if needed, we can replace each mass m_i with the corresponding concentration c_i .

$$\boxed{\frac{dc_i}{dt} \approx D \frac{c_{i-1} - 2c_i + c_{i+1}}{\Delta x^2}} \quad (6.65)$$

Eq. (6.62) is the finite volume solution to the heat equation. It is repeated here:

$$\boxed{\rho c_v \frac{dT_i}{dt} \approx k \frac{T_{i-1} - 2T_i + T_{i+1}}{\Delta x^2}} \quad (6.66)$$

6.9.9 Boundary Conditions

Eq. (6.64) through Eq. (6.66) apply to any interior volume in Figure 6.64 and 6.63, but do not apply to the volumes 1 and n on the edges. The equations for the edge volumes depend on the conditions at the boundaries. There are two types of boundary conditions that we will consider: (1) flux and (2) value. The flux type is also known as a “Neumann” boundary condition. The value type is called a “Dirichlet” boundary condition.

6.9 Modeling Dynamic Diffusion and Heat Transfer

6.9.10 Neumann Boundary Conditions

With a Neumann boundary condition, the flux into or out of the edge volume is known. For example, if the boundary condition for the 1st volume of Figure 6.64 is a Neumann type, then $N_{0,1}$ is known. Therefore, the mass balance for volume 1 is

$$\frac{dm_1}{dt} \approx MAN_{0,1} + D \frac{m_2 - m_1}{\Delta x^2} \quad (6.67)$$

where $MAN_{0,1}$ (g/s) is the mass flow rate through the boundary into volume 1.

The heat equation can also have Neumann type boundary conditions. For example, if the boundary condition for volume 1 in Figure 6.63 is a Neumann type, then the heat flux $q_{0,1}$ into volume 1 is known. Therefore, the conservation of energy equation for volume 1 is (using 6.47)

$$\rho c_v \frac{dT_1}{dt} \approx \frac{q_{0,1}}{\Delta x} + k \frac{T_2 - T_1}{\Delta x^2} \quad (6.68)$$

6.9.11 Dirichlet Boundary Conditions

With Dirichlet boundary conditions, the value at the boundary of the edge volume is known. For example, if the boundary condition for the last volume of Figure 6.64 is a Dirichlet type, then the concentration at the boundary, c_{n+1} , is known. There is still diffusion flux between the boundary and the center of volume n , but the diffusion distance is $\frac{\Delta x}{2}$ instead of Δx . By defining $m_{n+1} = MA\Delta x c_{n+1}$, the mass balance for volume n is therefore (using Eq. (6.42))

$$\frac{dm_n}{dt} \approx D \frac{m_{n-1} - 3m_n + 2m_{n+1}}{\Delta x^2} \quad (6.69)$$

which is only slightly different than the equation for the interior volumes Eq. (6.64).

The heat equation can also have Dirichlet type boundary conditions. For example, if the boundary condition for volume n in Figure 6.63 is a Dirichlet type, then the temperature T_{n+1} is known. There is still conduction flux between the boundary and the center of volume n , but the distance is $\frac{\Delta x}{2}$ instead of Δx . Therefore, the conservation of energy equation for volume n is (using 6.47)

$$\rho c_v \frac{dT_n}{dt} \approx k \frac{T_{n-1} - 3T_n + 2T_{n+1}}{\Delta x^2} \quad (6.70)$$

which is only slightly different than the equation for the interior volumes Eq. (6.66).

6.9.12 Diffusion Equation Solution by $\dot{x} = Ax + Bu$

If we have a Neumann (flux) boundary condition for the first volume and a Dirichlet boundary condition for the last volume, then we can summarize the finite volume solution to the diffusion equation with Eq. (6.64), Eq. (6.67), and Eq. (6.69). We will choose the state variables to be the masses m_1, m_2, \dots, m_n of the n volumes and the inputs to be $N_{0,1}$ and c_{n+1} . Then the finite volume method can be described by the state-space equation

$$\dot{x} = Ax + Bu \quad (6.71)$$

where the state vector x is

$$x = \begin{bmatrix} m_1 \\ m_2 \\ \vdots \\ m_{n-1} \\ m_n \end{bmatrix} \quad (6.72)$$

The input vector u is

$$u = \begin{bmatrix} N_{0,1} \\ c_{n+1} \end{bmatrix} \quad (6.73)$$

The A matrix is

$$A = \begin{bmatrix} -\frac{D}{\Delta x^2} & \frac{D}{\Delta x^2} & 0 & 0 & \cdots & 0 & 0 & 0 & 0 \\ \frac{D}{\Delta x^2} & -\frac{2D}{\Delta x^2} & \frac{D}{\Delta x^2} & 0 & \cdots & 0 & 0 & 0 & 0 \\ 0 & \frac{D}{\Delta x^2} & -\frac{2D}{\Delta x^2} & \frac{D}{\Delta x^2} & \ddots & 0 & 0 & 0 & 0 \\ \vdots & \ddots & \vdots \\ 0 & \ddots & 0 & \frac{D}{\Delta x^2} & -\frac{2D}{\Delta x^2} & \frac{D}{\Delta x^2} & 0 & \ddots & 0 \\ \vdots & \ddots & \vdots \\ 0 & 0 & 0 & 0 & \cdots & \frac{D}{\Delta x^2} & -\frac{2D}{\Delta x^2} & \frac{D}{\Delta x^2} & 0 \\ 0 & 0 & 0 & 0 & \cdots & 0 & \frac{D}{\Delta x^2} & -\frac{2D}{\Delta x^2} & \frac{D}{\Delta x^2} \\ 0 & 0 & 0 & 0 & \cdots & 0 & 0 & \frac{D}{\Delta x^2} & -\frac{3D}{\Delta x^2} \end{bmatrix} \quad (6.74)$$

and the B matrix is

$$B = \begin{bmatrix} MA & 0 \\ 0 & 0 \\ \vdots & \vdots \\ 0 & 0 \\ 0 & \frac{2MAD}{\Delta x} \end{bmatrix} \quad (6.75)$$

6.9.13 Heat Equation Solution by $\dot{x} = Ax + Bu$

The finite volume solution to the heat equation is similar to the diffusion equation solution. It is solved using $\dot{x} = Ax + Bu$.

If we have a Neumann (flux) boundary condition for the first volume and a Dirichlet boundary condition for the last volume, then we can summarize the finite volume solution to the heat equation with Eq. (6.66), Eq. (6.68), and Eq. (6.70). We will choose the state variables to be the temperatures T_1, T_2, \dots, T_n of the n volumes. The inputs are the heat flux into the first volume $q_{0,1}$ and the temperature T_{n+1} at the boundary of the last volume n . The state vector x is therefore

6.9 Modeling Dynamic Diffusion and Heat Transfer

$$x = \begin{bmatrix} T_1 \\ T_2 \\ \vdots \\ T_{n-1} \\ T_n \end{bmatrix} \quad (6.76)$$

The input vector u is

$$u = \begin{bmatrix} q_{0,1} \\ T_{n+1} \end{bmatrix} \quad (6.77)$$

The A matrix is

$$A = \left[\begin{array}{ccccccccc} \frac{-k}{\rho c_v \Delta x^2} & \frac{k}{\rho c_v \Delta x^2} & 0 & 0 & \cdots & 0 & 0 & 0 & 0 \\ \frac{k}{\rho c_v \Delta x^2} & \frac{-2k}{\rho c_v \Delta x^2} & \frac{k}{\rho c_v \Delta x^2} & 0 & \cdots & 0 & 0 & 0 & 0 \\ 0 & \frac{k}{\rho c_v \Delta x^2} & \frac{-2k}{\rho c_v \Delta x^2} & \frac{k}{\rho c_v \Delta x^2} & \ddots & 0 & 0 & 0 & 0 \\ \vdots & \ddots & \vdots \\ 0 & \ddots & 0 & \frac{k}{\rho c_v \Delta x^2} & \frac{-2k}{\rho c_v \Delta x^2} & \frac{k}{\rho c_v \Delta x^2} & 0 & \ddots & 0 \\ \vdots & \ddots & \vdots \\ 0 & 0 & 0 & 0 & \cdots & \frac{k}{\rho c_v \Delta x^2} & \frac{-2k}{\rho c_v \Delta x^2} & \frac{k}{\rho c_v \Delta x^2} & 0 \\ 0 & 0 & 0 & 0 & \cdots & 0 & \frac{k}{\rho c_v \Delta x^2} & \frac{-2k}{\rho c_v \Delta x^2} & \frac{k}{\rho c_v \Delta x^2} \\ 0 & 0 & 0 & 0 & \cdots & 0 & 0 & \frac{k}{\rho c_v \Delta x^2} & \frac{-3k}{\rho c_v \Delta x^2} \end{array} \right] \quad (6.78)$$

and the B matrix is

$$B = \begin{bmatrix} \frac{1}{\rho c_v \Delta x} & 0 \\ 0 & 0 \\ \vdots & \vdots \\ 0 & 0 \\ 0 & \frac{2k}{\rho c_v \Delta x^2} \end{bmatrix} \quad (6.79)$$

6.9.14 Example: Carburization

The following sections apply the methods from previous sections to model the carburization process for case-hardening the surface of a steel gear. Case-hardening by carburization is the process of diffusing carbon into a metal surface to harden it. The hardened surface is more wear resistant, enabling the gear to last much longer than if it had not been hardened. This process is described in more detail in materials science textbooks¹.

¹ see Chapter 6: Diffusion of W. D. Callister Jr., D. G. Rethwisch, Fundamentals of Materials Science and Engineering: an Integrated Approach, 5th Ed., pp. 186-215, 2015

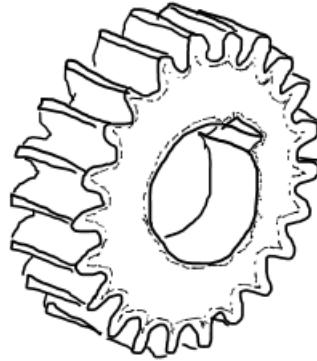


Figure 6.65 A gear that has been case-hardened. The dotted line near the surface shows the depth of the diffusion of the hardened surface.

6.9.15 Problem Statement

Consider a steel gear with an initial carbon content of 0.2 wt% carbon. The gear is heat-treated to austenite in a carburizing atmosphere at 1100°C (1373 K) which causes the surface carbon content to saturate at around 2 wt% carbon. The surface content is held at 2 wt% carbon for long enough that the carbon content reaches 1 wt% carbon at a depth of 0.1 mm beneath the surface, at which time the carburization process is stopped and the gear is quenched to room temperature. The goal is to determine the time it takes to complete the carburization process and to show the evolution of the carbon content within the gear during the process. It will use the finite volume solution discussed in Section 6.9.8 to do so.

The diffusion coefficient D (m^2/s) for carbon into steel depends on the temperature. It is calculated by the equation

$$D = D_0 \exp\left(\frac{-Q_d}{R_u T}\right) \quad (6.80)$$

where T (K) is the absolute temperature, $Q_d = 148000 \text{ J/mol}$ is the activation energy, $D_0 = 2.3 \times 10^{-5} \text{ m}^2/\text{s}$ is the preexponential coefficient, and $R_u = 8.314 \text{ J/(mol K)}$ is the universal ideal gas constant. With these values, the diffusion coefficient was calculated to be $D = 5.38 \times 10^{-11} \text{ m}^2/\text{s}$.

6.9.16 Finite Volume Solution

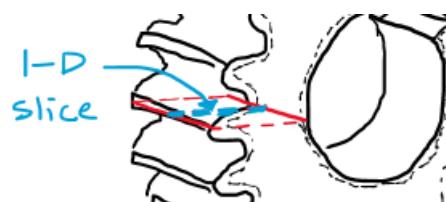


Figure 6.66 A one-dimensional slice from the gear

6.9 Modeling Dynamic Diffusion and Heat Transfer

This section uses the finite volume method to solve the carburization problem. It will take a one-dimensional slice (see Figure 6.66) from the gear and apply the equations derived in Section 6.9.8 to model the diffusion of carbon into the gear. The one dimensional slice is 1 cm long.

Because the length of the one-dimensional slice is so much greater than the diffusion depth of interest (0.1 mm), we will only use a fraction of the one-dimensional slice. We will only consider the first 1 mm of depth beneath the surface. During the diffusion process, we will assume that the carbon content at 1 mm depth is unaffected by the process, and so the flux of carbon diffusing out of it is zero. Therefore, the boundary is at 1 mm depth, and the boundary condition is $N_{n,n+1} = 0 \text{ mol}/(\text{m}^2\text{s})$. The other boundary is at the surface of the gear where the carbon content remains a constant 2 wt% carbon.

To use the finite volume solution, we will need to convert carbon content to carbon concentration. The following formula converts from weight percent (wt%) to concentration (mol/m³):

$$c = \frac{\frac{W_1}{M_1}}{\frac{W_1}{\rho_1} + \frac{100\% - W_1}{\rho_2}} \times 10^3 \quad (6.81)$$

where W_1 is the carbon content (wt%), $\rho_1 = 2.25 \text{ (g/cm}^3)$ is the density of carbon, $\rho_2 = 7.87 \text{ (g/cm}^3)$ is the density of iron, and $M_1 = 12.011 \text{ g/mol}$ is the molar mass of carbon.

With this formula, a 0.2 wt% carbon mild steel would have a carbon concentration of 1.304 mol/m³. A 1 wt% carbon content corresponds to a carbon concentration of 6.39 mol/m³. A 2 wt% carbon content corresponds to a carbon concentration of 12.48 mol/m³, therefore the surface boundary condition is $c_0 = 12.48 \text{ mol/m}^3$.

To use the finite volume method, we need to divide the one-dimensional slice into n volumes. We will make each volume $\Delta x = 0.01 \text{ mm}$ wide, and therefore $n = 100$.

We must formulate the state-equations $\dot{x} = Ax + Bu$ that govern the diffusion process. The input u to the carburization process is the boundary conditions: the surface concentration c_0 and the molar flux $N_{n,n+1}$ out of volume n .

$$u = \begin{bmatrix} c_0 \\ N_{n,n+1} \end{bmatrix} \quad (6.82)$$

The state variables are the carbon concentrations of each volume:

$$x = \begin{bmatrix} c_1 \\ c_2 \\ \vdots \\ c_{n-1} \\ c_n \end{bmatrix} \quad (6.83)$$

Initially the carbon concentration in each volume is 1.304 mol/m³, and so the initial condition is

$$x_0 = \begin{bmatrix} 1.304 \\ 1.304 \\ \vdots \\ 1.304 \\ 1.304 \end{bmatrix} \quad (6.84)$$

Because the first boundary is a Dirichlet type and the n^{th} boundary is a Neumann type, the A matrix is

$$A = \begin{bmatrix} \frac{-3D}{\Delta x^2} & \frac{D}{\Delta x^2} & 0 & 0 & \cdots & 0 & 0 & 0 & 0 \\ \frac{D}{\Delta x^2} & \frac{-2D}{\Delta x^2} & \frac{D}{\Delta x^2} & 0 & \cdots & 0 & 0 & 0 & 0 \\ 0 & \frac{D}{\Delta x^2} & \frac{-2D}{\Delta x^2} & \frac{D}{\Delta x^2} & \ddots & 0 & 0 & 0 & 0 \\ \vdots & \ddots & \vdots \\ 0 & \ddots & 0 & \frac{D}{\Delta x^2} & \frac{-2D}{\Delta x^2} & \frac{D}{\Delta x^2} & 0 & \ddots & 0 \\ \vdots & \ddots & \vdots \\ 0 & 0 & 0 & 0 & \cdots & \frac{D}{\Delta x^2} & \frac{-2D}{\Delta x^2} & \frac{D}{\Delta x^2} & 0 \\ 0 & 0 & 0 & 0 & \cdots & 0 & \frac{D}{\Delta x^2} & \frac{-2D}{\Delta x^2} & \frac{D}{\Delta x^2} \\ 0 & 0 & 0 & 0 & \cdots & 0 & 0 & \frac{D}{\Delta x^2} & \frac{-D}{\Delta x^2} \end{bmatrix} \quad (6.85)$$

and the B matrix is

$$B = \begin{bmatrix} \frac{2D}{\Delta x^2} & 0 \\ 0 & 0 \\ \vdots & \vdots \\ 0 & 0 \\ 0 & \frac{-1}{\Delta x} \end{bmatrix} \quad (6.86)$$

6.9.17 MATLAB Code

The following MATLAB code implements the finite volume solution to the carburization process.

```

close all
clear
clc

%Set the simulation time step and time parameters
dt = 1; %(s) time step
t = 0:dt:159; %time vector
N = length(t); %length of the time vector

%set some material constants
M_carbon = 12.011; %(g/mol) carbon molar mass
rho_iron = 7.87; %(g/cm3) density of iron

```

6.9 Modeling Dynamic Diffusion and Heat Transfer

```
rho_carbon = 2.25; % (g/cm3) density of carbon

% Set up the state space matrices
dx = 0.01e-3; %(mm) length of each volume
n = 100; % number of finite volumes
D = 5.38e-11; %(m2/s) diffusion coefficient
% get the A matrix
A = zeros(n,n);
for ii=1:n
    if ii == 1
        % boundary condition for first volume
        A(1,1) = -3*D/dx^2;
        A(1,2) = D/dx^2;
    elseif ii == n
        % boundary condition for last volume
        A(n,n-1) = D/dx^2;
        A(n,n) = -D/dx^2;
    else
        % interior volumes
        A(ii,ii-1) = D/dx^2;
        A(ii,ii) = -2*D/dx^2;
        A(ii,ii+1) = D/dx^2;
    end
end
% get the B matrix
B = zeros(n,2);
B(1,1) = 2*D/dx^2;
B(n,2) = -1/dx;
% get the discrete Ad and Bd matrices
[Ad,Bd] = StateSpaceSolution(A,B,dt);

% set up the input
%(mol/m3) surface carbon concentration
c_0 = Content2Concentration(2, M_carbon, rho_carbon, rho_iron);
N_n_1 = 0; %(mol/(m2 s)) molar flux out of the nth volume
u = [ones(1,N)*c_0; ones(1,N)*N_n_1]; % Input for dx=Ax+Bu

% initial conditions
%(mol/m3) initial carbon concentrations
x = ones(n,1)*Content2Concentration(0.2, M_carbon, rho_carbon, rho_iron);

% run and plot the simulated carburization process
figure(1), hold on
title('Carburization of a gear')
xlabel('Depth Beneath Surface (mm)')
ylabel('Carbon Content (wt%)')
plot((dx*(0:n-1)+dx/2)*1000, ...
x*100./((rho_iron*(1000/M_carbon+x/rho_iron-x/rho_carbon))))
for ii = 1:N
    if mod(ii,30) == 0
        % draw a new curve every 30 seconds
        plot((dx*(0:n-1)+dx/2)*1000, ...
        x*100./((rho_iron*(1000/M_carbon+x/rho_iron-x/rho_carbon))))
        drawnow
    end

```

```

x = Ad*x+Bd*u(:,ii);
end
plot((dx*(0:n-1)+dx/2)*1000, ...
x*100./((rho_iron*(1000/M_carbon+x/rho_iron-x/rho_carbon)))
plot([0,0.1],[1,1], 'k:', [0.1,0.1],[0,1], 'k:')
legend('t=0 s','t=30 s','t=60 s','t=90 s',...
't=120 s','t=150 s','t=159 s','Desired')
hold off

%% define the StateSpaceSolutions function
function [Ad, Bd] = StateSpaceSolution(A,B,dt)
% [Ad, Bd] = StateSpaceSolution(A,B,dt)
% Calculate the discrete-time matrices for the solution to
% the state-space equation dx = Ax+Bu
% This code even works when A is singular (not invertible)
% The A matrix is nxn
% The B matrix is nxm

n = length(A);
m = length(B(1,:));
Fd = expm([A*dt,B*dt,zeros(m,n+m)]);
Ad = Fd(1:n,1:n);
Bd = Fd(1:n,n+1:n+m);
end

%% define the function to convert carbon content to carbon concentration
function c = Content2Concentration(W1, M1, rho1, rho2)
%W1 weight percent
%M1 (g/mol) molar mass
%rho1 (g/cm3) density
%rho2 (g/cm3) density
%c (mol/m3) concentration
c = W1/M1/(W1/rho1+(100-W1)/rho2)*1000;
end

```

6.9.18 Simulation Results

Figure 6.67 shows the MATLAB simulation results using the finite volume method. The results show that after 159 s in the carburizing atmosphere, the carbon content reaches the desired level of 1% carbon content at a depth of 0.1 mm beneath the surface of the gear. The next section will show that this agrees with the known analytical solution to this problem. Figure 6.67 also shows a snapshot of the carbon distribution within the gear for each 30 s of the carburization process.

6.9.19 Analytical Solution

One of the purposes for choosing this carburization example is because it has a known analytical solution. We will compare the results of the numerical solutions of this book with the analytical solution.

The analytical solution is

$$\frac{W_{x,t} - W_0}{W_s - W_0} = 1 - \operatorname{erf}\left(\frac{x}{2\sqrt{Dt}}\right) \quad (6.87)$$

6.9 Modeling Dynamic Diffusion and Heat Transfer

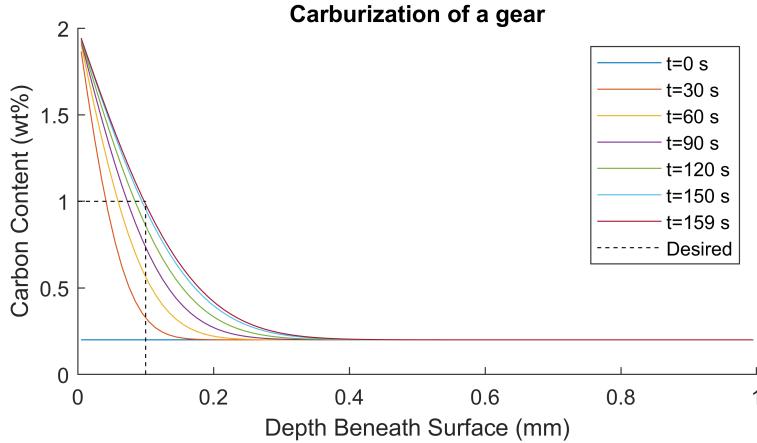


Figure 6.67 Finite volume solution to the carburization problem

where $W_{x,t}$ (wt%) is the carbon content at a depth of x (m) below the gear surface, W_0 (wt%) is the initial carbon content of the gear, W_s (wt%) is the surface carbon content during carburization, D (m^2/s) is the diffusion coefficient, and t (s) is the time duration of the process. The Gaussian error function $\text{erf}(z)$ is

$$\text{erf}(z) = \frac{2}{\sqrt{\pi}} \int_0^z \exp(-y^2) dy \quad (6.88)$$

Solving Eq. (6.87) for time t gives the analytical solution

$$t = \frac{1}{D} \left(\frac{x}{2 \text{erf}^{-1} \left(1 - \frac{W_{x,t} - W_0}{W_s - W_0} \right)} \right)^2 \quad (6.89)$$

where the inverse error function $\text{erf}^{-1}(z)$ can be calculated in MATLAB using the “erfinv” function.

Plugging in the values from the problem statement

$$t = \frac{1}{5.38 \times 10^{-11}} \left(\frac{0.0001}{2 \text{erf}^{-1} \left(1 - \frac{1-0.2}{2-0.2} \right)} \right)^2 = 159 \text{ s} \quad (6.90)$$

The time required to complete the carburization process is 159 s. This agrees with the finite volume solution of Figure 6.67.

Chapter 7

Modeling Flight Dynamics: North-East-Down

Contents

7.1	Equations for North-East-Down (NED) Coordinates	228
7.1.1	Airplane Parameters for NED	228
7.1.2	Fixed-Wing Aircraft Equations for NED Coordinate System	230
7.2	Derivation of the Airplane Equations of Motion	233
7.2.1	Quaternion Rotations	233
7.2.2	Properties of Orthonormal Rotation Matrices	234
7.2.3	Angular Velocity and Quaternions	235
7.2.4	Coordinate Frame Translations	235
7.3	Equations of Motion	235
7.3.1	General 6 DOF Motion of a Rigid Body	235
7.3.2	State-Equations for 6 DOF Rigid Body Motion	237
7.3.3	Equations of Motion for an Aircraft	239
7.4	Airspeed, Angle of Attack, and Side-slip Angle	241
7.5	Wind Model	241
7.6	Input Commands	243
7.7	Modeling Propellers	244
7.7.1	Propeller Thrust	244
7.7.2	Propeller Torque	246
7.7.3	Motor Equations	247
7.7.4	Computational Simplifications for Propeller Speed	248
7.7.5	Summary of Propeller Equations	248
7.8	Gravitational Forces	250
7.9	Aerodynamics of Fixed-Wing Aircraft	250
7.9.1	Lift, Drag, and Side-slip Forces	250
7.9.2	Aerodynamic Torques	251

7.1 Equations for North-East-Down (NED) Coordinates

Because of the importance of the North-East-Down (NED) coordinate system in aerospace applications, this chapter provides a summary of the aircraft equations in the NED inertial reference frame. It also provides the parameters for the different airplane models. In the NED inertial frame, the x-axis points towards the north, the y-axis points to the east, and the z-axis points down towards the center of the earth. The body-fixed coordinate system is also different. The body-fixed x-axis points out the nose of the aircraft, the y-axis points along the wing to the right, and the z-axis points down through the bottom of the aircraft as shown in Figure 7.1.

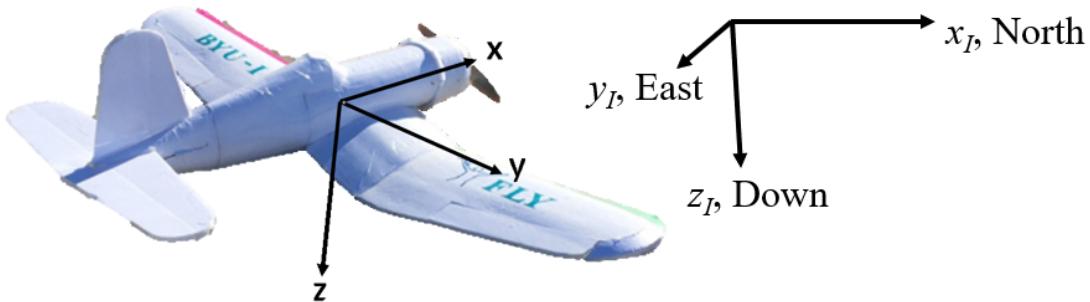


Figure 7.1 The traditional body-fixed coordinate system in the NED inertial frame.

7.1.1 Airplane Parameters for NED

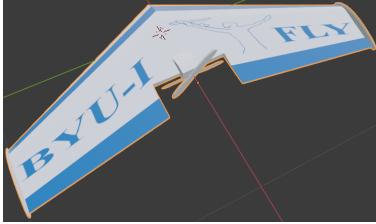
Table 7.1 lists general constant parameters, and Table 7.2 lists constant parameters for the BYU-I Wing and FT Corsair airplanes in the NED coordinate frame.

Table 7.1 General Parameters for North-East-Down (NED) Coordinates

Parameter	Value	Description
ρ	1.2682 kg/m ³	Air density
g	9.81 m/s ²	Gravitational Acceleration

7.1 Equations for North-East-Down (NED) Coordinates

Table 7.2 Constant Parameters for Airplane Simulations

		
	BYU-I Wing	FT Corsair
m	0.9 kg	1 kg
J_{xx}	0.115 kg m ²	0.12 kg m ²
J_{yy}	0.16 kg m ²	0.2 kg m ²
J_{zz}	0.17 kg m ²	0.18 kg m ²
J_{xy}	0 kg m ²	0 kg m ²
J_{xz}	0.0015 kg m ²	0.015 kg m ²
J_{yz}	0 kg m ²	0 kg m ²
S	1.42 m	1.5 m
c	0.33 m	0.3 m
A	0.47 m ²	0.45 m ²
$C_{L,0}$	-0.9	-0.9
$C_{L,1}$	2.5	2.5
$C_{L,2}$	1	1
$C_{L,3}$	11	11
$C_{L,4}$	0.004	0.004
C_{L,δ_e}	0.1	0.1
$C_{D,0}$	0.06	0.06
$C_{D,2}$	0.44	0.44
C_{D,δ_e}	0.001	0.001
$C_{y,\beta}$	0.01	0.015
C_{y,δ_r}	0	0.001
$C_{Ty,\alpha}$	0.15	0.1
C_{Ty,δ_e}	0.07	0.1
b_q	0.4 N m s	0.5 N m s
C_{δ_a}	0.02	0.03
b_p	1 N m s	1 N m s
$C_{Tz,\beta}$	0.005	0.0002
C_{Tz,δ_r}	0	0.04

Continued on next page

Table 7.2 – *Continued from previous page*

	BYU-I Wing	FT Corsair
b_r	0.5 N m s	0.5 N m s
n_{\max}	170 rev/s	170 rev/s
D	0.2286 m	0.254 m
α_b	5 inch	5 inch

7.1.2 Fixed-Wing Aircraft Equations for NED Coordinate System

This section lists the equations for the fixed-wing aircraft motion in the North-East-Down (NED) coordinate frame. The equations for the North-Up-East coordinate system were derived in earlier sections.

Airspeed, Angle of Attack, and Side-slip Angle

$$\begin{bmatrix} u_r \\ v_r \\ w_r \end{bmatrix} = \begin{bmatrix} u - u_w \\ v - v_w \\ w - w_w \end{bmatrix}$$

$$V_a = \sqrt{u_r^2 + v_r^2 + w_r^2}$$

$$\alpha = \tan^{-1}\left(\frac{w_r}{V_a}\right)$$

$$\beta = \sin^{-1}\left(\frac{v_r}{V_a}\right)$$

Wind Model

$$\dot{u}_{w_g} = -\frac{V_a}{L_u} u_{w_g} + \sigma_u \sqrt{\frac{2V_a}{\pi L_u}} \mathcal{N}_1$$

$$\begin{bmatrix} \dot{x}_1 \\ \dot{x}_2 \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ -\left(\frac{V_a}{L_v}\right)^2 & -2\frac{V_a}{L_v} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} + \begin{bmatrix} 0 \\ \sigma_v \sqrt{\frac{3V_a}{\pi L_v}} \end{bmatrix} \mathcal{N}_2$$

$$v_{w_g} = \begin{bmatrix} \frac{V_a}{\sqrt{3L_v}} & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$$

$$\begin{bmatrix} \dot{x}_3 \\ \dot{x}_4 \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ -\left(\frac{V_a}{L_w}\right)^2 & -2\frac{V_a}{L_w} \end{bmatrix} \begin{bmatrix} x_3 \\ x_4 \end{bmatrix} + \begin{bmatrix} 0 \\ \sigma_w \sqrt{\frac{3V_a}{\pi L_w}} \end{bmatrix} \mathcal{N}_3$$

$$w_{w_g} = \begin{bmatrix} \frac{V_a}{\sqrt{3L_w}} & 1 \end{bmatrix} \begin{bmatrix} x_3 \\ x_4 \end{bmatrix}$$

7.1 Equations for North-East-Down (NED) Coordinates

Gust Parameters	$L_u = L_v$ (m)	L_w (m)	$\sigma_u = \sigma_v$ (m/s)	σ_w (m/s)
≈ 50 m altitude, light	200	50	1.06	0.7
≈ 50 m altitude, moderate	200	50	2.12	1.4
≈ 600 m altitude, light	533	533	1.5	1.5
≈ 600 m altitude, moderate	533	533	3.0	3.0

$$\begin{bmatrix} u_w \\ v_w \\ w_w \end{bmatrix} = R_I^b \begin{bmatrix} w_{xI} \\ w_{yI} \\ w_{zI} \end{bmatrix} + \begin{bmatrix} u_{w_g} \\ v_{w_g} \\ w_{w_g} \end{bmatrix}$$

where R_I^b is the rotation matrix from the inertial to the body frame:

$$R_I^b = \begin{bmatrix} e_0^2 + e_1^2 - e_2^2 - e_3^2 & 2(e_0e_3 + e_1e_2) & 2(e_1e_3 - e_0e_2) \\ 2(e_1e_2 - e_0e_3) & e_0^2 - e_1^2 + e_2^2 - e_3^2 & 2(e_0e_1 + e_2e_3) \\ 2(e_0e_2 + e_1e_3) & 2(e_2e_3 - e_0e_1) & e_0^2 - e_1^2 - e_2^2 + e_3^2 \end{bmatrix}$$

Gravitational Forces

$$\begin{bmatrix} f_{x,g} \\ f_{y,g} \\ f_{z,g} \end{bmatrix} = mg \begin{bmatrix} 2(e_1e_3 - e_0e_2) \\ 2(e_0e_1 + e_2e_3) \\ e_0^2 - e_1^2 - e_2^2 + e_3^2 \end{bmatrix} \quad (7.1)$$

Propeller Torque and Thrust

$$n = n_{\max} (1 - (1 - \delta_t)^2)$$

Using the value of the relative speed u_r in the advance ratio $J = \frac{u_r}{nD}$, calculate the torque coefficient.

$$C_Q = z_Q C_{Q,L} + (1 - z_Q) C_{Q,u}$$

where

$$z_Q = \frac{1}{1 + e^{10(0.16 + 0.05\alpha_b - J)}}$$

$$C_{Q,L} = 0.0121 - 0.017(J - 0.1(\alpha_b - 5))$$

and

$$C_{Q,u} = 0.0057 + 0.00125(\alpha_b - 5) - 0.0005J$$

Calculate the propeller torque.

$$T_{x,Q} = \rho n^2 D^5 C_Q$$

Calculate the thrust coefficient C_T .

$$C_T = z C_{T,L} + (1 - z) C_{T,u}$$

where

$$z = \frac{1}{1 + e^{20(0.14 + 0.018\alpha_b - 0.6J)}}$$

$$C_{T,L} = 0.11 - 0.18(J - 0.105(\alpha_b - 5))$$

and

$$C_{T,u} = 0.1 + 0.009(\alpha_b - 5) - 0.065J$$

and α_b has units of inches. Finally, calculate the propeller thrust.

$$f_{x,T} = \rho n^2 D^4 C_T$$

Forces and Torques from Aerodynamics

The aerodynamic force is

$$f_{\text{aero}} = \frac{1}{2} \rho V_a^2 A$$

where A is the wing area (m^2), and ρ (kg/m^3) is the air density.

The lift force in the body-fixed z-direction is

$$f_{z,L} = f_{\text{aero}} \left(- \left(C_{L,0} + \frac{C_{L,1}}{C_{L,2} + e^{-C_{L,3}(\alpha + C_{L,4})}} \right) - C_{L,\delta_e} \delta_e \right)$$

The drag force in the body-fixed x-direction is

$$f_{x,D} = f_{\text{aero}} (- (C_{D,0} + C_{D,2}\alpha^2) - C_{D,\delta_e} \delta_e)$$

The side-slip force in the y-direction is

$$f_{y,\beta} = f_{\text{aero}} (-C_{y,\beta}\beta - C_{y,\delta_r} \delta_r)$$

The pitching torque about the y-axis is

$$T_y = f_{\text{aero}} c (-C_{Ty,\alpha}\alpha - C_{Ty,\delta_e} \delta_e) - b_q q$$

where c is the wing chord. The roll torque about the x-axis is

$$T_{x,\text{aero}} = f_{\text{aero}} S C_{\delta_a} \delta_a - b_p p$$

where S is the wing span. The yaw torque about the z-axis is

$$T_z = f_{\text{aero}} c (C_{Tz,\beta}\beta + C_{Tz,\delta_r} \delta_r) - b_r r$$

Combined Forces and Torques in the Body-Fixed Directions

$$f_x = f_{x,T} + f_{x,D} + f_{x,g}$$

$$f_y = f_{y,\beta} + f_{y,g}$$

$$f_z = f_{z,L} + f_{z,g}$$

$$T_x = T_{x,\text{aero}} + T_{x,Q}$$

7.2 Derivation of the Airplane Equations of Motion

Equations of motion for an aircraft:

$$\begin{aligned} \begin{bmatrix} \dot{u} \\ \dot{v} \\ \dot{w} \end{bmatrix} &= \begin{bmatrix} rv - qw \\ pw - ru \\ qu - pv \end{bmatrix} + \frac{1}{m} \begin{bmatrix} f_x \\ f_y \\ f_z \end{bmatrix} \\ \begin{bmatrix} \dot{p} \\ \dot{q} \\ \dot{r} \end{bmatrix} &= J^{-1} \left(\begin{bmatrix} qr(J_{yy} - J_{zz}) + pqJ_{xz} \\ pr(J_{zz} - J_{xx}) + (r^2 - p^2)J_{xz} \\ pq(J_{xx} - J_{yy}) - qrJ_{xz} \end{bmatrix} + \begin{bmatrix} T_x \\ T_y \\ T_z \end{bmatrix} \right) \\ \begin{bmatrix} \dot{x}_I \\ \dot{y}_I \\ \dot{z}_I \end{bmatrix} &= \begin{bmatrix} e_0^2 + e_1^2 - e_2^2 - e_3^2 & 2(e_1e_2 - e_0e_3) & 2(e_0e_2 + e_1e_3) \\ 2(e_0e_3 + e_1e_2) & e_0^2 - e_1^2 + e_2^2 - e_3^2 & 2(e_2e_3 - e_0e_1) \\ 2(e_1e_3 - e_0e_2) & 2(e_0e_1 + e_2e_3) & e_0^2 - e_1^2 - e_2^2 + e_3^2 \end{bmatrix} \begin{bmatrix} u \\ v \\ w \end{bmatrix} \\ \begin{bmatrix} \dot{e}_0 \\ \dot{e}_1 \\ \dot{e}_2 \\ \dot{e}_3 \end{bmatrix} &= \frac{1}{2} \begin{bmatrix} -e_1 & -e_2 & -e_3 \\ e_0 & -e_3 & e_2 \\ e_3 & e_0 & -e_1 \\ -e_2 & e_1 & e_0 \end{bmatrix} \begin{bmatrix} p \\ q \\ r \end{bmatrix} \end{aligned}$$

where

$$J^{-1} = \frac{1}{J_{xx}J_{yy}J_{zz} - J_{yy}J_{xz}^2} \begin{bmatrix} J_{yy}J_{zz} & 0 & J_{yy}J_{xz} \\ 0 & J_{xx}J_{zz} - J_{xz}^2 & 0 \\ J_{yy}J_{xz} & 0 & J_{xx}J_{yy} \end{bmatrix}$$

and each member of the quaternion e_i , $i = 0, 1, 2, 3$ is normalized:

$$e_i = \frac{e_i}{\sqrt{e_0^2 + e_1^2 + e_2^2 + e_3^2}}$$

7.2 Derivation of the Airplane Equations of Motion

The remainder of this chapter derives the equations presented in Section 7.1.2.

7.2.1 Quaternion Rotations

A 3D rotation about any unit vector $\hat{e} = \langle e_x \hat{i} \quad e_y \hat{j} \quad e_z \hat{k} \rangle$ pointing in any direction can be described by quaternions. When describing rotations, a quaternion consists of an angle ρ and a unit vector \hat{e} . A quaternion q can be described by four parts, e_0 , e_1 , e_2 , and e_3 as follows:

$$q = \cos \frac{\rho}{2} + \sin \frac{\rho}{2} \langle e_x \hat{i} \quad e_y \hat{j} \quad e_z \hat{k} \rangle \quad (7.2)$$

$$= e_0 + \langle e_1 \hat{i} \quad e_2 \hat{j} \quad e_3 \hat{k} \rangle \quad (7.3)$$

where $e_0 = \cos \frac{\rho}{2}$, $e_1 = \sin \frac{\rho}{2} e_x$, $e_2 = \sin \frac{\rho}{2} e_y$, and $e_3 = \sin \frac{\rho}{2} e_z$. A quaternion rotation is illustrated in Figure 7.2. The body-fixed point p is rotated ρ radians about the unit axis \hat{e} .

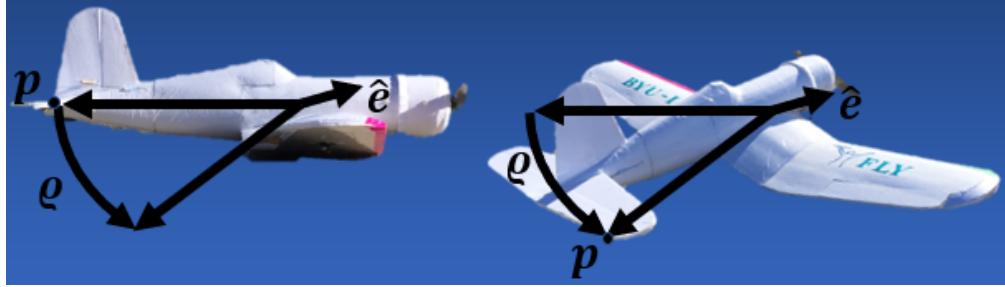


Figure 7.2 A rotation of ρ radians about the unit axis \hat{e}

Using quaternions, a rotation matrix can calculate the body-fixed x, y, and z coordinates in the inertial frame:

$$\begin{bmatrix} p_{x,I} \\ p_{y,I} \\ p_{z,I} \end{bmatrix} = \begin{bmatrix} e_0^2 + e_1^2 - e_2^2 - e_3^2 & 2(e_1 e_2 - e_0 e_3) & 2(e_0 e_2 + e_1 e_3) \\ 2(e_0 e_3 + e_1 e_2) & e_0^2 - e_1^2 + e_2^2 - e_3^2 & 2(e_2 e_3 - e_0 e_1) \\ 2(e_1 e_3 - e_0 e_2) & 2(e_0 e_1 + e_2 e_3) & e_0^2 - e_1^2 - e_2^2 + e_3^2 \end{bmatrix} \begin{bmatrix} p_x \\ p_y \\ p_z \end{bmatrix} \quad (7.4)$$

Eq. (7.4) describes the locations of the body-fixed x, y, and z positions (p_x, p_y, p_z) with coordinates ($p_{x,I}, p_{y,I}, p_{z,I}$) in the inertial frame.

7.2.2 Properties of Orthonormal Rotation Matrices

The rotation matrix in Eq. (7.4) is orthonormal. Normalized quaternions must satisfy the condition

$$e_i = \frac{e_i}{\sqrt{e_0^2 + e_1^2 + e_2^2 + e_3^2}} \quad (7.5)$$

for $i = 0, 1, 2, 3$.

Orthonormal rotation matrices have some useful properties. First, the inverse of an orthonormal rotation matrix \mathcal{R} is equal to its transpose:

$$\mathcal{R}^{-1} = \mathcal{R}^T \quad (7.6)$$

where the superscript T indicates the matrix transpose. As a result, a point $(p_{x,I}, p_{y,I}, p_{z,I})$ in the inertial coordinate frame has the following body-fixed coordinates:

$$\begin{bmatrix} p_x \\ p_y \\ p_z \end{bmatrix} = \begin{bmatrix} e_0^2 + e_1^2 - e_2^2 - e_3^2 & 2(e_0 e_3 + e_1 e_2) & 2(e_1 e_3 - e_0 e_2) \\ 2(e_1 e_2 - e_0 e_3) & e_0^2 - e_1^2 + e_2^2 - e_3^2 & 2(e_0 e_1 + e_2 e_3) \\ 2(e_0 e_2 + e_1 e_3) & 2(e_2 e_3 - e_0 e_1) & e_0^2 - e_1^2 - e_2^2 + e_3^2 \end{bmatrix} \begin{bmatrix} p_{x,I} \\ p_{y,I} \\ p_{z,I} \end{bmatrix} \quad (7.7)$$

Another useful property is that the determinant of an orthonormal rotation matrix is one.

$$\det(\mathcal{R}) = 1$$

Orthonormal quaternion rotations are used extensively in the kinematic equations for an aircraft.

7.3 Equations of Motion

7.2.3 Angular Velocity and Quaternions

Angular velocity ω (rad/s) of a rigid body in three-dimensional space describes the rate of rotation of the rigid body. It is represented by an array of three body-fixed components:

$$\omega = \begin{bmatrix} p \\ q \\ r \end{bmatrix} \quad (7.8)$$

where p (rad/s) is the instantaneous roll rotation rate about the x-axis, q is the instantaneous pitch rotation rate about the y-axis, and r is the instantaneous yaw rotation about the z-axis.

Graf¹ does an excellent job explaining quaternion math. He shows the derivation to relate the time-derivative of a unit quaternion to the angular velocities.

$$\begin{bmatrix} \dot{e}_0 \\ \dot{e}_1 \\ \dot{e}_2 \\ \dot{e}_3 \end{bmatrix} = \frac{1}{2} \begin{bmatrix} -e_1 & -e_2 & -e_3 \\ e_0 & -e_3 & e_2 \\ e_3 & e_0 & -e_1 \\ -e_2 & e_1 & e_0 \end{bmatrix} \begin{bmatrix} p \\ q \\ r \end{bmatrix} \quad (7.9)$$

7.2.4 Coordinate Frame Translations

After applying rotations, we can translate the aircraft to a new position relative to the origin of the inertial frame. To do so, we add the inertial frame displacements to each point on the aircraft. For example, if the center of gravity of the aircraft is translated 5 m north of the origin of the inertial frame, we must also translate every point on the aircraft 5 m north after first applying any rotations.

7.3 Equations of Motion

This section derives the equations of motion for the aircraft. These equations are used to determine the position and orientation of the aircraft. First we will derive the equations of motion for any rigid body in 3D space. Then we will use the symmetry of aircraft to simplify the equations.

7.3.1 General 6 DOF Motion of a Rigid Body

Consider any rigid body having six degrees of freedom (DOF). The first three DOF are translations in the x, y, and z orthogonal directions. The velocity with respect to a body-fixed coordinate frame is $V = [u \ v \ w]^T$, where u , v , and w are the linear velocities in the x, y, and z body-fixed directions respectively. The next three DOF are rotational: first rotation ϕ_x around the body-fixed x-axis, then rotation ψ_y around the body-fixed y-axis, and finally rotation θ_z around the body fixed z-axis. The subscripts on the angle symbols indicate the axes of rotation. The angular velocities in the body-fixed frame are $\omega = [p \ q \ r]^T$, where p , q , and r are the angular velocities about the x, y, and z axes respectively.

¹see Basile Graf, “Quaternions and Dynamics”, 18 Nov. 2008, Cornell University, Online (accessed March 2021): <https://arxiv.org/pdf/0811.2889.pdf>

Newton's equations of motion describe the dynamic behavior of the airplane due to applied forces and torques. These equations of motion about the center of gravity for any rotating rigid body with these six DOF can be written as follows:

$$F = m \frac{dV_T}{dt} \quad (7.10)$$

$$T = \frac{dH_T}{dt} \quad (7.11)$$

where $F = [f_x \ f_y \ f_z]^T$ are the external forces and $T = [T_x \ T_y \ T_z]$ are the external torques acting on the rigid body. Angular momentum H in the body-fixed reference frame is calculated as follows

$$H = J \begin{bmatrix} p \\ q \\ r \end{bmatrix} \quad (7.12)$$

where J is the body-fixed moment of inertia matrix:

$$J = \begin{bmatrix} J_{xx} & -J_{xy} & -J_{xz} \\ -J_{xy} & J_{yy} & -J_{yz} \\ -J_{xz} & -J_{yz} & J_{zz} \end{bmatrix} \quad (7.13)$$

For a rotating reference frame, the total time-derivatives $\frac{da_T}{dt}$ of a vector a_T can be calculated as the summation of linear and rotational parts. The linear part is $\frac{da}{dt}$, and the rotational part is the cross-product between the angular velocity ω and the vector a :

$$\frac{da_T}{dt} = \frac{da}{dt} + \omega \times a \quad (7.14)$$

Therefore, the total derivative of the velocity $\frac{dV_T}{dt}$ in the body frame of the rigid body becomes

$$\begin{aligned} \frac{dV_T}{dt} &= \frac{dV}{dt} + \omega \times V \\ &= \begin{bmatrix} \dot{u} \\ \dot{v} \\ \dot{w} \end{bmatrix} + \begin{bmatrix} qw - rv \\ ru - pw \\ pv - qu \end{bmatrix} \end{aligned} \quad (7.15)$$

Likewise, the total derivative $\frac{dH_T}{dt}$ of angular momentum relative to the body-fixed coordinate system in a

7.3 Equations of Motion

rotating rigid body is calculated by

$$\begin{aligned}
 \frac{dH_T}{dt} &= \frac{dH}{dt} + \omega \times H \\
 &= \begin{bmatrix} J_{xx} & -J_{xy} & -J_{xz} \\ -J_{xy} & J_{yy} & -J_{yz} \\ -J_{xz} & -J_{yz} & J_{zz} \end{bmatrix} \begin{bmatrix} \dot{p} \\ \dot{q} \\ \dot{r} \end{bmatrix} + \begin{bmatrix} p \\ q \\ r \end{bmatrix} \times \begin{bmatrix} J_{xx} & -J_{xy} & -J_{xz} \\ -J_{xy} & J_{yy} & -J_{yz} \\ -J_{xz} & -J_{yz} & J_{zz} \end{bmatrix} \begin{bmatrix} p \\ q \\ r \end{bmatrix} \\
 &= J \begin{bmatrix} \dot{p} \\ \dot{q} \\ \dot{r} \end{bmatrix} + \begin{bmatrix} (r^2 - q^2) J_{yz} + qr(J_{zz} - J_{yy}) - pqJ_{xz} + prJ_{xy} \\ (p^2 - r^2) J_{xz} + pr(J_{xx} - J_{zz}) + pqJ_{yz} - qrJ_{xy} \\ (q^2 - p^2) J_{xy} + pq(J_{yy} - J_{xx}) - prJ_{yz} + qrJ_{xz} \end{bmatrix}
 \end{aligned} \tag{7.16}$$

where J is the body-fixed inertia matrix defined in Eq. (7.13). We can substitute Eq. (7.16) back into Eq. (7.11) for the rotational dynamics equation. These dynamic equations, plus some kinematic equations, will describe the equations of motion in 3D for a rigid body.

7.3.2 State-Equations for 6 DOF Rigid Body Motion

State-Equations for 6 DOF Rigid Body Motion

$$\begin{aligned}
 \begin{bmatrix} \dot{u} \\ \dot{v} \\ \dot{w} \end{bmatrix} &= \begin{bmatrix} rv - qw \\ pw - ru \\ qu - pv \end{bmatrix} + \frac{1}{m} \begin{bmatrix} f_x \\ f_y \\ f_z \end{bmatrix} \\
 \begin{bmatrix} \dot{p} \\ \dot{q} \\ \dot{r} \end{bmatrix} &= J^{-1} \left(\begin{bmatrix} (q^2 - r^2) J_{yz} + qr(J_{yy} - J_{zz}) + pqJ_{xz} - prJ_{xy} \\ (r^2 - p^2) J_{xz} + pr(J_{zz} - J_{xx}) - pqJ_{yz} + qrJ_{xy} \\ (p^2 - q^2) J_{xy} + pq(J_{xx} - J_{yy}) + prJ_{yz} - qrJ_{xz} \end{bmatrix} + \begin{bmatrix} T_x \\ T_y \\ T_z \end{bmatrix} \right) \\
 \begin{bmatrix} \dot{x}_I \\ \dot{y}_I \\ \dot{z}_I \end{bmatrix} &= \begin{bmatrix} e_0^2 + e_1^2 - e_2^2 - e_3^2 & 2(e_1e_2 - e_0e_3) & 2(e_0e_2 + e_1e_3) \\ 2(e_0e_3 + e_1e_2) & e_0^2 - e_1^2 + e_2^2 - e_3^2 & 2(e_2e_3 - e_0e_1) \\ 2(e_1e_3 - e_0e_2) & 2(e_0e_1 + e_2e_3) & e_0^2 - e_1^2 - e_2^2 + e_3^2 \end{bmatrix} \begin{bmatrix} u \\ v \\ w \end{bmatrix} \\
 \begin{bmatrix} \dot{e}_0 \\ \dot{e}_1 \\ \dot{e}_2 \\ \dot{e}_3 \end{bmatrix} &= \frac{1}{2} \begin{bmatrix} -e_1 & -e_2 & -e_3 \\ e_0 & -e_3 & e_2 \\ e_3 & e_0 & -e_1 \\ -e_2 & e_1 & e_0 \end{bmatrix} \begin{bmatrix} p \\ q \\ r \end{bmatrix}
 \end{aligned}$$

The objective in deriving Newton's equations is to find a set of state-equations that describes the motion of the 6 DOF rigid body. To do so, we substitute the derivatives defined in Eqs. (7.15) and (7.16) into Newton's equations, Eqs. (7.10) and (7.11). Rearranging them results in a set of state equations for the

body fixed velocities u , v , and w and the angular velocities p , q , and r .

$$\begin{bmatrix} \dot{u} \\ \dot{v} \\ \dot{w} \end{bmatrix} = \begin{bmatrix} rv - qw \\ pw - ru \\ qu - pv \end{bmatrix} + \frac{1}{m} \begin{bmatrix} f_x \\ f_y \\ f_z \end{bmatrix} \quad (7.17)$$

$$\begin{bmatrix} \dot{p} \\ \dot{q} \\ \dot{r} \end{bmatrix} = J^{-1} \left(\begin{bmatrix} (q^2 - r^2)J_{yz} + qr(J_{yy} - J_{zz}) + pqJ_{xz} - prJ_{xy} \\ (r^2 - p^2)J_{xz} + pr(J_{zz} - J_{xx}) - pqJ_{yz} + qrJ_{xy} \\ (p^2 - q^2)J_{xy} + pq(J_{xx} - J_{yy}) + prJ_{yz} - qrJ_{xz} \end{bmatrix} + \begin{bmatrix} T_x \\ T_y \\ T_z \end{bmatrix} \right) \quad (7.18)$$

The variables in Eqs. (7.17) and (7.18) are as follows: \dot{u} , \dot{v} , and \dot{w} are the body-fixed x-axis, y-axis, and z-axis accelerations respectively; p , q , and r are the angular velocities around the body-fixed x, y, and z axes respectively; f_x , f_y , and f_z are the external forces acting on the rigid body aligned with the respective body-fixed x, y, and z axes; T_x , T_y , and T_z are the external torques with respect to the body-fixed axes; m is the mass of the rigid body; J_{xx} , J_{yy} , and J_{zz} are the mass moments of inertia about the body-fixed x, y, and z axes respectively. J_{xy} , J_{xz} , and J_{yz} are the body-fixed products of inertia; and J^{-1} is calculated as follows:

$$J^{-1} = \begin{bmatrix} J_{xx} & -J_{xy} & -J_{xz} \\ -J_{xy} & J_{yy} & -J_{yz} \\ -J_{xz} & -J_{yz} & J_{zz} \end{bmatrix}^{-1} \quad (7.19)$$

$$= \frac{1}{|J|} \begin{bmatrix} J_{yy}J_{zz} - J_{yz}^2 & J_{xy}J_{zz} + J_{xz}J_{yz} & J_{xy}J_{yz} + J_{xz}J_{yy} \\ J_{xy}J_{zz} + J_{yz}J_{xz} & J_{xx}J_{zz} - J_{xz}^2 & J_{xx}J_{yz} + J_{xz}J_{xy} \\ J_{xy}J_{yz} + J_{yy}J_{xz} & J_{xx}J_{yz} + J_{xy}J_{xz} & J_{xx}J_{yy} - J_{xy}^2 \end{bmatrix} \quad (7.20)$$

where the determinant $|J|$ of the moment of inertia matrix is

$$|J| = J_{xx}J_{yy}J_{zz} - 2J_{xy}J_{yz}J_{xz} - J_{xx}J_{yz}^2 - J_{yy}J_{xz}^2 - J_{zz}J_{xy}^2 \quad (7.21)$$

In addition to calculating velocities, it may be desirable to know the displacement and rotational orientation of the rigid body with respect to the inertial frame. When the angles, ϕ_x , ψ_y , and θ_z are zero, we assume that the body-fixed frame has the same orientation as the inertial frame. The x, y, and z axes of both frames are in the same directions.

A 3D rotation about any unit vector $\hat{e} = \langle e_x \hat{i} \quad e_y \hat{j} \quad e_z \hat{k} \rangle$ pointing in any direction can be described by quaternions. When describing rotations, a quaternion consists of an angle ρ and a unit vector \hat{e} . A quaternion q can be described by four parts, e_0 , e_1 , e_2 , and e_3 as follows:

$$q = \cos \frac{\rho}{2} + \sin \frac{\rho}{2} \langle e_x \hat{i} \quad e_y \hat{j} \quad e_z \hat{k} \rangle \quad (7.22)$$

$$= e_0 + \langle e_1 \hat{i} \quad e_2 \hat{j} \quad e_3 \hat{k} \rangle \quad (7.23)$$

where $e_0 = \cos \frac{\rho}{2}$, $e_1 = \sin \frac{\rho}{2} e_x$, $e_2 = \sin \frac{\rho}{2} e_y$, and $e_3 = \sin \frac{\rho}{2} e_z$. Using quaternions, a rotation from the inertial frame orientation to a different orientation is calculated using a quaternion rotation matrix. The

7.3 Equations of Motion

rotation matrix helps determine how the body-fixed x, y, and z coordinates are mapped to the inertial frame:

$$\begin{bmatrix} x_I \\ y_I \\ z_I \end{bmatrix} = \begin{bmatrix} e_0^2 + e_1^2 - e_2^2 - e_3^2 & 2(e_1 e_2 - e_0 e_3) & 2(e_0 e_2 + e_1 e_3) \\ 2(e_0 e_3 + e_1 e_2) & e_0^2 - e_1^2 + e_2^2 - e_3^2 & 2(e_2 e_3 - e_0 e_1) \\ 2(e_1 e_3 - e_0 e_2) & 2(e_0 e_1 + e_2 e_3) & e_0^2 - e_1^2 - e_2^2 + e_3^2 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} \quad (7.24)$$

Eq. (7.24) describes the x, y, and z positions in the body-frame with respect to the inertial frame x_I , y_I , and z_I . The body-fixed velocities u , v , and w at the center of gravity can also be described in the inertial frame using the quaternion rotation matrix:

$$\begin{bmatrix} \dot{x}_I \\ \dot{y}_I \\ \dot{z}_I \end{bmatrix} = \begin{bmatrix} e_0^2 + e_1^2 - e_2^2 - e_3^2 & 2(e_1 e_2 - e_0 e_3) & 2(e_0 e_2 + e_1 e_3) \\ 2(e_0 e_3 + e_1 e_2) & e_0^2 - e_1^2 + e_2^2 - e_3^2 & 2(e_2 e_3 - e_0 e_1) \\ 2(e_1 e_3 - e_0 e_2) & 2(e_0 e_1 + e_2 e_3) & e_0^2 - e_1^2 - e_2^2 + e_3^2 \end{bmatrix} \begin{bmatrix} u \\ v \\ w \end{bmatrix} \quad (7.25)$$

This kinematic state-equation, Eq. (7.25), can be solved to find the position of the center of gravity of a rigid body in the inertial frame.

The time-derivative of a unit quaternion can be calculated as a function of the angular velocities p , q , and r as follows²:

$$\begin{bmatrix} \dot{e}_0 \\ \dot{e}_1 \\ \dot{e}_2 \\ \dot{e}_3 \end{bmatrix} = \frac{1}{2} \begin{bmatrix} 0 & -p & -q & -r \\ p & 0 & r & -q \\ q & -r & 0 & p \\ r & q & -p & 0 \end{bmatrix} \begin{bmatrix} e_0 \\ e_1 \\ e_2 \\ e_3 \end{bmatrix} \quad (7.26)$$

or

$$\begin{bmatrix} \dot{e}_0 \\ \dot{e}_1 \\ \dot{e}_2 \\ \dot{e}_3 \end{bmatrix} = \frac{1}{2} \begin{bmatrix} -e_1 & -e_2 & -e_3 \\ e_0 & -e_3 & e_2 \\ e_3 & e_0 & -e_1 \\ -e_2 & e_1 & e_0 \end{bmatrix} \begin{bmatrix} p \\ q \\ r \end{bmatrix} \quad (7.27)$$

The complete set of state-equations for 6 DOF rigid body motion combines Eqs. (7.17), (7.18), (7.25), and (7.27). These equations were summarized at the beginning of the section.

7.3.3 Equations of Motion for an Aircraft

Traditionally, when modeling aircraft, the body-fixed coordinate frame is oriented as follows (see Figure 7.3): the x-axis extends from the center of gravity out through the nose at the front of the plane; the y-axis extends from the center of gravity out through the right wing; the z-axis extends from the center of gravity down through the bottom of the fuselage.

In most airplanes, there is symmetry about the x-z plane. Therefore $J_{xy} = J_{yz} = 0$, and the equations of motion given in Section 7.3.2 can be simplified.

²see Basile Graf, “Quaternions and Dynamics”, 18 Nov. 2008, Cornell University, Online (accessed March 2021): <https://arxiv.org/pdf/0811.2889.pdf>

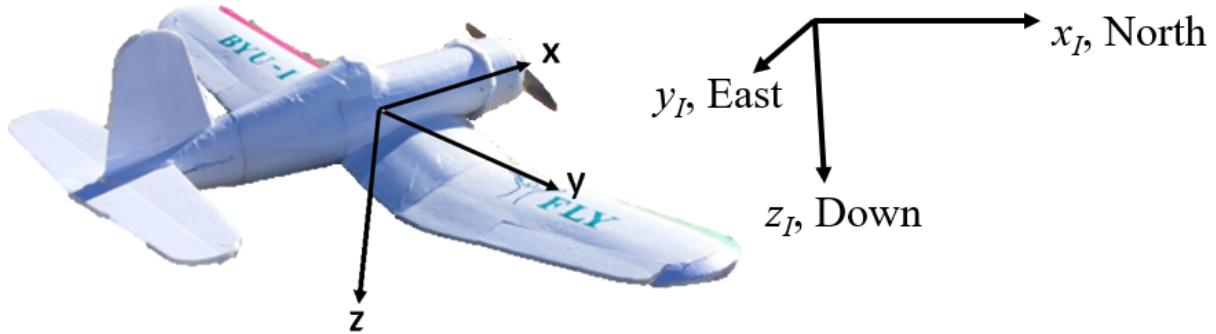


Figure 7.3 Left: Traditional body-fixed coordinate system.

Equations of motion for an aircraft:

$$\begin{bmatrix} \dot{u} \\ \dot{v} \\ \dot{w} \end{bmatrix} = \begin{bmatrix} rv - qw \\ pw - ru \\ qu - pv \end{bmatrix} + \frac{1}{m} \begin{bmatrix} f_x \\ f_y \\ f_z \end{bmatrix} \quad (7.28)$$

$$\begin{bmatrix} \dot{p} \\ \dot{q} \\ \dot{r} \end{bmatrix} = J^{-1} \left(\begin{bmatrix} qr(J_{yy} - J_{zz}) + pqJ_{xz} \\ pr(J_{zz} - J_{xx}) + (r^2 - p^2)J_{xz} \\ pq(J_{xx} - J_{yy}) - qrJ_{xz} \end{bmatrix} + \begin{bmatrix} T_x \\ T_y \\ T_z \end{bmatrix} \right) \quad (7.29)$$

$$\begin{bmatrix} \dot{x}_I \\ \dot{y}_I \\ \dot{z}_I \end{bmatrix} = \begin{bmatrix} e_0^2 + e_1^2 - e_2^2 - e_3^2 & 2(e_1e_2 - e_0e_3) & 2(e_0e_2 + e_1e_3) \\ 2(e_0e_3 + e_1e_2) & e_0^2 - e_1^2 + e_2^2 - e_3^2 & 2(e_2e_3 - e_0e_1) \\ 2(e_1e_3 - e_0e_2) & 2(e_0e_1 + e_2e_3) & e_0^2 - e_1^2 - e_2^2 + e_3^2 \end{bmatrix} \begin{bmatrix} u \\ v \\ w \end{bmatrix} \quad (7.30)$$

$$\begin{bmatrix} \dot{e}_0 \\ \dot{e}_1 \\ \dot{e}_2 \\ \dot{e}_3 \end{bmatrix} = \frac{1}{2} \begin{bmatrix} -e_1 & -e_2 & -e_3 \\ e_0 & -e_3 & e_2 \\ e_3 & e_0 & -e_1 \\ -e_2 & e_1 & e_0 \end{bmatrix} \begin{bmatrix} p \\ q \\ r \end{bmatrix} \quad (7.31)$$

where

$$J^{-1} = \frac{1}{J_{xx}J_{yy}J_{zz} - J_{yy}J_{xz}^2} \begin{bmatrix} J_{yy}J_{zz} & 0 & J_{yy}J_{xz} \\ 0 & J_{xx}J_{zz} - J_{xz}^2 & 0 \\ J_{yy}J_{xz} & 0 & J_{xx}J_{yy} \end{bmatrix} \quad (7.32)$$

* These equations require that the quaternion is a unit quaternion. This can be accomplished by normalization at each iteration in the simulation after solving Eq. (7.31). Each member of the quaternion e_i , $i = 0, 1, 2, 3$ is normalized as follows:

$$e_i = \frac{e_i}{\sqrt{e_0^2 + e_1^2 + e_2^2 + e_3^2}} \quad (7.33)$$

7.4 Airspeed, Angle of Attack, and Side-slip Angle

7.4 Airspeed, Angle of Attack, and Side-slip Angle

This section summarizes the equations that are used to calculate the forces and torques on the airplane. These forces and torques become the inputs to the equations of motion that describe the airplane dynamics and kinematics (see Eq. (7.28) - Eq. (7.33)). The force and torque equations depend on the airspeed V_a (m/s) of the airplane. The airspeed vector V_a^b in the body frame is affected by the body-frame wind speeds u_w , v_w , and w_w (m/s) and aircraft speeds u , v , and w (m/s) (see Section 7.5) as follows:

$$V_a^b = \begin{bmatrix} u_r \\ v_r \\ w_r \end{bmatrix} = \begin{bmatrix} u - u_w \\ v - v_w \\ w - w_w \end{bmatrix} \quad (7.34)$$

The airspeed V_a is then the magnitude of the airspeed vector V_a^b :

$$V_a = \sqrt{u_r^2 + v_r^2 + w_r^2} \quad (7.35)$$

The angle of attack α (rad) is another frequently used variable. It is a function of the relative air speeds w_r and V_a (m/s) and is calculated as follows:

$$\alpha = \tan^{-1} \left(\frac{w_r}{V_a} \right) \quad (7.36)$$

Finally, the side-slip angle β (rad) is another critical variable. It is dependent on the relative airspeed v_r (m/s) in the body y-axis and the airspeed V_a (m/s).

$$\beta = \sin^{-1} \left(\frac{v_r}{V_a} \right) \quad (7.37)$$

7.5 Wind Model

The wind model consists of two different parts: (1) constant wind speeds and (2) gusts. The constant wind speeds in the inertial coordinate frame are w_{xI} , w_{yI} , and w_{zI} (m/s) in the x, y, and z directions respectively.

The wind gusts are generated using random Gaussian noise processes \mathcal{N}_1 , \mathcal{N}_2 , and \mathcal{N}_3 , each of which are zero mean with a standard deviation of one. These become inputs in the Dryden Gust Model to calculate the wind gusts. The wind gust speed in the body-fixed x-axis u_{wg} (m/s) is calculated by solving the differential equation:

$$\dot{u}_{wg} = -\frac{V_a}{L_u} u_{wg} + \sigma_u \sqrt{\frac{2V_a}{\pi L_u}} \mathcal{N}_1 \quad (7.38)$$

The gust speed in the body-fixed y-axis v_{wg} (m/s) is calculated by solving

$$\begin{bmatrix} \dot{x}_1 \\ \dot{x}_2 \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ -\left(\frac{V_a}{L_v}\right)^2 & -2\frac{V_a}{L_v} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} + \begin{bmatrix} 0 \\ \sigma_v \sqrt{\frac{3V_a}{\pi L_v}} \end{bmatrix} \mathcal{N}_2 \quad (7.39)$$

$$v_{w_g} = \begin{bmatrix} \frac{V_a}{\sqrt{3L_v}} & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \quad (7.40)$$

The gust speed in the body-fixed z-axis w_{w_g} (m/s) is calculated by solving

$$\begin{bmatrix} \dot{x}_3 \\ \dot{x}_4 \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ -\left(\frac{V_a}{L_w}\right)^2 & -2\frac{V_a}{L_w} \end{bmatrix} \begin{bmatrix} x_3 \\ x_4 \end{bmatrix} + \begin{bmatrix} 0 \\ \sigma_w \sqrt{\frac{3V_a}{\pi L_w}} \end{bmatrix} \mathcal{N}_3 \quad (7.41)$$

$$w_{w_g} = \begin{bmatrix} \frac{V_a}{\sqrt{3L_w}} & 1 \end{bmatrix} \begin{bmatrix} x_3 \\ x_4 \end{bmatrix} \quad (7.42)$$

The Dryden Gust Model parameters are given in Table 7.3. The initial conditions for the state variables x_1 , x_2 , x_3 , and x_4 are all zero.

Table 7.3 Parameters for the Dryden Gust Model

Gust Parameters	$L_u = L_v$ (m)	L_w (m)	$\sigma_u = \sigma_v$ (m/s)	σ_w (m/s)
≈ 50 m altitude, light	200	50	1.06	0.7
≈ 50 m altitude, moderate	200	50	2.12	1.4
≈ 600 m altitude, light	533	533	1.5	1.5
≈ 600 m altitude, moderate	533	533	3.0	3.0

To get the overall wind speeds u_w , v_w , and w_w (m/s) in the body-fixed x, y, and z axes respectively, the constant wind speeds w_{xI} , w_{yI} , and w_{zI} (m/s) are converted to the body-frame and added to the gust speeds:

$$\begin{bmatrix} u_w \\ v_w \\ w_w \end{bmatrix} = R_I^b \begin{bmatrix} w_{xI} \\ w_{yI} \\ w_{zI} \end{bmatrix} + \begin{bmatrix} u_{w_g} \\ v_{w_g} \\ w_{w_g} \end{bmatrix} \quad (7.43)$$

where R_I^b is the rotation matrix from the inertial-frame to the body-frame:

$$R_I^b = \begin{bmatrix} e_0^2 + e_1^2 - e_2^2 - e_3^2 & 2(e_0e_3 + e_1e_2) & 2(e_1e_3 - e_0e_2) \\ 2(e_1e_2 - e_0e_3) & e_0^2 - e_1^2 + e_2^2 - e_3^2 & 2(e_0e_1 + e_2e_3) \\ 2(e_0e_2 + e_1e_3) & 2(e_2e_3 - e_0e_1) & e_0^2 - e_1^2 - e_2^2 + e_3^2 \end{bmatrix} \quad (7.44)$$

The quaternion variables e_0, e_1, e_2, e_3 are calculated using the equations of motion derived in Section 7.3.3.

7.6 Input Commands

7.6 Input Commands

A four-channel remote controller with two joysticks can command four input signals: δ_e , δ_a , δ_t , and δ_r (see Figure 7.4). The command δ_e (-1 to +1) controls the pitching rate of the aircraft. A value of $\delta_e = +1$ causes the aircraft to nose down. A value of $\delta_e = -1$ causes the aircraft to pitch upwards. In traditional fixed-wing aircraft, δ_e controls the elevator. For a quadcopter, δ_e causes a difference between the speeds of the fore and aft propellers.



Figure 7.4 The input commands δ_e , δ_a , δ_t , and δ_r in their +1 positions

The command δ_a (-1 to +1) controls the rolling rate of the aircraft. A value of $\delta_a = +1$ causes the aircraft to roll to the right. A value of $\delta_a = -1$ causes the aircraft to roll to the left. In traditional fixed-wing airplanes, the command δ_a controls the ailerons. In a quadcopter, δ_a causes a difference between the speeds of the left and right propellers.

The command δ_t (0 to +1) controls the propeller speed. A value of $\delta_t = +1$ commands full throttle. A value of $\delta_t = 0$ stops the propellers from spinning.

The command δ_r (-1 to +1) controls the yaw rate of an aircraft. A command of $\delta_r = +1$ causes the aircraft to yaw to the right. A command of $\delta_r = -1$ causes the aircraft to yaw to the left. In a traditional fixed-wing airplane, δ_r controls the rudder. In a quadcopter, it causes a difference in the speeds of diagonally opposite propellers, which results in a net yaw torque.

7.7 Modeling Propellers



Figure 7.5 A 0945 propeller

The goal of this section is to determine a mathematical relationship between the throttle command δ_t (0-1) from the remote controller and the propeller thrust f_T (N) and torque T_Q (N m). These relationships depend on a number of parameters: propeller diameter, blade pitch, number of blades, blade area, motor torque and speed ratios, motor size, battery voltage, battery current limits, air density, relative airspeed, and remote controller throttle curves. Though long, this list is still incomplete. However, we will use some approximations to simplify the analysis while attempting to retain the most relevant physical relationships.

7.7.1 Propeller Thrust

Aerodynamic force f_{aero} depends on the density of the fluid ρ , the fluid velocity V , the area A of the object in the fluid stream, and a lift or drag coefficient C .

$$f_{\text{aero}} = \frac{1}{2} \rho V^2 A C \quad (7.45)$$

The relative fluid velocity, area, and pitch vary along the length of the propeller, thereby causing the aerodynamic force to vary as well. We would need to integrate these forces along the length of the propeller to find the overall force. The complicated shape of the propeller makes this a difficult task.

Dimensional analysis and experimental studies³ suggest that the overall propeller thrust force f_T (N) can be approximated by the dimensionless relationship

$$\frac{f_T}{\rho n^2 D^4} \approx C_T(J, \alpha_b) \quad (7.46)$$

where f_T (N) is the propeller thrust, D (m) is the propeller diameter, ρ (kg/m^3) is the air density, and n (rev/s) is the propeller angular speed. The thrust coefficient $C_T(J, \alpha_b)$ is dimensionless. It is usually

³see McCormick, "Aerodynamics, Aeronautics and Flight Mechanics", 2nd edition

7.7 Modeling Propellers

determined experimentally. It is a function of the advance ratio $J = \frac{u_r}{nD}$ and the average pitch α_b (inch). The pitch α_b is defined as the ratio of the forward travel of the propeller per rotation assuming no slip. u_r (m/s) is the relative airspeed; for example, if there is no wind u_r is the x-axis forward velocity of a fixed-wing aircraft.

To understand the thrust coefficient $C_T(J, \alpha_b)$ better, consider a few different possibilities for a fixed-wing aircraft. Thrust is proportional to the change in the velocity of the air behind the propeller minus the velocity of the air in front of it: $f_T = \dot{m}(u_{\text{behind}} - u_{\text{ahead}})$. The greater the change in velocity, the greater the thrust. If the propeller speed is constant, nD is constant. As the plane flies faster, $u_r = u_{\text{ahead}}$ increases. If u_r increases but nD is constant, the thrust decreases because the change in velocity is less. Therefore as the advance ratio $J = \frac{u_r}{nD}$ increases, the thrust f_T (and C_T) decreases. If the advance ratio J decreases, the thrust f_T (and C_T) increases.

Next consider the effect of the propeller blade pitch α_b . If it is zero, the propeller produces no thrust. As α_b gets larger up to a certain point, it produces more thrust. Once the pitch is so great that the propeller is parallel to the airflow, it again produces no thrust. Above this pitch, it begins producing thrust in the opposite direction.

Barnes McCormick published a set of experimentally determined thrust coefficient C_p curves that depend on the advance ratio J and blade pitch α_b . A graph of these curves can be found on page 306 of his book, "Aerodynamics, Aeronautics and Flight Mechanics", 2nd edition.

McCormick's data was for a propeller with 3 blades. More recent curves have been produced for common two-blade model airplane propellers. Researchers at the University of Illinois Urbana-Champaign (UIUC) reproduced matching data to those collected at Ohio State University⁴ (OSU). The data are for 1050 and 1070 propellers, *i.e.*, 10 inch diameter propellers with 5 inch and 7 inch pitches respectively.

In this document, we derived an equation to fit the thrust coefficient data collected by UIUC and OSU. Figure 7.6 shows the fit equations on the same graph as the data.

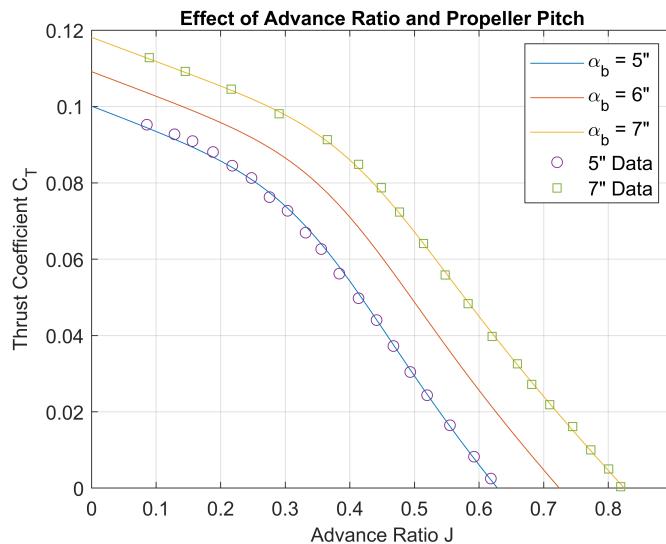


Figure 7.6 The propeller thrust coefficient C_T is affected by the pitch α_b and the advance ratio $J = \frac{u_r}{nD}$

⁴J.B. Brandt, R.W. Deters, G.K. Ananda, O.D. Dantsker, and M.S. Selig , UIUC Propeller Database, Vols 1-3, University of Illinois at Urbana-Champaign, retrieved 2 February 2021 from <https://m-selig.ae.illinois.edu/props/propDB.html>.

The empirical equation used to generate the curves in Figure 7.6 is

$$C_T = zC_{T,L} + (1 - z)C_{T,u} \quad (7.47)$$

where

$$z = \frac{1}{1 + e^{20(0.14+0.018\alpha_b-0.6J)}} \quad (7.48)$$

$$C_{T,L} = 0.11 - 0.18(J - 0.105(\alpha_b - 5)) \quad (7.49)$$

and

$$C_{T,u} = 0.1 + 0.009(\alpha_b - 5) - 0.065J \quad (7.50)$$

and α_b has units of inches.

Note that a single equation, Eq. (7.47), is used to fit the data for both the 1050 and 1070 propellers. The advantage of having a single equation for the thrust coefficient is that it gives us the ability to interpolate and extrapolate to fit slightly different propellers. For example, the thrust coefficient C_T for a 0945 propeller (9 inch diameter with a 4.5 inch pitch) can be approximated by using $\alpha_b = 4.5$ inches in Eq. (7.47) through Eq. (7.50).

Using Eqs. (7.46) and (7.47), the propeller thrust is

$$f_T = \rho n^2 D^4 C_T \quad (7.51)$$

7.7.2 Propeller Torque

The derivation for the propeller torque is nearly identical to the propeller thrust in the previous section. However, thrust is a force, but torque is a force multiplied by a moment arm. The dimensional analysis for torque treats the propeller diameter as the moment arm distance. Therefore, the equation for propeller torque T_Q (N m) is like the thrust equation Eq. (7.51) except it has an additional power on the diameter (D^5) and a dimensionless torque coefficient C_Q instead of a thrust coefficient C_T .

$$T_Q = \rho n^2 D^5 C_Q \quad (7.52)$$

In addition to thrust coefficient curves, the researchers at UIUC and OSU also published torque coefficient data for the 1050 and 1070 propellers. We fit empirical equations to this data. The data and fit equations are graphed in Figure 7.7.

The empirical equation used to generate the curves in Figure 7.6 is

$$C_Q = z_Q C_{Q,L} + (1 - z_Q) C_{Q,u} \quad (7.53)$$

where

$$z_Q = \frac{1}{1 + e^{10(0.16+0.05\alpha_b-J)}} \quad (7.54)$$

$$C_{Q,L} = 0.0121 - 0.017(J - 0.1(\alpha_b - 5)) \quad (7.55)$$

and

$$C_{Q,u} = 0.0057 + 0.00125(\alpha_b - 5) - 0.0005J \quad (7.56)$$

and α_b has units of inches.

Note that a single equation, Eq. (7.53), is used to fit the data for both the 1050 and 1070 propellers. The advantage of having a single equation for the torque coefficient is that it gives us the ability to interpolate

7.7 Modeling Propellers

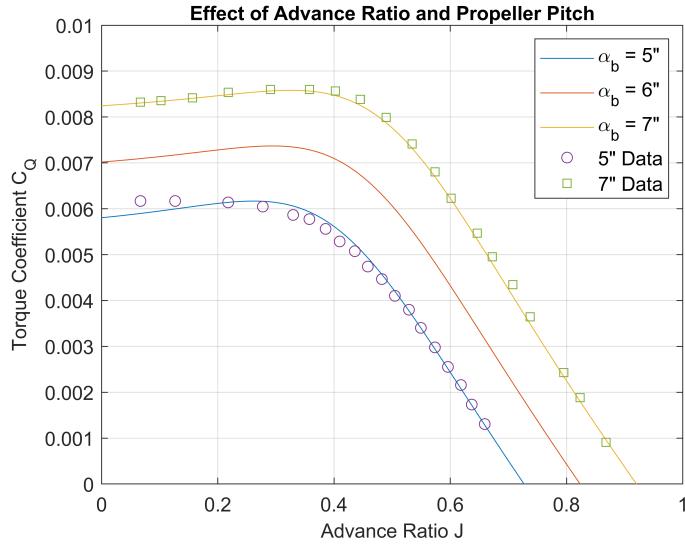


Figure 7.7 The propeller torque coefficient C_Q is affected by the pitch α_b and the advance ratio $J = \frac{u_r}{nD}$

and extrapolate to fit slightly different propellers. For example, the torque coefficient C_Q for a 0945 propeller (9 inch diameter with a 4.5 inch pitch) can be approximated by using $\alpha_b = 4.5$ in Eq. (7.53) through Eq. (7.56).

7.7.3 Motor Equations

Recall that our ultimate goal for modeling propellers is to find a relationship between the throttle command δ_t and the thrust and torque produced by the propeller. Section 7.7.1 modeled propeller thrust, and Section 7.7.2 modeled propeller torque; however, both models are incomplete without a model of the motor.

When spinning at a constant (steady) rate, brushed or brushless DC (Direct Current) motors can be modeled by the following relationship⁵:

$$T_Q = k_T \left(\frac{1}{R} (V_{in} - 2\pi k_V n) - I_0 \right) \quad (7.57)$$

where T_Q (N m) is the motor torque, k_T (Nm/A) is the motor torque constant, R (Ω) is the motor electrical resistance, V_{in} (V) is the supplied voltage, n (rev/s) is the rotational speed, and I_0 (A) is the no-load current of the motor. The values for k_T , R , and I_0 are often provided by the manufacturer. Note that when converted to the same units, the motor speed constant k_V (rad/s/V) and the motor torque constant k_T (Nm/A) have the same value.

Setting the torque from Eq. (7.57) equal to that of Eq. (7.52) results in the following equation:

$$\rho D^5 C_Q n^2 + \frac{2\pi k_T k_V}{R} n + \left(\frac{k_T}{R} V_{in} - k_T I_0 \right) = 0 \quad (7.58)$$

⁵see Chapter 4 - Forces and Moments of chap4.pdf PDF Slides by R. Beard and T. McLain <https://uavbook.byu.edu/doku.php>, downloaded 2 February 2021

which at first glance appears to be a quadratic function of the rotational propeller speed n . However, since C_Q is a function of the advance ratio $J = \frac{u}{nD}$, Eq. (7.58) is a complicated function of speed n . We would like to solve it for n as a function of the supplied voltage V_{in} . However, because of the complicated nature of C_Q , we cannot simply apply the quadratic formula to get an exact result in a single step.

Fortunately, however, Eq. (7.58) is numerically convergent, meaning that we can use an iterative guess-and-check method to calculate the value of n . First, we start by guessing a value of n and use it to find C_Q . Then we use the quadratic formula to solve Eq. (7.58) for n . If the calculated value matches our initial guess, we are done; otherwise, we use the new calculated value of n as our new guess to get C_Q and repeat the process. This iterative method is completed once the guessed value of n matches the value calculated by the quadratic equation to within an acceptable tolerance. The guess-and-check method is computationally expensive; therefore, Section 7.7.4 suggests alternative approaches.

The final relationship required for our motor model is to relate the supply voltage V_{in} and the throttle command δ_t . For this, we use a simple linear relationship:

$$V_{in} = V_{max}\delta_t \quad (7.59)$$

Since δ_t varies from 0 to 1, we use the maximum supply voltage V_{max} as the proportionality constant.

7.7.4 Computational Simplifications for Propeller Speed

The complicated nature of Eq. (7.58) can cause the calculation of propeller speed n to be computationally expensive. To improve computational speed, this section proposes a numerical simplification: Eq. (7.58) is solved offline and only once to find a maximum speed n_{max} (or n_{max} is determined experimentally). The offline calculations use the maximum voltage V_{max} at zero relative velocity $u_r = 0$. The resulting propeller speed is called n_{max} . Then the relationship between the throttle command δ_t (0-1) and the propeller speed n (rev/s) is approximated with a quadratic function.

$$n = n_{max} (1 - (1 - \delta_t)^2) \quad (7.60)$$

7.7.5 Summary of Propeller Equations

How to calculate f_T and T_Q :

The calculation of the propeller thrust force f_T and torque T_Q consists of two steps. The first step is performed offline and only once. The second step is performed at each iteration in the simulation.

Offline calculation solved only once:

(a) Determine n_{max} experimentally

OR

(b) Using the value $u_r = 0$ in the advance ratio $J = \frac{u_r}{nD}$ and $V_{in} = V_{max}$, calculate the propeller speed n using the equations

$$C_Q = z_Q C_{Q,L} + (1 - z_Q) C_{Q,u}$$

where

$$z_Q = \frac{1}{1 + e^{10(0.16 + 0.05\alpha_b - J)}}$$

7.7 Modeling Propellers

$$C_{Q,L} = 0.0121 - 0.017(J - 0.1(\alpha_b - 5))$$

and

$$C_{Q,u} = 0.0057 + 0.00125(\alpha_b - 5) - 0.0005J$$

Use a root-finding algorithm to determine the propeller speed n (rev/s).

$$\rho D^5 C_Q n^2 + \frac{2\pi k_T k_V}{R} n + \left(\frac{k_T}{R} V_{in} - k_T I_0 \right) = 0$$

Call the resulting propeller speed n_{max} .

Online calculations solved at each iteration of the simulation

Use the following relationship to find the propeller speed n :

$$n = n_{max} (1 - (1 - \delta_t)^2)$$

Using the value of the relative speed u_r in the advance ratio $J = \frac{u_r}{nD}$, calculate the torque coefficient.

$$C_Q = z_Q C_{Q,L} + (1 - z_Q) C_{Q,u}$$

where

$$z_Q = \frac{1}{1 + e^{10(0.16+0.05\alpha_b-J)}}$$

$$C_{Q,L} = 0.0121 - 0.017(J - 0.1(\alpha_b - 5))$$

and

$$C_{Q,u} = 0.0057 + 0.00125(\alpha_b - 5) - 0.0005J$$

Calculate the propeller torque.

$$T_{x,Q} = \rho n^2 D^5 C_Q$$

Calculate the thrust coefficient C_T .

$$C_T = z C_{T,L} + (1 - z) C_{T,u}$$

where

$$z = \frac{1}{1 + e^{20(0.14+0.018\alpha_b-0.6J)}}$$

$$C_{T,L} = 0.11 - 0.18(J - 0.105(\alpha_b - 5))$$

and

$$C_{T,u} = 0.1 + 0.009(\alpha_b - 5) - 0.065J$$

and α_b has units of inches. Finally, calculate the propeller thrust.

$$f_{x,T} = \rho n^2 D^4 C_T$$

7.8 Gravitational Forces

Gravitational forces act to pull the airplane down towards the center of the earth. Gravity exerts a force in the positive z-axis of the inertial frame. Because the plane may have a different orientation than the inertial frame, gravity results in components $f_{x,g}$, $f_{y,g}$, and $f_{z,g}$ (N) in potentially all three body-fixed directions. We use the quaternion rotation matrix R_I^b (see Eq. (7.44)) to convert from the inertial frame to the body-fixed frame:

$$\begin{bmatrix} f_{x,g} \\ f_{y,g} \\ f_{z,g} \end{bmatrix} = \begin{bmatrix} e_0^2 + e_1^2 - e_2^2 - e_3^2 & 2(e_0e_3 + e_1e_2) & 2(e_1e_3 - e_0e_2) \\ 2(e_1e_2 - e_0e_3) & e_0^2 - e_1^2 + e_2^2 - e_3^2 & 2(e_0e_1 + e_2e_3) \\ 2(e_0e_2 + e_1e_3) & 2(e_2e_3 - e_0e_1) & e_0^2 - e_1^2 - e_2^2 + e_3^2 \end{bmatrix} \begin{bmatrix} 0 \\ 0 \\ mg \end{bmatrix} \quad (7.61)$$

These forces are functions of the airplane mass m (kg) and the gravitational acceleration $g = 9.81 \text{ m/s}^2$. The equations depend on the quaternions e_0 , e_1 , e_2 , and e_3 . They are summarized below.

Gravitational Forces

$$\begin{bmatrix} f_{x,g} \\ f_{y,g} \\ f_{z,g} \end{bmatrix} = mg \begin{bmatrix} 2(e_1e_3 - e_0e_2) \\ 2(e_0e_1 + e_2e_3) \\ e_0^2 - e_1^2 - e_2^2 + e_3^2 \end{bmatrix} \quad (7.62)$$

7.9 Aerodynamics of Fixed-Wing Aircraft

Aerodynamic forces f_{aero} in general depend on the density of the fluid ρ , the fluid velocity V , the area A of the object in the fluid stream, and a lift or drag coefficient C .

$$f_{\text{aero}} = \frac{1}{2}\rho V^2 AC \quad (7.63)$$

Since torque result from forces multiplied by a moment arm, aerodynamic torques T_{aero} are calculated as

$$T_{\text{aero}} = \frac{1}{2}\rho V^2 ALC \quad (7.64)$$

where L is the length of the moment arm.

7.9.1 Lift, Drag, and Side-slip Forces

Aerodynamics cause lift f_z and drag f_x forces that act on the aircraft. The airspeed V_a (m/s) (see Eq. (7.35)) is the velocity V in the calculation of the aerodynamic force Eq. (7.63). The lift, drag, and side-slip coefficients in the calculations depend on a number of variables.

The independent variables in the drag and lift coefficients are the elevator command δ_e (-1 to 1) and the angle of attack α (rad) (see Eq. (7.36)). In this document, the lift C_L and drag coefficients are determined using equations that were loosely fit to data from an experimental airfoil (NACA 4412) in Figure 7.8. The fit equation for the lift does not completely account for the loss in lift at stall angles of

7.9 Aerodynamics of Fixed-Wing Aircraft

attack. Experimental data was not available for angles of attack exceeding 20 degrees. The offset between the CD NACA 4412 data and the CD Fit data is to account for the extra drag of the body of the airplane.

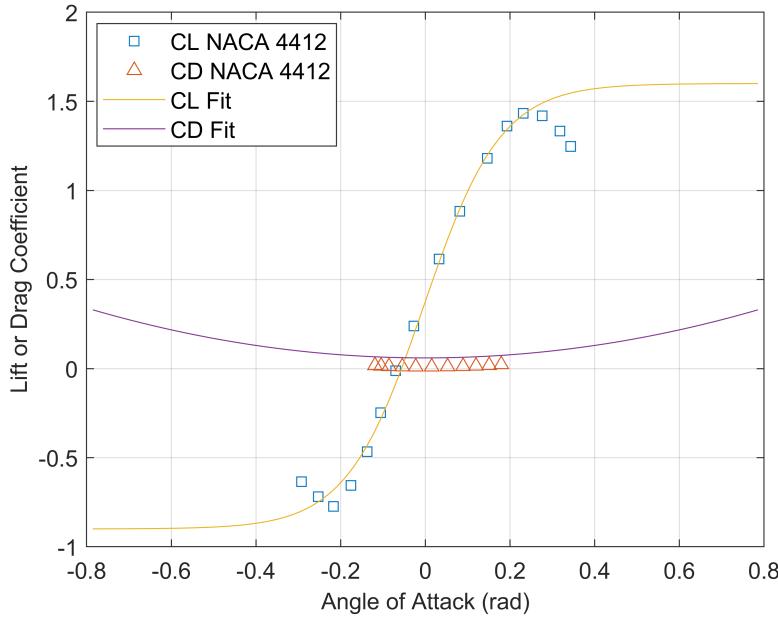


Figure 7.8 Lift (CL) and drag (CD) coefficients as a function of the angle of attack α . The NACA 4412 markers correspond to experimental airfoil data.

The lift coefficient is a sigmoidal function of the angle of attack α and a linear function of the elevator command δ_e .

$$C_L = - \left(C_{L,0} + \frac{C_{L,1}}{C_{L,2} + e^{-C_{L,3}(\alpha+C_{L,4})}} \right) - C_{L,\delta_e} \delta_e \quad (7.65)$$

The drag coefficient C_D is a quadratic function of the angle of attack α (see Figure 7.8) and a linear function of the elevator command δ_e .

$$C_D = - (C_{D,0} + C_{D,2}\alpha^2) - C_{D,\delta_e} \delta_e \quad (7.66)$$

The side-slip coefficient C_β is a linear function of the side-slip angle β from Eq. (7.37) and the rudder command δ_r (-1 to 1).

$$C_\beta = -C_{z,\beta}\beta - C_{z,\delta_r}\delta_r \quad (7.67)$$

7.9.2 Aerodynamic Torques

Aerodynamic torques are caused by two fundamental phenomena. First, if the airplane is spinning about any axis, air resistance dampens the spinning motion. For example, if the airplane is rolling about the x-axis with a roll rate of p rad/s, a resistive torque $T_{b,p}$ dampens the motion. We model this resistive torque with a linear relationship with a sign opposite to the rotation. The resistive roll torque is

$$T_{b,p} = -b_p p \quad (7.68)$$

The resistive pitch torque is

$$T_{b,q} = -b_q q \quad (7.69)$$

The resistive yaw torque is

$$T_{b,r} = -b_r r \quad (7.70)$$

The second phenomenon that creates aerodynamic torques is caused by aerodynamic forces that act on the aircraft at a moment arm distance from the center of gravity. These force-and-moment-arm combinations cause torques that can be modeled by the aerodynamic force multiplied by a distance.

$$T_{\text{aero}} = \frac{1}{2} \rho V_a^2 ALC \quad (7.71)$$

The moment arm is L , and the coefficient is C . The torque coefficient C_{Ty} for the pitching torque is a linear function of the angle of attack α and the elevator command δ_e :

$$C_{Ty} = -C_{Ty,\alpha} \alpha - C_{Ty,\delta_e} \delta_e \quad (7.72)$$

The moment arm is the wing chord c (m).

The coefficient for the roll torque about the x-axis is a linear function of the aileron command δ_a (-1 to 1):

$$C_{Tx} = C_{\delta_a} \delta_a \quad (7.73)$$

The moment arm is the wing span S (m).

The coefficient for the yaw torque around the z-axis is a linear function of the side-slip angle β and the rudder command δ_r (-1 to 1):

$$C_{Tz} = C_{Tz,\beta} \beta + C_{Tz,\delta_r} \delta_r \quad (7.74)$$

The moment arm is the length L of the airplane from nose to tail.

Forces and Torques from Aerodynamics

The airspeed is a function of the body-fixed relative velocities:

$$V_a = \sqrt{u_r^2 + v_r^2 + w_r^2}$$

The angle of attack α (rad) is a function of the relative air speeds w_r and u_r (m/s) and is calculated as follows:

$$\alpha = \tan^{-1} \left(\frac{w_r}{V_a} \right)$$

The side-slip angle is

$$\beta = \sin^{-1} \left(\frac{v_r}{V_a} \right)$$

Aerodynamic forces are calculated by

7.9 Aerodynamics of Fixed-Wing Aircraft

$$f_{\text{aero}} = \frac{1}{2} \rho V_a^2 A$$

where A is the wing area (m^2), and ρ (kg/m^3) is the air density.

The lift force in the body-fixed z-direction is

$$f_{z,L} = f_{\text{aero}} \left(- \left(C_{L,0} + \frac{C_{L,1}}{C_{L,2} + e^{-C_{L,3}(\alpha+C_{L,4})}} \right) - C_{L,\delta_e} \delta_e \right)$$

The drag force in the body-fixed x-direction is

$$f_{x,D} = f_{\text{aero}} (- (C_{D,0} + C_{D,2}\alpha^2) - C_{D,\delta_e} \delta_e)$$

The side-slip force in the y-direction is

$$f_{y,\beta} = f_{\text{aero}} (-C_{y,\beta}\beta - C_{y,\delta_r} \delta_r)$$

The pitching torque about the y-axis is

$$T_y = f_{\text{aero}} c (-C_{Ty,\alpha}\alpha - C_{Ty,\delta_e} \delta_e) - b_q q$$

where c is the wing chord. The roll torque about the x-axis is

$$T_{x,\text{aero}} = f_{\text{aero}} S C_{\delta_a} \delta_a - b_p p$$

where S is the wing span. The yaw torque about the z-axis is

$$T_z = f_{\text{aero}} c (C_{Tz,\beta}\beta + C_{Tz,\delta_r} \delta_r) - b_r r$$

Combined Forces and Torques in the Body-Fixed Directions

$$f_x = f_{x,T} + f_{x,D} + f_{x,g}$$

$$f_y = f_{y,\beta} + f_{y,g}$$

$$f_z = f_{z,L} + f_{z,g}$$

$$T_x = T_{x,\text{aero}} + T_{x,Q}$$

Chapter 8

Airplanes, Microcontrollers, and Sensors

Contents

8.1	Boomerang Warbler	255
8.2	Flight Hardware Bill-of-Materials	259
8.3	Reading Measurements from the IMU Sensor	260
8.3.1	Magnetometer Calibration	265
8.4	Reading Measurements from the BMP280 Sensor	273
8.5	Reading Measurements from the GPS Sensor	274
8.6	Interpreting GPS Data	276
8.6.1	Interpreting Latitude and Longitude	279
8.6.2	Latitude and Longitude to Distance in Meters	280
8.6.3	Converting Distances to Latitude and Longitude	282
8.7	GPS Displacement Algorithm	282
8.7.1	Testing the GPS Displacement Algorithm	283
8.7.2	GPS Velocity Algorithm	284

This chapter introduces airplane designs, microcontrollers, and sensors. It offers initial experiments to verify that the sensors are producing expected signals. It discusses calibrating the sensors and filtering raw signals to produce cleaner data. Sensor setup and calibration is essential for successful flight monitoring and autonomous control.

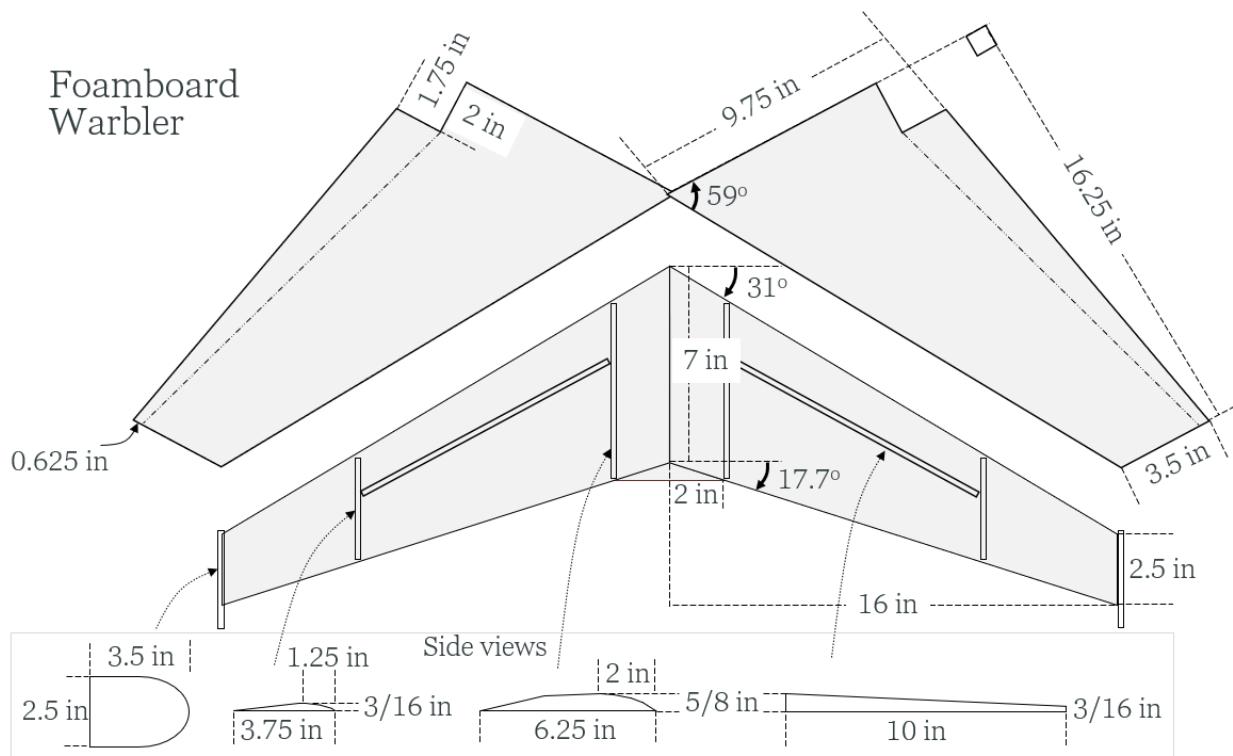
8.1 Boomerang Warbler

The **Boomerang Warbler**, see Figure 8.1, is a Warbler airplane autonomously controlled by a Boomerang flight controller. It weighs about 190 g (its maximum weight must be less than 250 g). The Warbler is a flying wing airplane made from foamboard with the paper removed. The foamboard pieces and dimensions for the Warbler are shown in Figure 8.2.

The Boomerang flight controller, see Figure 8.3, is a custom-built controller. It consists of a Raspberry Pi Pico RP2040 microcontroller, a TDK InvenSense ICM-20948 9DOF inertial measurement unit (IMU), a Bosch Sensortec BMP280 pressure sensor, a micro-SD card adapter, four servo-motor / ESC motor



Figure 8.1 The Boomerang Warbler



To reduce weight, all paper should be removed from the foamboard

Figure 8.2 Dimensions for the Warbler airplane

controller connectors, a micro-USB port, and a NEO-6M GPS breakout module by GOOUUU Tech. Any software used to program a Raspberry Pi Pico can also program the Boomerang microcontroller. In fact, during development, the Boomerang was originally made from a Raspberry Pi Pico connected to various breakout modules and sensors using a custom PCB, see Figure 8.4. The wiring diagram for the Boomerang flight controller is shown in Figure 8.5.

8.1 Boomerang Warbler

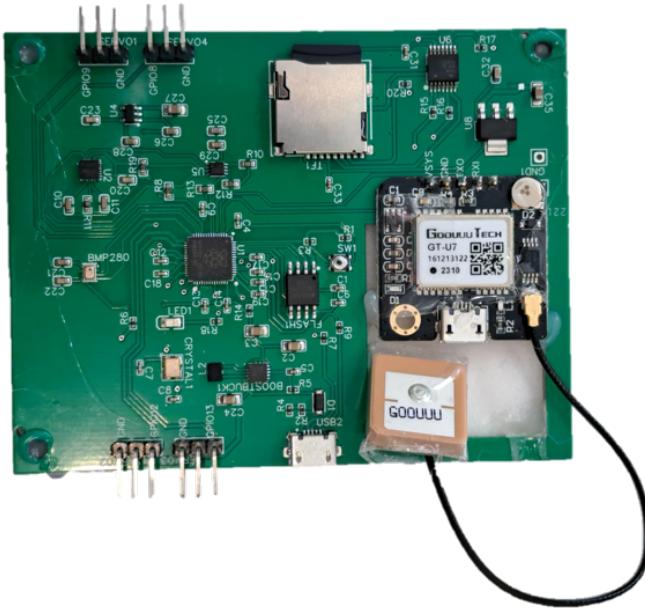


Figure 8.3 The Boomerang flight controller

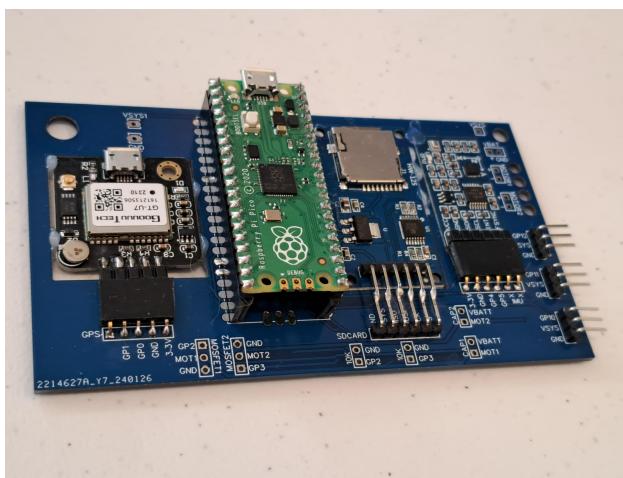


Figure 8.4 A modular version of the Boomerang flight controller during development

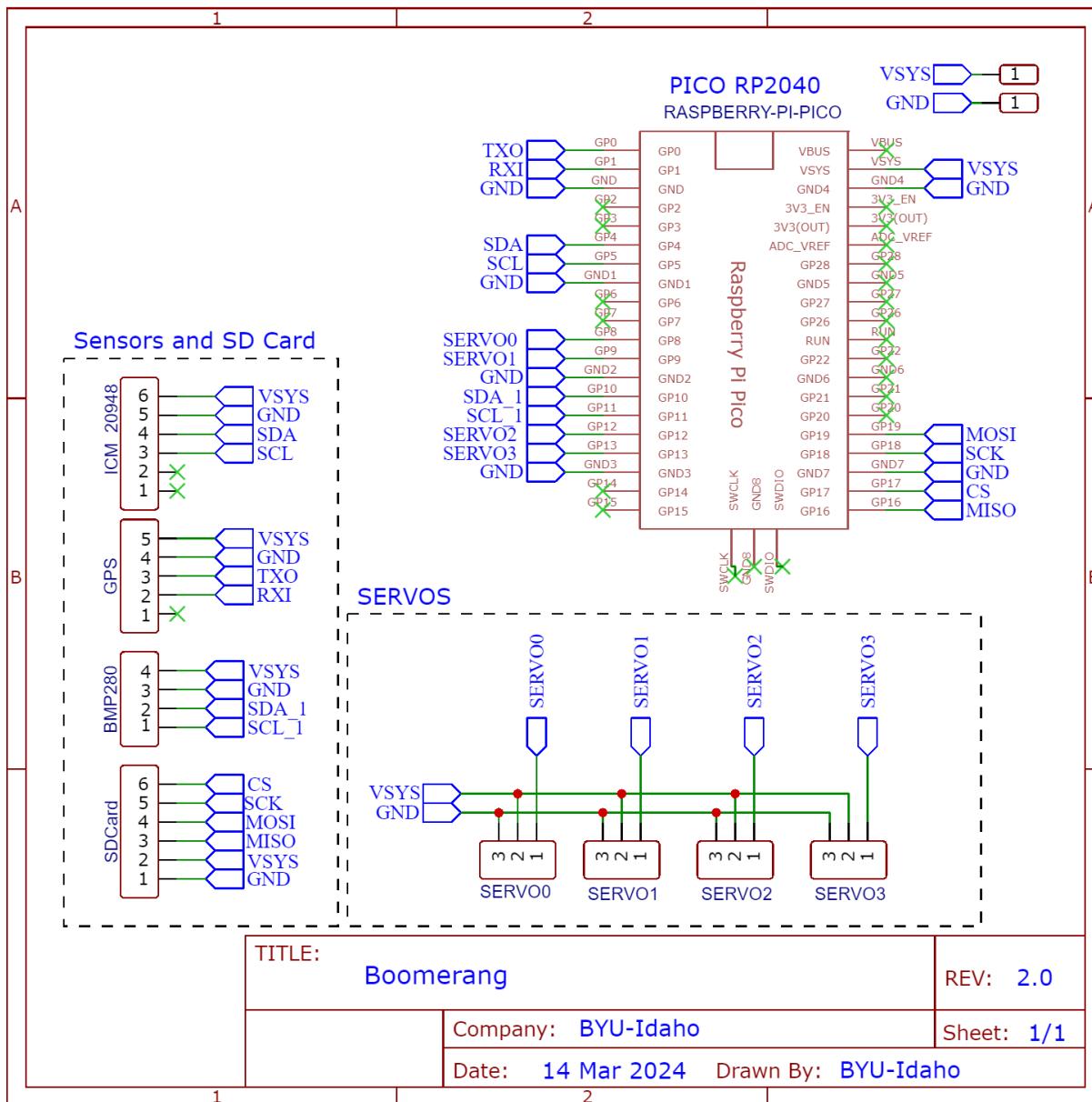


Figure 8.5 Wiring diagram for the Boomerang flight controller

8.2 Flight Hardware Bill-of-Materials

8.2 Flight Hardware Bill-of-Materials

The required materials for the flight hardware are listed below:

1. One Boomerang microcontroller, see Figure 8.3
2. One 20×30 inch sheet of foamboard
3. One NEO-6M GPS module, see Figure 8.9
4. Two 9 g servo motors with metal gears
5. Two 1 mm diameter servo rods (push rods) and control horns
6. One toothpick drone brushless motor, size of about 1303-5000kv, (e.g., FliteTest Gremlin 1104-5400KV motor (Figure 8.6) or iFlight XING Nano 1303-5000kv)
7. One T2.5x2.5x3 (2.5×2.5-inch, 3-blade, T-type) propeller that fits the brushless motor, see Figure 8.6
8. One paint stir stick or other small (1.5 cm×3 cm×0.3 cm) piece of wood for a motor mount
9. One 12 A Electronic Speed Controller (ESC) with a 5 V Battery Eliminator Circuit (BEC), the connector should fit the LiPo battery
10. One 650 mAh, two-cell (2S), lithium polymer (LiPo) battery, the connector should fit the 12A ESC
11. A two-cell (2S) LiPo battery charger
12. One single row 40-pin 2.54 mm male bent pin header
13. One single row 40-pin 2.54 mm female pin header
14. One micro-SD card
15. A USB-A to Micro USB cable to program the flight controller
16. A soldering iron and solder
17. A hot glue gun with four or more hot glue sticks
18. One roll of clear packaging tape
19. A razor blade for cutting the foamboard and pin headers
20. A drill with 3/16" and 5/64" bits to drill holes for the wooden motor mount
21. Heat-shrink tubing if needed to cover soldered joints between the motor and the ESC



Figure 8.6 The FliteTest Gremlin motor with a T2.5x2.5x3 propeller

8.3 Reading Measurements from the IMU Sensor

Using a microcontroller to read data from sensors is often made easier by using libraries created by others that are available to the community. This section provides an example of how to use Arduino libraries to read gyrometer, accelerometer, magnetometer, and temperature data from an InvenSense ICM-20948 inertial measurement sensor. The InvenSense ICM-20948 9dof sensor includes a 3-axis magnetometer, accelerometer, and gyrometer.

Wolfgang Ewald created an Arduino library, ICM20948_WE, for the ICM-20948 sensor. This section uses version 1.1.6. The code below is an Example Sketch, ICM20948_05_acc_gyr_temp_mag_data. It provides code to setup and calibrate the sensor, read accelerometer, gyrometer, temperature, and magnetometer data from the sensor, and write the results to Arduino's Serial Monitor. The comments in the code explain how the library is used to read measurements from the sensor. .

```

/*
 * Example sketch for the ICM20948_WE library
 *
 * This sketch shows how to retrieve accelerometer, gyroscope, temperature
 * and magnetometer data from the ICM20948.
 *
 * Further information can be found on:
 *
 * https://wolles-elektronikkiste.de/icm-20948-9-achsensensor-teil-i (German)
 * https://wolles-elektronikkiste.de/en/icm-20948-9-axis-sensor-part-i (English)
 *
 */
#include <Wire.h>
#include <ICM20948_WE.h>
#define ICM20948_ADDR 0x68

/* There are several ways to create your ICM20948 object:
 * ICM20948_WE myIMU = ICM20948_WE()           -> uses Wire / I2C Address = 0x68
 * ICM20948_WE myIMU = ICM20948_WE(ICM20948_ADDR) -> uses Wire / ICM20948_ADDR

```

8.3 Reading Measurements from the IMU Sensor

```
* ICM20948_WE myIMU = ICM20948_WE(&wire2) -> uses the TwoWire object wire2 / ICM20948_ADDR
* ICM20948_WE myIMU = ICM20948_WE(&wire2, ICM20948_ADDR) -> all together
* ICM20948_WE myIMU = ICM20948_WE(CS_PIN, spi);-> uses SPI, see SPI example
* ICM20948_WE myIMU = ICM20948_WE(&SPI, CS_PIN, spi);-> uses SPI / passes the SPI object, spi
*/
ICM20948_WE myIMU = ICM20948_WE(ICM20948_ADDR);

void setup() {
    Wire.begin();
    Serial.begin(115200);
    while(!Serial) {}

    if(!myIMU.init()){
        Serial.println("ICM20948 does not respond");
    }
    else{
        Serial.println("ICM20948 is connected");
    }

    if(!myIMU.initMagnetometer()){
        Serial.println("Magnetometer does not respond");
    }
    else{
        Serial.println("Magnetometer is connected");
    }

    /* Check ICM20948 */
    // byte reg = myIMU.whoAmI();
    //Serial.println(reg);

    /***** Basic Settings *****/
    /*
     * This is a method to calibrate. You have to determine the minimum and maximum
     * raw acceleration values of the axes determined in the range +/- 2 g.
     * You call the function as follows: setAccOffsets(xMin,xMax,yMin,yMax,zMin,zMax);
     * The parameters are floats.
     * The calibration changes the slope / ratio of raw acceleration vs g. The zero point is
     * set as (min + max)/2.
     */
    //myIMU.setAccOffsets(-16330.0, 16450.0, -16600.0, 16180.0, -16520.0, 16690.0);

    /*
     * The starting point, if you position the ICM20948 flat,
     * is not necessarily 0g/0g/1g for x/y/z.
     * The autoOffset function measures offset. It assumes your
     * ICM20948 is positioned flat with its
     * x,y-plane. The more you deviate from this, the less accurate will be your results.
     * It overwrites the zero points of setAccOffsets, but keeps the correction of the slope.
     * The function also measures the offset of the gyroscope data.
     * The gyroscope offset does not
     * depend on the positioning.
     * This function needs to be called after setAccOffsets but before other
     * settings since it will overwrite settings!
     */
}
```

```

// Serial.println("Position your ICM20948 flat and don't move it - calibrating... ");
// delay(1000);
// myIMU.autoOffsets();
// Serial.println("Done!");

/* The gyroscope data is not zero, even if you don't move the ICM20948.
 * To start at zero, you can apply offset values. These are the gyroscope raw
 * values you obtain using the +/- 250 degrees/s range.
 * Use either autoOffset or setGyrOffsets, not both.
 */
//myIMU.setGyrOffsets(-115.0, 130.0, 105.0);

/* ICM20948_ACC_RANGE_2G      2 g    (default)
 * ICM20948_ACC_RANGE_4G      4 g
 * ICM20948_ACC_RANGE_8G      8 g
 * ICM20948_ACC_RANGE_16G     16 g
 */
myIMU.setAccRange(ICM20948_ACC_RANGE_2G);

/* Choose a level for the Digital Low Pass Filter or switch it off.
 * ICM20948_DLPF_0, ICM20948_DLPF_2, ..... ICM20948_DLPF_7, ICM20948_DLPF_OFF
 *
 * IMPORTANT: This needs to be ICM20948_DLPF_7 if DLPF is used in cycle mode!
 *
 * DLPF      3dB Bandwidth [Hz]      Output Rate [Hz]
 * 0          246.0                 1125/(1+ASRD)
 * 1          246.0                 1125/(1+ASRD)
 * 2          111.4                 1125/(1+ASRD)
 * 3          50.4                  1125/(1+ASRD)
 * 4          23.9                  1125/(1+ASRD)
 * 5          11.5                  1125/(1+ASRD)
 * 6          5.7                  1125/(1+ASRD)
 * 7          473.0                 1125/(1+ASRD)
 * OFF        1209.0                4500
 *
 * ASRD = Accelerometer Sample Rate Divider (0...4095)
 * You achieve lowest noise using level 6
 */
myIMU.setAccDLPF(ICM20948_DLPF_6);

/* Acceleration sample rate divider divides the output rate of the accelerometer.
 * Sample rate = Basic sample rate / (1 + divider)
 * It can only be applied if the corresponding DLPF is not off!
 * Divider is a number 0...4095 (different range compared to gyroscope)
 * If sample rates are set for the accelerometer and the gyroscope, the gyroscope
 * sample rate has priority.
 */
//myIMU.setAccSampleRateDivider(10);

/* ICM20948_GYRO_RANGE_250      250 degrees per second (default)
 * ICM20948_GYRO_RANGE_500      500 degrees per second
 * ICM20948_GYRO_RANGE_1000     1000 degrees per second
 * ICM20948_GYRO_RANGE_2000     2000 degrees per second

```

8.3 Reading Measurements from the IMU Sensor

```
/*
//myIMU.setGyrRange(ICM20948_GYRO_RANGE_250);

/* Choose a level for the Digital Low Pass Filter or switch it off.
* ICM20948_DLDPF_0, ICM20948_DLDPF_2, ..... ICM20948_DLDPF_7, ICM20948_DLDPF_OFF
*
* DLPP      3dB Bandwidth [Hz]      Output Rate [Hz]
* 0          196.6                 1125/(1+GSRD)
* 1          151.8                 1125/(1+GSRD)
* 2          119.5                 1125/(1+GSRD)
* 3          51.2                  1125/(1+GSRD)
* 4          23.9                  1125/(1+GSRD)
* 5          11.6                  1125/(1+GSRD)
* 6          5.7                  1125/(1+GSRD)
* 7          361.4                 1125/(1+GSRD)
* OFF        12106.0                9000
*
* GSRD = Gyroscope Sample Rate Divider (0...255)
* You achieve lowest noise using level 6
*/
myIMU.setGyrDLPP(ICM20948_DLDPF_6);

/* Gyroscope sample rate divider divides the output rate of the gyroscope.
* Sample rate = Basic sample rate / (1 + divider)
* It can only be applied if the corresponding DLPP is not OFF!
* Divider is a number 0...255
* If sample rates are set for the accelerometer and the gyroscope, the gyroscope
* sample rate has priority.
*/
//myIMU.setGyrSampleRateDivider(10);

/* Choose a level for the Digital Low Pass Filter.
* ICM20948_DLDPF_0, ICM20948_DLDPF_2, ..... ICM20948_DLDPF_7, ICM20948_DLDPF_OFF
*
* DLPP      Bandwidth [Hz]      Output Rate [Hz]
* 0          7932.0                9
* 1          217.9                 1125
* 2          123.5                 1125
* 3          65.9                  1125
* 4          34.1                  1125
* 5          17.3                  1125
* 6          8.8                  1125
* 7          7932.0                9
*
*
* GSRD = Gyroscope Sample Rate Divider (0...255)
* You achieve lowest noise using level 6
*/
myIMU.setTempDLPP(ICM20948_DLDPF_6);

/* You can set the following modes for the magnetometer:
* AK09916_PWR_DOWN          Power down to save energy
* AK09916_TRIGGER_MODE       Measurements on request, a measurement is triggered by
```

```

/*
 *          calling setMagOpMode(AK09916_TRIGGER_MODE)
 * AK09916_CONT_MODE_10HZ   Continuous measurements, 10 Hz rate
 * AK09916_CONT_MODE_20HZ   Continuous measurements, 20 Hz rate
 * AK09916_CONT_MODE_50HZ   Continuous measurements, 50 Hz rate
 * AK09916_CONT_MODE_100HZ  Continuous measurements, 100 Hz rate (default)
 */
myIMU.setMagOpMode(AK09916_CONT_MODE_20HZ);

}

void loop() {
    myIMU.readSensor();
    xyzFloat gValue = myIMU.getGValues();
    xyzFloat gyr = myIMU.getGyrValues();
    xyzFloat magValue = myIMU.getMagValues();
    float temp = myIMU.getTemperature();
    float resultantG = myIMU.getResultantG(gValue);

    Serial.println("Acceleration in g (x,y,z):");
    Serial.print(gValue.x);
    Serial.print(" ");
    Serial.print(gValue.y);
    Serial.print(" ");
    Serial.println(gValue.z);
    Serial.print("Resultant g: ");
    Serial.println(resultantG);

    Serial.println("Gyroscope data in degrees/s: ");
    Serial.print(gyr.x);
    Serial.print(" ");
    Serial.print(gyr.y);
    Serial.print(" ");
    Serial.println(gyr.z);

    Serial.println("Magnetometer Data in uTesla: ");
    Serial.print(magValue.x);
    Serial.print(" ");
    Serial.print(magValue.y);
    Serial.print(" ");
    Serial.println(magValue.z);

    Serial.print("Temperature in C: ");
    Serial.println(temp);

    Serial.println("*****");
    delay(1000);
}

```

8.3 Reading Measurements from the IMU Sensor

8.3.1 Magnetometer Calibration

Hard iron and soft iron disturbances cause errors in magnetometer measurements. **Hard iron** distortions are caused by local magnetic sources, such as permanent magnets or electromagnetic coils. Hard iron effects are common in electric aircraft because motors can have both permanent magnets and electromagnets. **Soft iron** disturbances are caused by ferromagnetic materials such as iron or nickel that can distort or deflect a magnetic field. Nuts and bolts, motor casing and shafts, or other local metal parts can cause soft iron distortions. Magnetometer calibration can help minimize the effect of these distortions.

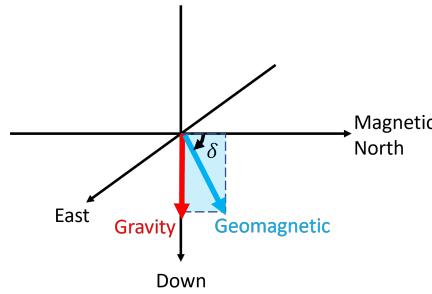


Figure 8.7 Gravitational and geomagnetic vectors in the NED reference frame

In the inertial frame, the geomagnetic field is a constant vector pointing in a single direction, see Figure 8.7. However, in the rotating body-frame, the magnitude is constant, but the direction changes. Therefore, in the body-frame, undistorted geomagnetic measurements would lie on the surface of a sphere centered around the origin (0,0,0) with a radius equal to the geomagnetic field strength B . Hard iron disturbances shift the entire geomagnetic field away from the (0,0,0) origin. Soft iron disturbances deform the sphere into an ellipsoid. These effects can be modeled by the following equation:

$$M_d = K^{-1} M + \beta \quad (8.1)$$

M_d is the raw magnetometer signal. It is the distorted magnetometer reading. M is the true, but unknown, undistorted geomagnetic vector from the perspective of the body-frame:

$$M = \begin{bmatrix} M_x \\ M_y \\ M_z \end{bmatrix} \quad (8.2)$$

The hard iron effect β is the amount the measurement has been shifted from the (0,0,0) origin:

$$\beta = \begin{bmatrix} \beta_x \\ \beta_y \\ \beta_z \end{bmatrix} \quad (8.3)$$

The soft iron effect is due to the symmetric distortion matrix K :

$$K = \begin{bmatrix} k_{11} & k_{12} & k_{13} \\ k_{12} & k_{22} & k_{23} \\ k_{13} & k_{23} & k_{33} \end{bmatrix} \quad (8.4)$$

Calibration of the magnetometer signal involves calculating both the offset β and distortion matrix K . If these are known, the undistorted geomagnetic vector is calculated by

$$M = K(M_d - \beta) \quad (8.5)$$

Determining β and K is done by first collecting many magnetometer measurements while the body is rotating. Finding the centroid of the measurements determines β . The distortion matrix is found by fitting the equation for an ellipsoid to the data.

There are many techniques to find the centroid β . One way is to iteratively find the average radius of the geomagnetic sphere and move β so that each point is as close as possible to the surface of the sphere. The MATLAB algorithm below uses this approach to find the centroid β .

There are also many methods to find the distortion matrix K . The method in the MATLAB algorithm below uses a steepest descent optimization algorithm. To improve effectiveness, the MATLAB algorithm below down-samples the data to have a more uniform distribution around the magnetometer ellipsoid. The result of the algorithm on the data provided at the end of this section is shown in the graph of Figure 8.8. The calibrated data is labeled “Best Fit”. The calibrated magnetometer data are more spherical (better fit to the surface of a sphere) than the original and are centered around the origin (0,0,0).

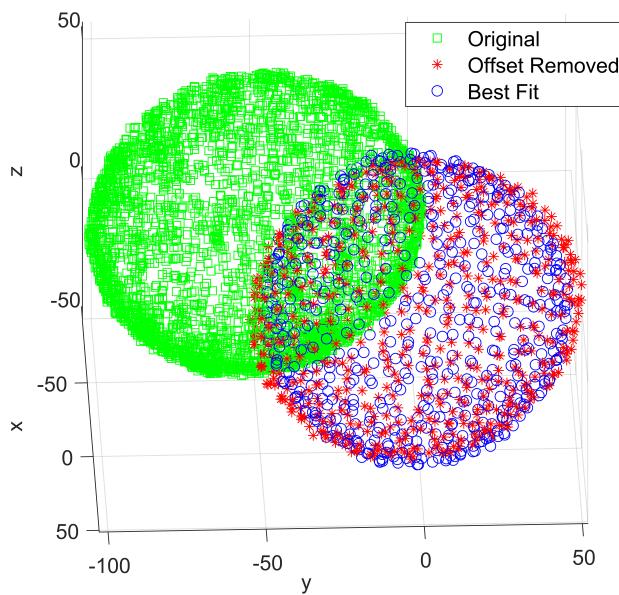


Figure 8.8 Performance of the MATLAB magnetometer calibration algorithm on actual magnetometer data

MATLAB algorithm to calibrate a magnetometer:

8.3 Reading Measurements from the IMU Sensor

```
close all
clear all
clc

%% import the data from a .csv file
[matlabFile,path] = uigetfile('*.csv', 'Select The IMU data');
mData = readtable([path,matlabFile]);

%% extract the magnetometer data
xM = mData.Bx; %x-axis magnetometer data
yM = mData.By; %y-axis magnetometer data
zM = mData.Bz; %z-axis magnetometer data

%remove zeros from the data
Na = length(xM);
xMag = [ ]; %empty array to store nonzero x-axis magnetometer data
yMag = [ ]; %empty array to store nonzero y-axis magnetometer data
zM = [ ]; %empty array to store nonzero z-axis magnetometer data
for ii = 1:Na
    if xM(ii) ~= 0 && yM(ii) ~= 0 && zM(ii) ~= 0
        %only populate with nonzero data
        xMag = [xMag;xM(ii)]; %nonzero x-axis magnetometer data
        yMag = [yMag;yM(ii)]; %nonzero y-axis magnetometer data
        zMag = [zM;zM(ii)]; %nonzero z-axis magnetometer data
    end
end

%% downsample the data to make it more uniform
% Data are separated by at least the threshold distance
threshold = 10;
[Kept,Remainder] = function_Downsample(xMag,yMag,zMag, threshold);
XYZdata = Kept;

%% Find the offset (beta) of the data
[bx,by,bz,Ravg] = function_FitSphereBeta(XYZdata);
beta = [bx,by,bz];

%% remove major outliers
XYZdata = removeMajorOutliers(XYZdata, beta);

%% Find the distortion matrix (K) of the data
K = function_FitSphereK(XYZdata, beta);
%remove the bias
x = XYZdata(:,1)-bx;
y = XYZdata(:,2)-by;
z = XYZdata(:,3)-bz;
%rotate and scale the data
xyz = K * [x';y';z'];

% remove outliers
xM = XYZdata(:,1);
yM = XYZdata(:,2);
zM = XYZdata(:,3);
xN = [];
```

```

yN = [] ;
zN = [] ;
for ii = 1:length(xM)
    vec = xyz(:,ii);
    if abs(sqrt(vec'*vec)-Ravg) < 5
        xN = [xN;xM(ii)];
        yN = [yN;yM(ii)];
        zN = [zN;zM(ii)];
    end
end
XYZdata = [xN,yN,zN];

% Find beta again, but with outliers removed
[bx,by,bz,~] = function_FitSphereBeta(XYZdata);
beta = [bx,by,bz]

%% Find the distortion matrix (K) of the data
K = function_FitSphereK(XYZdata, beta)
%remove the bias
x = XYZdata(:,1)-bx;
y = XYZdata(:,2)-by;
z = XYZdata(:,3)-bz;
%rotate and scale the data
xyz = K * [x';y';z'];

% plot the results
figure
scatter3(xMag,yMag,zMag,'gs')
hold on
scatter3(XYZdata(:,1)-bx,XYZdata(:,2)-by,XYZdata(:,3)-bz,'r*')
scatter3(xyz(1,:),xyz(2,:),xyz(3,:),'bo')
legend('Original','Offset Removed','Best Fit')
xlabel('x')
ylabel('y')
zlabel('z')
axis('equal')

%% Function to get the downsampled data
function [Kept,Remainder] = function_Downsample(xMag,yMag,zMag, threshold)
    % get the length of the data
    N = length(xMag);
    %Get the first point that will be kept
    ind = randi(N);
    Kept = [xMag(ind),yMag(ind),zMag(ind)];
    Remainder = [];
    for ii = 1:N
        nXYZ = size(Kept,1); %get the length of the kept-data array
        flag = 0; %0 indicates that the current (ii) magnetometer data is not
                   %within the threshold distance to any other kept data
        for jj = 1:nXYZ
            %check if the current (ii) magnetometer data is within the
            %threshold distance to any other kept data
            if sqrt((xMag(ii)-Kept(jj,1))^2+...
                     (yMag(ii)-Kept(jj,2))^2+...
                     (zMag(ii)-Kept(jj,3))^2) < threshold

```

8.3 Reading Measurements from the IMU Sensor

```
%the data is too close to existing kept data. Remove it.
flag = 1;
break;
end
end
if ~flag
    %The data was not too close to existing kept data. Keep it.
    Kept = [Kept;[xMag(ii),yMag(ii),zMag(ii)]];
else
    %The data was too close to keep
    Remainder = [Remainder; [xMag(ii),yMag(ii),zMag(ii)]];
end
end
end

%% function to calculate the offset beta
function [bx,by,bz,Ravg] = function_FitSphereBeta(XYZdata)
    %Get the total number of magnetometer readings
    N = length(XYZdata); %Number of magnetometer data points
    %Get a rough prediction of the location of the sphere's centroid for
    % an initial condition
    beta = [mean(XYZdata(:,1)),mean(XYZdata(:,2)),mean(XYZdata(:,3))];
    for jj = 1:2000
        % find the average radius
        Ravg = 0;
        for ii = 1:N
            %get the vector from the centroid (beta) to the data point
            vec = XYZdata(ii,:)-beta;
            %assign a distance from the centroid to each XYZdata point
            R = sqrt(vec*vec');
            %update the average radius
            Ravg = Ravg + R/N;
        end
        %move the center slightly towards the point that will minimize
        % distance - Ravg
        for ii = 1:N
            %get the vector from the centroid (beta) to the data point
            vec = XYZdata(ii,:)-beta;
            %assign a distance from the centroid to each XYZdata point
            R = sqrt(vec*vec');
            %get the unit vector
            u = vec / norm(vec);
            %step in the direction that would minimize R-Ravg
            beta = beta + 0.001*u*(R-Ravg);
        end
    end

    %output the offset values
    bx = beta(1); %magnetometer x-axis offset
    by = beta(2); %magnetometer y-axis offset
    bz = beta(3); %magnetometer z-axis offset
end

%% function to remove major outliers
```

```

function Mn = removeMajorOutliers(M, beta)
%get the standard deviation in the radius
N = length(M);
Rvec = zeros(N,1);
for ii = 1:N
    %get the vector from the centroid (beta) to the data point
    vec = M(ii,:)-beta;
    %assign a distance from the centroid to each XYZdata point
    Rvec(ii) = sqrt(vec*vec');
end
Ravg = mean(Rvec);
Rstd = std(Rvec);

%remove data that is more than 3 standard deviations away from the mean
Mn = [];
for ii = 1:N
    %get the vector from the centroid (beta) to the data point
    vec = M(ii,:)-beta;
    %assign a distance from the centroid to each XYZdata point
    R = sqrt(vec*vec');
    if (R < Ravg + 3 * Rstd) && ...
        (R > Ravg - 3*Rstd)
        %Keep it
        Mn = [Mn;M(ii,:)];
    end
end
end

%% Get the distortion matrix K
function K = function_FitSphereK(XYZdata, beta)
%Get the total number of magnetometer readings
N = length(XYZdata); %Number of magnetometer data points
%remove the bias
bx = beta(1);
by = beta(2);
bz = beta(3);
x = XYZdata(:,1)-bx;
y = XYZdata(:,2)-by;
z = XYZdata(:,3)-bz;
xyz = [x';y';z'];
% find the average radius
Ravg = 0;
for ii = 1:N
    %get the vector from the centroid (beta) to the data point
    vec = xyz(:,ii)';
    %assign a distance from the centroid to each XYZdata point
    R = sqrt(vec*vec');
    %update the average radius
    Ravg = Ravg + R/N;
end
%Set the initial six K values to the identity matrix
k11=1; k22=1; k33=1; k12=0; k13=0; k23=0;
Kvec = [k11;k12;k13;k22;k23;k33];
mu = 2e-10; %Steepest descent step size
for kk = 1:1000

```

8.3 Reading Measurements from the IMU Sensor

```
for ii = 1:N
    %get the vector from the centroid (beta) to the data point
    vec = xyz(:,ii)';
    %get M'*M
    kv1 = Kvec(1)*vec(1)+Kvec(2)*vec(2)+Kvec(3)*vec(3);
    kv2 = Kvec(2)*vec(1)+Kvec(4)*vec(2)+Kvec(5)*vec(3);
    kv3 = Kvec(3)*vec(1)+Kvec(5)*vec(2)+Kvec(6)*vec(3);
    MM = kv1^2 + kv2^2 + kv3^2;
    %get M'*M-Ravg^2, which is related to the step size
    MR = MM - Ravg^2;
    %Get the steepest descent direction
    J = [vec(1)*kv1;...
          vec(2)*kv1+vec(2)*kv2;...
          vec(3)*kv1+vec(1)*kv3;...
          vec(2)*kv2;...
          vec(3)*kv2+vec(2)*kv3;...
          vec(3)*kv3];
    %Take a small step in the direction of the steepest descent
    Kvec = Kvec - mu * MR * J;
end
end
K = [Kvec(1),Kvec(2),Kvec(3);...
      Kvec(2),Kvec(4),Kvec(5);
      Kvec(3),Kvec(5),Kvec(6)];
end
```

Below are some actual magnetometer data to test the algorithm:

mag0	mag1	mag2
-35.9	-38.3	-32.2
-33.1	-26.3	-27.4
-47.3	-24.1	-29.7
-22.1	-43.5	-30.4
-9.5	-37.3	-24.8
0.1	-37.1	-15.3
4.3	-32.0	-2.3
2.8	-26.4	11.1
-1.4	-24.6	25.5
-12.3	-20.9	35.0
-24.7	-20.2	44.4
-38.0	-21.1	49.0
-53.1	-26.1	50.2
-65.6	-27.6	45.8
-78.2	-28.4	36.3
-85.3	-27.8	24.5
-85.9	-24.8	8.9
-78.6	-21.1	-4.5
-70.6	-21.8	-15.9
-21.4	-21.6	-21.3
-8.1	-24.6	-14.6
-61.2	-31.7	-25.9
-2.1	-34.5	35.0
-13.4	-35.7	44.4

Chapter 8 Airplanes, Microcontrollers, and Sensors

-25.5	-37.4	51.1
-39.9	-41.1	53.6
-56.3	-43.7	53.1
-71.1	-42.9	46.5
-82.5	-43.4	36.8
-89.0	-40.2	19.7
-89.6	-38.1	6.4
-81.8	-37.3	-11.0
-73.3	-40.9	-21.3
-58.8	-45.3	-30.4
-44.0	-50.4	-34.4
-29.2	-54.9	-33.5
-9.0	-52.5	-26.6
2.3	-52.3	-16.8
9.2	-47.5	-2.7
9.6	-42.6	11.5
-71.2	-53.8	-23.7
-57.0	-61.0	-29.7
-19.9	-64.9	-30.4
-3.7	-63.3	-20.5
7.5	-61.2	-8.1
10.7	-59.0	6.5
9.6	-56.0	20.7
2.7	-56.0	32.2
-6.8	-55.4	42.7
-18.6	-58.0	49.3
-37.1	-59.1	53.4
-53.1	-59.4	53.0
-71.1	-58.4	46.1
-88.3	-53.8	26.6
-90.5	-55.8	8.0
-83.8	-60.8	-6.7
-74.7	-66.0	-19.2
-47.1	-71.2	-30.5
-30.5	-72.5	-30.7
-26.6	-70.0	49.9
6.1	-38.6	24.3
-56.0	-14.2	41.5
-69.4	-17.5	37.4
-2.8	-20.6	-1.7
-77.9	-15.6	23.4
3.7	-72.4	24.6
-8.3	-83.4	29.8
-21.7	-88.0	34.8
-40.3	-92.9	34.3
-56.6	-92.5	28.5
-71.0	-88.0	19.4
-71.9	-87.4	4.5
-59.9	-93.1	-4.7
-47.8	-94.6	-9.9
-33.8	-92.5	-13.9
-20.1	-88.8	-17.1
-7.6	-87.2	-9.7
-1.2	-85.1	2.6
-61.5	-96.0	14.7

8.4 Reading Measurements from the BMP280 Sensor

-21.9	-97.3	-5.9
-12.0	-95.0	2.6
-4.0	-87.7	18.2
-8.3	-78.2	-19.7
-75.7	-69.5	36.5
-81.8	-71.8	24.3
-84.5	-72.6	11.0
-79.1	-74.3	-4.1
7.6	-73.4	11.2
-38.8	-75.6	48.6
-53.3	-73.0	46.8
-64.2	-82.6	-14.4
-49.7	-85.9	-19.4
-32.5	-83.2	-22.9
-60.8	-81.0	39.7
-16.4	-14.6	-10.1
-32.5	-85.1	41.6
-39.2	-100.9	17.9
-54.7	-16.9	-20.2
-44.2	-4.0	-8.2
-31.5	-0.5	3.0
-38.5	2.3	13.6
-38.6	-3.1	32.4
-32.9	-101.3	2.7
-60.0	-10.2	-10.3
-16.3	-5.3	7.9
-7.8	-14.0	20.9
-3.3	-69.1	36.9
-26.5	-98.3	22.5
-51.2	0.8	6.2
-40.7	-14.0	-20.4
-15.9	-76.6	41.2
-47.5	-100.4	7.3
-20.6	-7.2	25.7
-28.3	-11.3	-15.0
-87.6	-47.6	-3.1
3.0	-74.5	-6.7
-71.3	-5.5	15.9
-60.0	-1.6	22.1
-76.6	-10.7	4.2
-64.4	-4.4	1.0
-28.3	-10.3	36.6

8.4 Reading Measurements from the BMP280 Sensor

The Arduino library BMP280_DEV is helpful for reading measurements from the BMP280 pressure and altitude sensor. This section used BMP280_DEV version 1.0.21 by Martin Lindupp. The BMP280_DEV library has an example sketch named BMP280_I2C_Normal.ino. The Boomerang flight controller, see Figure 8.3, uses the secondary I2C on the Raspberry Pi Pico microcontroller to communicate with the BMP280 sensor. Therefore, we must modify the example sketch to use Wire1 instead of the default I2C. The code that was modified is `bmp280.begin(Wire1);` in which `Wire1` was added as an argument to the

`begin()` function.

```
//////////  
// BMP280_DEV - I2C Communications, Default Configuration, Normal Conversion  
//////////  
  
#include <BMP280_DEV.h>    // Include the BMP280_DEV.h library  
  
float temperature, pressure, altitude; //Create the temperature, pressure, and altitude vars  
BMP280_DEV bmp280; // Create a BMP280_DEV object and set-up for I2C operation (address 0x77)  
  
void setup()  
{  
    Serial.begin(115200); // Initialize the serial port  
    bmp280.begin(Wire1); // Default initialization, place the BMP280 into SLEEP_MODE  
    //bmp280.setPresOversampling(OVERSAMPLING_X4); // Set the pressure oversampling to X4  
    //bmp280.setTempOversampling(OVERSAMPLING_X1); // Set the temperature oversampling to X1  
    //bmp280.setIIRFilter(IIR_FILTER_4);           // Set the IIR filter to setting 4  
    bmp280.setTimeStandby(TIME_STANDBY_2000MS); // Set the standby time to 2 seconds  
    bmp280.startNormalConversion(); // Start BMP280 continuous conversion in NORMAL_MODE  
}  
  
void loop()  
{  
    // Check if the measurement is complete  
    if (bmp280.getMeasurements(temperature, pressure, altitude))  
    {  
        Serial.print(temperature);                // Display the results  
        Serial.print(F("*C  "));  
        Serial.print(pressure);  
        Serial.print(F("hPa  "));  
        Serial.print(altitude);  
        Serial.println(F("m"));  
    }  
}
```

8.5 Reading Measurements from the GPS Sensor

This section uses the Arduino library TinyGPS, version 13.0.0 by Mikal Hart. It also uses the Arduino library SoftwareSerial.h for serial communication with the GPS module. The GPS sensor is a NEO 6M Arduino GPS module by GOOUUU TECH, see Figure 8.9. The following Arduino code reads latitude, longitude, altitude, and speed data from the GPS sensor.

```
#include <SoftwareSerial.h> //SoftwareSerial: GPS communication  
#include <TinyGPS.h>       //TinyGPS: GPS Latitude, Longitude, Altitude
```

8.5 Reading Measurements from the GPS Sensor

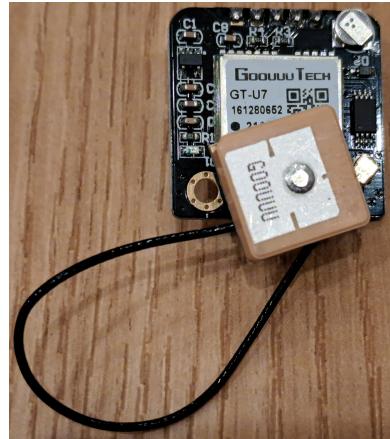


Figure 8.9 A NEO 6M Arduino GPS module by GOOUUU TECH.

```
//#####Variables for GPS#####
TinyGPS gps;
const int rxPin = 1;
const int txPin = 0;
SoftwareSerial ss(rxPin, txPin);
static void readGPS(unsigned long ms);
float GPSlat = 0.0; //declare GPS latitude and longitude variables
float GPSlon = 0.0;
float GPSspeed = 0.0; //Declare GPS speed (m/s)
float GPSAlt = 0.0; //Declare GPS altitude (m)
//#####End of Variables for GPS#####

void setup() {
    // Start serial communication
    Serial.begin(9600);
    delay(3000);
    //Start communication with the GPS module
    ss.begin(9600);
    unsigned long GPSdelay = 1000; //Time delay in ms
    readGPS(GPSdelay);
}

void loop() {
    double GPSspeed = 0.0; // (m/s)
    //Get GPS measurements
    unsigned long GPSdelay = 1; //Time delay in ms
    readGPS(GPSdelay);
    unsigned long age;
    //Read the latitude and longitude
    gps.f_get_position(&GPSlat, &GPSlon, &age);
    //Read the GPS altitude (m)
    GPSAlt = gps.f_altitude();
    //get the GPS speed (m/s)
    GPSspeed = (double)gps.f_speed_mps(); // (m/s)
```

```

//Print the results to the Serial Monitor
Serial.print("Latitude (deg): ");
Serial.println(GPSlat);
Serial.print("Longitude (deg): ");
Serial.println(GPSlon);
Serial.print("Altitude (m): ");
Serial.println(GPSAlt);
Serial.print("Speed (m/s): ");
Serial.println(GPSspeed);
Serial.println(" ");
delay(1000);
}

static void readGPS(unsigned long ms) {
    unsigned long start = millis();
    do {
        while (ss.available())
            gps.encode(ss.read());
    } while (millis() - start < ms);
}

```

8.6 Interpreting GPS Data

GPS (Global Positioning System) provides an accurate method for determining position with respect to an inertial reference frame. A GPS sensor, (see Figure 8.10) receives time-stamp information from GPS satellites orbiting the earth (see Figure 8.11). From this information, the GPS sensor's algorithms can determine its geographical location, usually specified in coordinates of latitude, longitude (see Figure 8.12), and altitude above sea level.



Figure 8.10 A GPS sensor by Adafruit (Adafruit Mini GPS PA1010D Module)

For example, the Adafruit_GPS.h library created for Arduino uses the command `lastNMEA()`. It returns a string containing an NMEA (National Marine Electronics Association) sentence. To understand the NMEA sentence, consider two NMEA example sentences. Each is for a GPS position located near the statue of a boy throwing an airplane (Boy in Flight) in front of the Mark Austin Building on the BYU-Idaho campus. The first is a GGA (Fix information) sentence for a GPS sensor:

8.6 Interpreting GPS Data

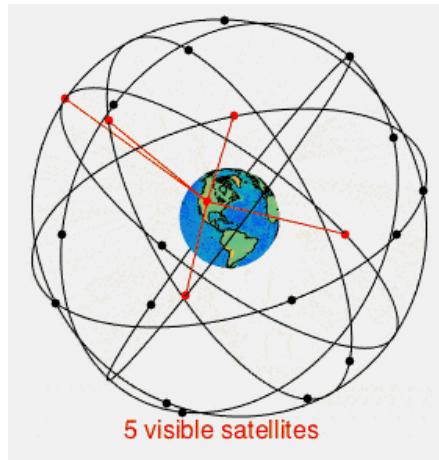


Figure 8.11 An example of a 24-satellite GPS constellation. Five satellites are visible to the GPS sensor. This image from https://en.wikipedia.org/wiki/Global_Positioning_System#/media/File:GPS24goldenSML.gif is licensed under the Creative Commons-Share Alike 4.0 International license.

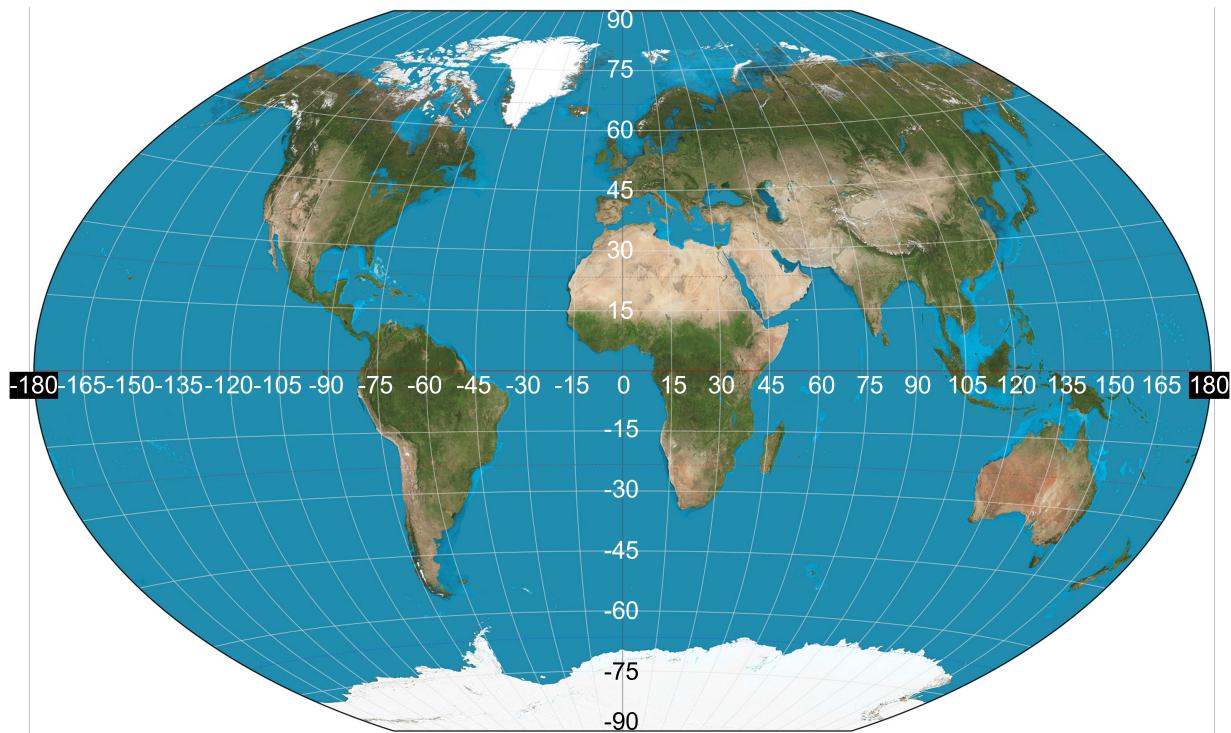


Figure 8.12 A world map with latitude and longitude coordinates. The original map by NASA is in the public domain (accessed June 2022 at https://en.wikipedia.org/wiki/File:Winkel_triple_projection_SW.jpg). The original image was modified to include values for latitude and longitude.

```
$GNGGA,154010.715,4348.9690,N,11147.0400,W,1,3,6.65,1669.0,M,-16.9,M,,*48
```

Where:

```

$          Begin sentence indicator
GN         Multi GNSS solution
GGA        Global Positioning System Fix Data
154010.715  Fix taken at 15:40:10.715 UTC (3:40 p.m.)
4348.9690,N Latitude 43 deg 48.9690' N
11147.0400,W Longitude 111 deg 47.0400' W (-111 deg 47.0400' E)
1          Fix quality: 0 = invalid
1 = GPS fix (SPS)
2 = DGPS fix
3 = PPS fix
4 = Real Time Kinematic
5 = Float RTK
6 = estimated (dead reckoning) (2.3 feature)
7 = Manual input mode
8 = Simulation mode
3          Number of satellites being tracked
6.65       Horizontal dilution of position
1669.0,M   Altitude, Meters, above mean sea level
-16.9,M    Height of geoid (mean sea level) above WGS84 ellipsoid
(empty)     Time in seconds since last DGPS update
(empty)     DGPS station ID number
*48        Checksum data, always begins with *

```

The second NMEA example sentence is a RMC (Recommended Minimum data) sentence. It contains position, velocity, and time (but not altitude) information:

```
$GNRMC,154005.715,A,4348.9690,N,11147.0400,W,0.11,21.74,030622,,A*55
```

Where:

```

$          Begin sentence indicator
GN         Multi GNSS solution
RMC        Recommended Minimum Sentence C
154005.715  Fix taken at 15:40:07.715 UTC (3:40 p.m.)
4348.9690,N Latitude 43 deg 48.9690' N
11147.0400,W Longitude 111 deg 47.0400' W (-111 deg 47.0400' E)
0.11       Speed over the ground in knots
21.74      Track angle in degrees
030622     Date (3 June 2022)
(empty,empty)Magnetic variation, Direction
*48        Checksum data, always begins with *

```

8.6 Interpreting GPS Data

Before the GPS sensor has a position fix, it may output PMTK commands¹. These commands communicate that the GPS sensor is powering-on or restarted, or they communicate other settings related to the status of the GPS sensor.

8.6.1 Interpreting Latitude and Longitude

As discussed above, the NMEA format communicates the GPS sensor's position in a specific format. Specifically, the NMEA latitude is given by four digits before the decimal point, and up to seven digits after the decimal point: d d m m . m m m m m m m. A letter "N" (North) indicates that the latitude is positive, an "S" indicates a negative latitude. The first two digits (d d) indicate the latitude in degrees. The remaining digits (m m . m m m m m m m) have units of minutes. It can be useful to convert the latitude to decimal degrees. This is done by converting minutes to degrees by dividing by 60 and adding the result to the first two digits. This is shown in the following example.

Example: Convert the NMEA latitude 4348.9690,N to decimal degrees.

Solution: The latitude will be positive, as indicated by "N" (North). The first two digits (43) indicate the latitude in degrees. The remaining digits (48.9690) have units of minutes. We must convert them to decimal degrees by dividing them by 60. Then we can add them to the first two digits (43) to get the latitude in decimal degrees. The conversion from NMEA latitude to decimal degrees is

$$4348.9690, N = 43 + \frac{48.9690}{60} = 43.8162$$

The first two digits were grayed out to show that they already have the correct units and only the remaining digits must be converted. Therefore, the latitude is 43.8162 decimal degrees.

A similar approach is used to convert the NMEA longitude to decimal degrees. The only difference is that there are five digits before the decimal point – the first three are already in decimal degrees. A letter "E" (East) indicates a positive longitude whereas "W" (West) is negative.

Example: Convert the NMEA longitude 11147.0400,W to decimal degrees.

Solution: The longitude will be negative, as indicated by "W" (West). The first three digits (111) indicate the longitude in degrees. The remaining digits (47.0400) have units of minutes. We must convert them to decimal degrees by dividing them by 60. Then, we can add them to the first three digits (111) to get the longitude in decimal degrees. The conversion from NMEA longitude to decimal degrees is

$$11147.0400, W = -\left(111 + \frac{47.0400}{60}\right) = -111.7840$$

The first three digits were grayed out to show that they already have the correct units and only the

¹see <https://www.digikey.com/htmldatasheets/production/1801407/0/0/1/pmtk-command-packet.html>, accessed June, 2022

remaining digits must be converted. Therefore, the longitude is -111.7840 decimal degrees.

8.6.2 Latitude and Longitude to Distance in Meters

Assuming the earth to be spherical, the distance along its surface from one point to another is equal to the arc length. The arc length s of a circular arc of radius r and angle θ in radians is

$$s = r\theta \quad (8.6)$$

Eq. (8.6) requires knowledge of the angle between the two points as measured from the center of the earth. The cosine formula uses the dot-product to find the angle between two vectors \vec{A} and \vec{B} :

$$\cos\theta = \frac{\vec{A} \cdot \vec{B}}{\|\vec{A}\| \|\vec{B}\|} \quad (8.7)$$

Consider a geocentric coordinate system with its origin at the center of the earth, see Figure 8.13. The x -axis points from the center of the earth through the equator (latitude of 0°) and longitude of 0° . The y -axis points from the center of the earth through the (latitude, longitude) point $(0^\circ, 90^\circ)$. The z -axis points from the center of the earth through the north pole at the (latitude, longitude) point $(90^\circ, 0^\circ)$.

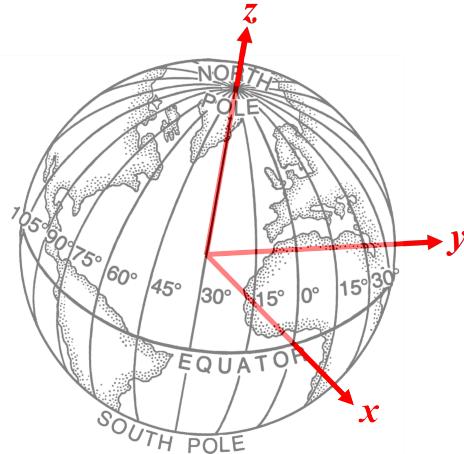


Figure 8.13 Geocentric coordinate system.

An arbitrary point A (see Figure 8.14) on the surface of the earth with latitude ϕ_A and longitude λ_A could be represented in the geocentric coordinate system by the vector

$$\vec{A} = r \langle \cos \lambda_A \cos \phi_A \quad \sin \lambda_A \cos \phi_A \quad \sin \phi_A \rangle \quad (8.8)$$

where r is the radius of the earth. The radius of the earth at sea level at the equator is 6,378,137 m. However, the earth's radius varies, and the local value should be used for highest accuracy.

Combining Eqs. (8.7) and (8.8), the angle θ between two vectors \vec{A} and \vec{B} pointing from the earth's center to points A and B on the earth's surface is

8.6 Interpreting GPS Data

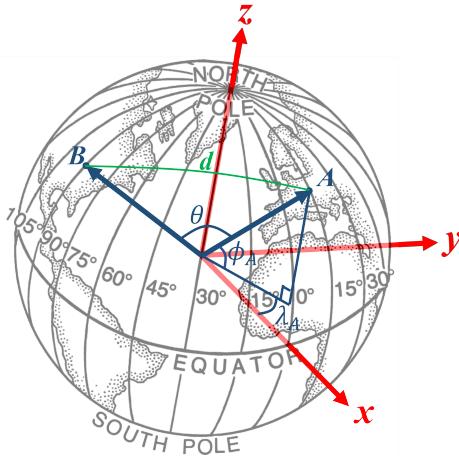


Figure 8.14 Arbitrary points A and B on the earth's surface can be represented by vectors.

$$\cos\theta = \frac{r \langle \cos\lambda_A \cos\phi_A \quad \sin\lambda_A \cos\phi_A \quad \sin\phi_A \rangle \cdot r \langle \cos\lambda_B \cos\phi_B \quad \sin\lambda_B \cos\phi_B \quad \sin\phi_B \rangle}{r^2} \quad (8.9)$$

where ϕ_B and λ_B are the latitude and longitude respectively of point B . Eq. (8.9) can be simplified to

$$\begin{aligned} \cos\theta &= \langle \cos\phi_A \quad 0 \quad \sin\phi_A \rangle \cdot \langle \cos(\lambda_B - \lambda_A) \cos\phi_B \quad \sin(\lambda_B - \lambda_A) \cos\phi_B \quad \sin\phi_B \rangle \\ &= \cos\phi_A \cos\phi_B \cos(\lambda_B - \lambda_A) + \sin\phi_A \sin\phi_B \end{aligned} \quad (8.10)$$

The shortest distance d (see Figure 8.14) from A to B along the surface of the earth can be calculated by Eq. (8.6) as

$$d = r \cos^{-1} (\cos\phi_A \cos\phi_B \cos(\lambda_B - \lambda_A) + \sin\phi_A \sin\phi_B) \quad (8.11)$$

where r is the average earth radius, ϕ_A and λ_A are the latitude and longitude respectively of point A , and ϕ_B and λ_B are the latitude and longitude respectively of point B .

It is often useful to know the relative distance between two points A and B in terms of a two-dimensional grid with North and East as the axes. North is like the x-axis and East is like the y-axis. The distance d_N along the North axis is calculated by the arc-length formula:

$$d_N = r (\phi_B - \phi_A) \quad (8.12)$$

where the latitudes ϕ_A and ϕ_B must have units of radians².

The distance d_N is independent of the longitude. Conversely, the distance d_E along the East axis depends on the latitude; farther from the equator, two points with the same latitude $\phi_A = \phi_B$ but different longitudes $\lambda_A \neq \lambda_B$ are closer together. The distance is derived from Eq. (8.11):

$$d_E = \text{sign}(\lambda_B - \lambda_A) r \cos^{-1} (1 + \cos^2 \phi_A (\cos(\lambda_B - \lambda_A) - 1)) \quad (8.13)$$

²To convert decimal degrees to radians, multiply by $\frac{\pi}{180}$.

The “sign” operator returns the sign (\pm) of its input argument ($\lambda_B - \lambda_A$).

When distances are small enough that the surface of the earth between the two points A and B can be approximated as flat, the distance d calculated in Eq. (8.11) can be approximated as

$$d \approx \sqrt{d_N^2 + d_E^2} \quad (8.14)$$

When calculating distances between two points on the earth, care must be taken when crossing the $\pm 180^\circ$ longitude. For example, the distance along the equator from $\lambda_A = -170^\circ$ to $\lambda_B = 170^\circ$ longitude would be $-20\frac{\pi}{180}r$ and not $-340\frac{\pi}{180}r$.

To correct for longitudes whose difference results in an angle greater than 180° , the value 360° should first be added to the negative-valued longitude. For example, the distance along the equator from $\lambda_A = -170^\circ$ to $\lambda_B = 170^\circ$ longitude would be $(170 - (-170 + 360))\frac{\pi}{180}r = -20\frac{\pi}{180}r$.

8.6.3 Converting Distances to Latitude and Longitude

It may sometimes be beneficial to convert the distances d_N and d_E along north and east axes respectively to changes in latitude and longitude. From Eq. (8.12), the latitude change $\Delta\phi$ in radians is

$$\Delta\phi = \frac{d_N}{r} \quad (8.15)$$

From Eq. (8.13), the longitude change $\Delta\lambda$ in radians is

$$\Delta\lambda = \text{sign}(d_E) \cos^{-1} \left(1 + \frac{\cos\left(\frac{d_E}{r}\right) - 1}{\cos^2 \phi_A} \right) \quad (8.16)$$

where ϕ_A is the latitude at which the change in eastern position d_E occurred.

8.7 GPS Displacement Algorithm

This section presents an algorithm that uses GPS sensor signals to calculate the North, East, and Down inertial displacement of the airplane. The algorithm is provided below:

GPS Displacement Algorithm

GPS latitude and longitude are usually provided in units of degrees. In this algorithm, latitude and longitude are first converted to units of radians. It is often possible to detect that new GPS information is available if the values for GPS latitude or longitude have changed. If new GPS information is available, latitude ϕ_k (rad), longitude λ_k (rad), and altitude h_k (m) are used to calculate the inertial North-East-Down displacements $x_{I,k}$, $y_{I,k}$, and $z_{I,k}$ (m) at time t_k respectively:

8.7 GPS Displacement Algorithm

```
xI,k = r(ϕk - ϕ0)
if ( |λk - λ0| <= π )
    Δλ = λk - λ0
else if ( |λk - λ0| > π )
    if ( λk > λ0 )
        Δλ = λk - (λ0 + 2π)
    else
        Δλ = (2π + λk) - λ0
    end
end
yI,k = sign(Δλ) r cos-1((cos(Δλ) - 1) cos2 ϕk + 1)
zI,k = -(hk - h0)
```

(8.17)

The constants ϕ_0 (rad), λ_0 (rad), and h_0 are the latitude, longitude, and altitude of a fixed origin from which all displacements x_I , y_I , and z_I are measured. For example, the origin could be the initial location from which the airplane is launched. The earth's radius r (m) is approximately 6,371,000 m, but a more accurate local value can be used if necessary. The subscript k relates to the present iteration of the microcontroller.

8.7.1 Testing the GPS Displacement Algorithm

When the inputs to GPS displacement algorithm are (latitude and longitude are given in degrees and need to be converted to radians)

```
ϕk = 41.69504995
λk = -112.2965657
hk = 1204.951409
ϕ0 = 41.69486814
λ0 = -112.2962295
h0 = 1168.0498866
```

The outputs should be

```
xI,k = 20.2163
yI,k = -27.9143
zI,k = -36.9015
```

8.7.2 GPS Velocity Algorithm

In addition, when new GPS information becomes available, the GPS linear velocity vector V_{GPS} (m/s) and its low-pass filtered version V_I (m/s), both in the inertial frame, are calculated as

GPS Velocity Algorithm

```

if (new GPS information)
     $\Delta t_{GPS} = t_k - t_{GPS}$ 
     $t_{GPS} = t_k$ 
     $V_{GPS} = \frac{X_{I,k} - X_{I,k-1}}{\Delta t_{GPS}}$  (8.18)
     $V_{I,k} = (1 - \gamma) V_{I,k-1} + \gamma V_{GPS}$ 
else
     $V_{I,k} = V_{I,k-1}$ 
end

```

$X_{I,k-1}$ (m) and $V_{I,k-1}$ (m/s) are the vectors of the positions and velocities from the previous GPS timestep, *i.e.*, one time-step Δt_{GPS} ago. The constant $0 \leq \gamma \leq 1$ is a user-defined tuning parameter. Values close to $\gamma = 1$ are typical when the GPS signal is reliable. Detecting changes in GPS latitude and / or longitude can indicate that new GPS information is available.

Chapter 9

Signal Filtering and Linear State Estimation

Contents

9.1	Low-Pass, High-Pass, Band-Pass, and Band-Stop Filters	286
9.1.1	Transfer Functions of Low-Pass Filters	287
9.1.2	Digital Implementation of a Low-Pass Filter	289
9.1.3	Electrical Analog Implementation of a Low-Pass Filter	291
9.1.4	Transfer Functions of High-Pass Filters	292
9.1.5	Digital Implementation of a High-Pass Filter	293
9.1.6	Electrical Analog Implementation of High-Pass Filters	293
9.1.7	Band-Pass Filters	294
9.1.8	Band-Stop or Notch Filters	295
9.2	State Estimation using a Kalman Filter	296
9.2.1	The Kalman Filter Algorithm	296
9.3	Luenberger Observer State-Estimation	301
9.3.1	Observability	301
9.3.2	The Luenberger Observer	303
9.3.3	Calculating the Observer Gain L	304
9.3.4	Comparing the Luenberger Observer with the Kalman Filter	309
9.4	Complementary Filters and Altitude	309
9.4.1	The Complementary Filter	311
9.4.2	Backward Euler Implementation of a Low-Pass Filter	312
9.4.3	Complementary Filter For Altitude Sensor Fusion	312
9.4.4	Altitude Complementary Filter Experimental Results	313
9.5	Estimating Gravity	314
9.5.1	Deriving the Gravity Estimation State-Equations	314
9.5.2	Sensor Measurements for Gravity Estimation	316
9.6	Gravity Estimation using a Kalman Filter	317
9.6.1	Simulation Example	318
9.6.2	Kalman Filtering Results: Actual Flight Data	318

9.7 Gravity Estimation Using a Low Pass Filter	321
9.7.1 Simulation and Experimental Results	322

Mechatronic applications often require signal processing, filtering, or estimation. A **signal** may be a sensor measurement, a calculated state variable, an input, or a system output. Sensor signals can be noisy and biased. In this book, **noise** usually refers to random and high-frequency deviations around the expected value of the true signal. **Bias** refers to a constant or low-frequency, time-varying offset of the signal from its expected value. Signal filtering and estimation can reduce the negative effects of noise and bias. Filtering and estimation can also calculate unmeasured signals and state-variables. The estimated state-variable can be used for estimator-based full-state feedback controllers.

9.1 Low-Pass, High-Pass, Band-Pass, and Band-Stop Filters

Filtering is a signal processing technique that can reduce the noise or bias in a signal. In a noisy signal, the *signal frequency* is often different from the *noise frequency*. For example, perhaps you have used a digital scale where you observed rapid changes in the last decimal places displayed on the scale even though the weight on the scale was not changing. Since the actual weight is not changing, the signal frequency is zero. The noise frequency, however, is high, as can be seen by the rapidly changing numbers. The high-frequency changes observed on the scale are likely caused by electronic noise. Most electrical signals experience high-frequency noise. When the signal frequency and noise frequency are different, however, filtering can reduce the noise.

Many types of filters exist. Some basic types are named according to the range of frequencies they remove or allow to pass. A **low-pass filter** allows low-frequency signals to pass, but attenuates (dampens out) high-frequency signals. Low-pass filters are used to remove high-frequency noise from a signal. **High-pass filters** allow high-frequency signals to pass, but attenuate low-frequency signals. High-pass filters remove offset bias and other slowly changing parts of a signal. A **band-pass filter** only allows signals in a specified band of frequencies to pass, but attenuates signals of all other frequencies. A **band-stop filter – i.e., notch filter** – attenuates signals in a specified band of frequencies, but allows signals of all other frequencies to pass.

Since the type of filter depends on the frequencies it attenuates or allows to pass, this chapter will use the frequency domain analysis tools of Section 2.8. A filter's type can often be determined by its Bode plot or frequency response plot. The following example demonstrates this.

Bode Plots and Filters

Example 9.1.1. Determine the filter by inspecting its Bode plot

Figure 9.1 shows Bode plots for four different types of filters: (1) Low-pass, (2) high-pass, (3) band-pass, and (4) band-stop. Match the filter type with the Bode plot. Describe your reasoning.

9.1 Low-Pass, High-Pass, Band-Pass, and Band-Stop Filters

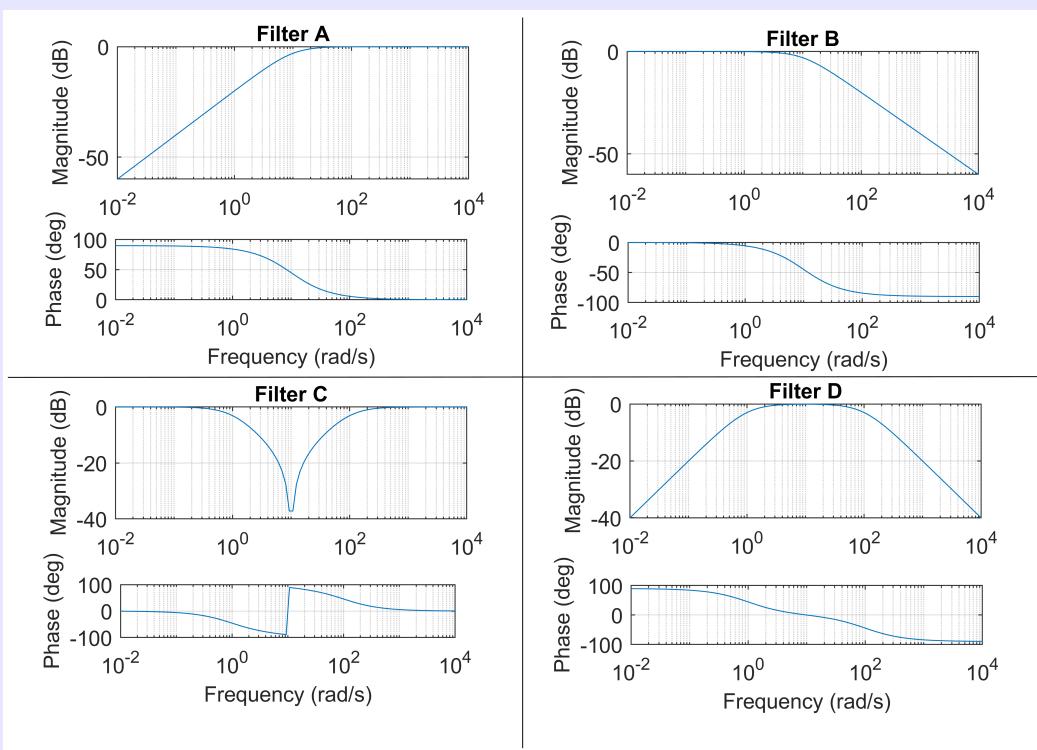


Figure 9.1 Bode plots of different types of filters

Solution: We can identify the type of filter by inspecting its Bode magnitude plot. A Bode magnitude plot shows how a dynamic system affects (by attenuating, amplifying, or passing) the output signal relative to the input over a range of frequencies. Inspecting Filter A, we see that low frequency signals between 10^{-2} rad/s to 10 rad/s are attenuated compared to higher frequency signals. The higher frequency signals are not attenuated but allowed to pass. Therefore, **Filter A is a high-pass filter**. The opposite is true for Filter B. The Magnitude plot of Filter B shows that low frequency signals from 10^{-2} rad/s to 10 rad/s have an attenuation magnitude of 0 dB, but higher frequency signals are attenuated to lower decibels. Because Filter B allows low frequency signals to pass but stops high frequency signals, **Filter B is a low-pass filter**.

Signals between 1 to 100 rad/s are attenuated by Filter C; therefore, **Filter C is a band-stop filter** because it stops signals in a band of frequencies. Filter D is the opposite. It allows signals in a band of frequencies from 1 to 100 rad/s to pass but stops signals of all other frequencies. **Filter D is therefore a band-pass filter**.

9.1.1 Transfer Functions of Low-Pass Filters

Filters can be represented mathematically by Laplace transfer functions. A **first-order low-pass filter** has the following transfer function:

$$\frac{Y}{U} = \frac{1}{\tau s + 1} \quad (9.1)$$

where τ (usually units of seconds) is the time-constant of the filter. In the frequency domain, $\frac{1}{\tau}$ (usually units of rad/s) is the cutoff frequency of the filter. Filter B in Figure 9.1 is a first order filter with $\tau = 0.1$.

A **second-order low-pass filter** has the following transfer function:

$$\frac{Y}{U} = \frac{\omega_n^2}{s^2 + 2\zeta\omega_n s + \omega_n^2} \quad (9.2)$$

where the damping ratio is $\zeta > 0$, and $\omega_n > 0$ is the natural frequency of the filter. If $\zeta \geq 1$, the second-order filter transfer function can be obtained by multiplying two first-order filter transfer functions. Figure 9.2 shows the effect of varying the damping ratio ζ from 0.01 to 10. Lower values of ζ have a sharper roll-off at the cutoff frequency of $\omega_n = 10$, but if $\zeta < 0.707$, the filter amplifies signals immediately at and near the cutoff frequency. A value of $\zeta = \frac{1}{\sqrt{2}} \approx 0.707$ provides the sharpest roll-off without any signal amplification.

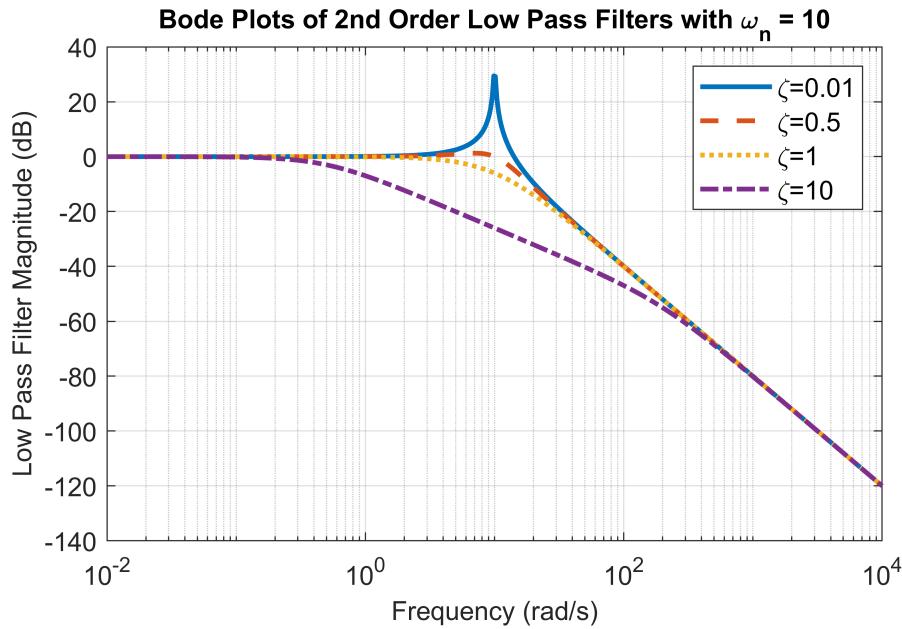


Figure 9.2 Effects of varying ζ on the Bode plot of a second-order low-pass filter

The first-order low-pass filter, Filter B of Figure 9.1, and the second-order low-pass filter of Figure 9.2 have the same cutoff frequency of $\omega = 10$. At high frequencies, $\omega > 100$ rad/s, the slope of the first-order filter is -20 dB/decade; however, the second-order filter has a slope of -40 dB/decade. In general, higher-order filters have steeper attenuation slopes. The attenuation slope of a third order filter is -60 dB/decade; a fourth-order filter's slope is -80 dB/decade, etc. Transfer functions for higher-order filters are obtained by multiplying the transfer functions of combinations of first and second-order low-pass filters.

9.1 Low-Pass, High-Pass, Band-Pass, and Band-Stop Filters

9.1.2 Digital Implementation of a Low-Pass Filter

Signal filtering can be digital or analog. Computer code implements a digital filter by solving the filter's governing dynamic equation. The next section will discuss analog implementations.

To implement a digital filter, we must first derive the discrete-time solution to the filter's transfer function. To implement the second-order filter of Eq. (9.2), for example, we could convert the transfer function to state-space using controllable canonical form (see Section 2.2.3) then use any of the techniques of Table 3.1 to obtain the discrete-time numerical implementation. The following example implements a first-order low-pass filter using the backward Euler numerical method.

Digital First-Order Low-Pass Filter

Example 9.1.2. Implementing a digital first-order low-pass filter

Use the backward Euler numerical method of Table 3.1 and write MATLAB code to implement a digital low-pass filter. The filter transfer function is

$$\frac{V_{\text{filtered}}(s)}{V_{\text{in}}(s)} = \frac{1}{\tau s + 1}$$

with $\tau = 1$ second. The input signal is a voltage signal shown in the top graph of Figure 9.3 with two sinusoidal frequencies $\omega_1 = 0.1 \text{ rad/s}$ and $\omega_2 = 10 \text{ rad/s}$, and two amplitudes $A_1 = 1 \text{ V}$ and $A_2 = 0.2 \text{ V}$:

$$V_{\text{in}}(t) = \sin(0.1t) + 0.2 \sin(10t)$$

The higher frequency part ($0.2 \sin(10t)$) is considered noise that should be attenuated by the low-pass filter. Implement the filter in MATLAB and graph the filtered output signal.

Solution: The observable canonical form (see Section 2.2.3) of the first-order low-pass filter transfer function is

$$\begin{aligned} y &= x \\ \dot{x} &= -\frac{1}{\tau}x + \frac{1}{\tau}V_{\text{in}} \end{aligned}$$

The backward Euler implementation (see Table 3.1) of the filter is

$$V_{\text{filtered},k} = y_k = x_k \quad (9.3)$$

$$x_{k+1} = \frac{\tau}{\Delta t + \tau} x_k + \frac{\Delta t}{\Delta t + \tau} V_{\text{in},k} \quad (9.4)$$

where Δt is the timestep. MATLAB code for the implementation of the digital filter is shown below:

```
close all %Close all figures
clear all %Clear all variables and functions
clc %Clear the Command Window

dt = 0.01; %(s) timestep
```

```

t = 0:dt:100; %(s) time vector
tau = 1; %(s) filter time-constant
Vin = sin(0.1*t)+0.2*sin(10*t); %Input noisy voltage signal
N = length(t); %Number of timesteps
x = 0; %Initial condition
V_filtered = zeros(1,N); %Allocate memory to store the filtered voltage
for ii = 1:N
    V_filtered(ii) = x; %Store the filtered voltage
    %Backward Euler digital implementation of the low-pass filter
    x = tau/(dt+tau)*x+dt/(dt+tau)*Vin(ii);
end
%Graph the filtered output voltage
figure
subplot(211)
plot(t,Vin)
grid on
ylabel('Noisy Input Signal (V)')
title('Digital Low-Pass Filtered Signal')
subplot(212)
plot(t,V_filtered)
xlabel('Time (s)')
ylabel('Filtered Output Signal (V)')
grid on

```

The bottom graph of Figure 9.3 shows the filtered output signal. This plot looks like the low frequency part of V_{in} , *i.e.*, $\sin(0.1t)$, indicating that the high frequency content was attenuated or mostly removed. The low frequency content of the input signal was allowed to pass. This example demonstrates that digital low-pass filters can be implemented using one or two lines of computer code (the two lines of code in the FOR loop). Filters can remove noise from signals.

9.1 Low-Pass, High-Pass, Band-Pass, and Band-Stop Filters

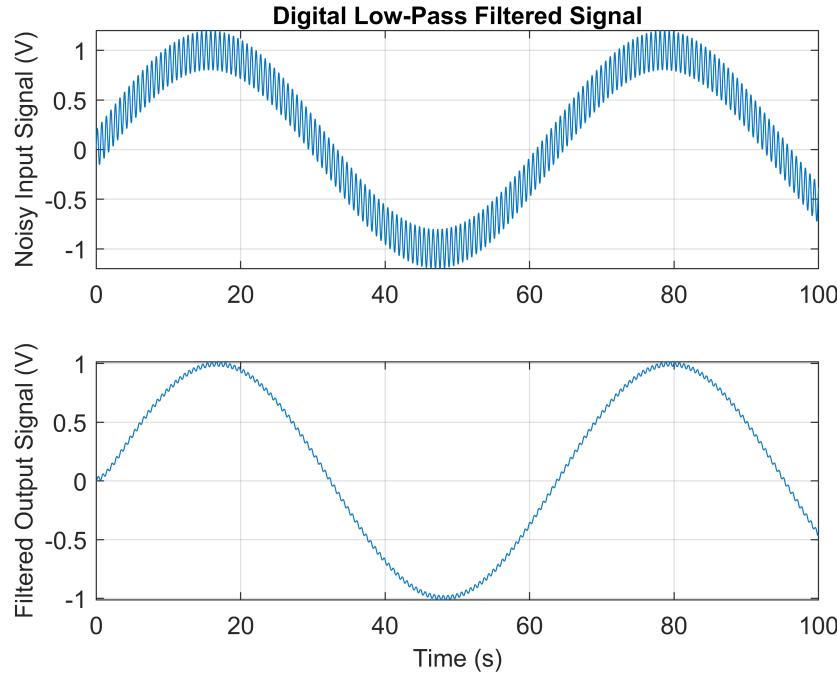


Figure 9.3 The digital low-pass filter attenuates the high-frequency noise.

9.1.3 Electrical Analog Implementation of a Low-Pass Filter

Filters can be analog as well as digital. For example, a car suspension is a mechanical example of an analog low-pass filter. The car suspension dampens the high-frequency road vibrations to provide a more comfortable ride for the passengers. The passengers do not feel all the bumps from the gravel on the road, but they do feel the low-frequency, slowly changing, road grades of hills.

Electrical circuits can be designed as analog filters for voltage signals. A first-order low-pass filter consists of a resistor and capacitor in series with the input voltage signal V_{in} . The filtered output voltage V_{out} is the voltage drop across the capacitor. The first-order analog low-pass filter is the circuit shown in Figure 9.4.

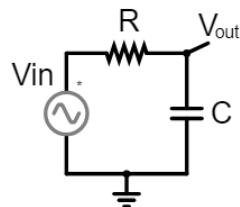


Figure 9.4 An electrical circuit that acts as an analog low-pass filter for the electrical signal V_{in}

Changing the capacitance and resistance of the circuit affects the time-constant and cutoff frequency of the filter. The following example demonstrates how to select the resistance and capacitance to design

the analog equivalent of the digital low-pass filter of Example 9.1.2.

Analog First-Order Low-Pass Filter Design

Example 9.1.3. Design an analog low-pass filter

Select the resistance R and capacitance C for the low-pass filter circuit of Figure 9.4 to design the analog equivalent of the digital low-pass filter of Example 9.1.2.

Solution: To match the digital low-pass filter of Example 9.1.2, we must design a circuit with a time-constant of $\tau = 1$ s. The transfer function model of the circuit must be

$$\frac{V_{\text{filtered}}(s)}{V_{\text{in}}(s)} = \frac{1}{\tau s + 1}$$

V_{out} of Figure 9.4 is the filtered voltage V_{filtered} . The impedance of a capacitor is $\frac{1}{sC}$. The impedance of a resistor is R . Therefore, the voltage divider law for the circuit of Figure 9.4 is

$$\begin{aligned} \frac{V_{\text{out}}}{V_{\text{in}}} &= \frac{\frac{1}{sC}}{R + \frac{1}{sC}} \\ &= \frac{1}{RCs + 1} \end{aligned}$$

Comparing this transfer function with the transfer function for a first-order low-pass filter indicates that $\tau = RC$. To make this analog filter match the digital filter of Example 9.1.2, we must choose R and C so that $RC = 1$ s. There are an infinite number of possible combinations. One combination is $R = 10 \text{ k}\Omega$ and $C = 100 \mu\text{F}$. Then $\tau = RC = 1$ s and the solution is complete.

Adding an analog low-pass filter to an existing electrical circuit creates another path for the flow of electricity. It is critical to consider this effect when filtering electrical signals, therefore, Example 9.1.3 is only part of the process of selecting components for the filter. In addition to Example 9.1.3, the impedance of the filter should be high enough to prevent significant electrical current from flowing into the filter. If the output of filter must drive an electrical load, an active op amp filter must be used instead of the passive filter of Figure 9.4. An op amp provides high impedance at its input and low impedance at its output which can buffer the input circuit from the output circuit. An active, first-order, low-pass filter circuit using an op amp is shown in Figure 9.5. Like the passive filter of Figure 9.4, components of the active low-pass filter of Figure 9.5 must be selected to have sufficiently high impedance to avoid significantly altering the dynamic behavior of the existing circuit.

Higher-order analog filters can be designed using electrical circuits. A second-order, low-pass filter requires two capacitors. A third-order filter requires three capacitors, etc. Since inductors store energy, a low-pass filter could use an inductor instead of a capacitor. For example, a second-order filter could use one capacitor and one inductor; or, it could use two inductors and no capacitors. However, filters are more commonly designed using capacitors rather than inductors.

9.1.4 Transfer Functions of High-Pass Filters

A first-order high-pass filter has the following transfer function:

9.1 Low-Pass, High-Pass, Band-Pass, and Band-Stop Filters

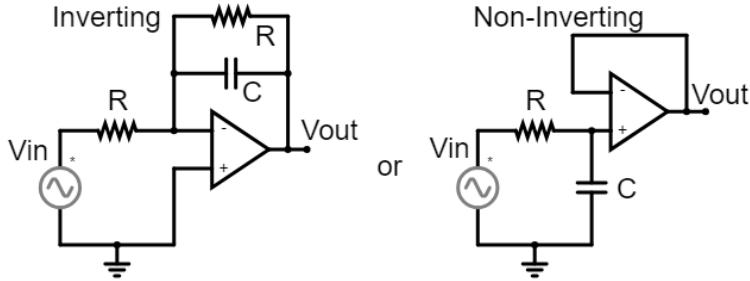


Figure 9.5 An active analog low-pass filter requires an op amp. These are examples of active, first-order, low-pass filters.

$$\frac{Y}{U} = \frac{\tau s}{\tau s + 1} \quad (9.5)$$

where τ (usually units of seconds) is the time-constant of the filter. In the frequency domain, $\frac{1}{\tau}$ (usually units of rad/s) is the corner frequency of the filter. Filter A in Figure 9.1 is a first order filter with $\tau = 0.1$, i.e., it has a corner frequency of 10 rad/s.

A second-order high-pass filter has the following transfer function:

$$\frac{Y}{U} = \frac{s^2}{s^2 + 2\zeta\omega_n s + \omega_n^2} \quad (9.6)$$

where the damping ratio is $\zeta > 0$, and $\omega_n > 0$ is the natural frequency of the filter. If $\zeta \geq 1$, the second-order filter transfer function can be obtained by multiplying two first-order filter transfer functions. Using the same reasoning as presented in Section 9.1.1, a value of $\zeta \approx 0.707$ produces the sharpest corner without signal amplification. The corner frequency of a second-order high-pass filter is ω_n . Higher-order filters can be obtained by multiplying first and/or second-order transfer function of high-pass filters.

9.1.5 Digital Implementation of a High-Pass Filter

Using the observable canonical form of the transfer function of Eq. (9.5) and the backward Euler numerical method of Table 3.1, the digital implementation of the first-order high-pass filter is

$$y_k = -x_k + u_k \quad (9.7)$$

$$x_{k+1} = \frac{\tau}{\Delta t + \tau} x_k + \frac{\Delta t}{\Delta t + \tau} u_k \quad (9.8)$$

Comparing Eqs. (9.7) and (9.8) with the digital implementation of a the low-pass filter in Example 9.1.2 reveals that the high-pass filter simply subtracts the low-pass filtered signal x_k from the input signal u_k to get the high-pass filtered signal y_k . Digital implementations of higher-order filters are obtained by numerically solving their transfer functions.

9.1.6 Electrical Analog Implementation of High-Pass Filters

Swapping the resistor and capacitor in a low-pass filter (Figure 9.4) results in a high-pass filter. The circuit of Figure 9.6 is an electrical implementation of an analog high-pass filter. The voltage drop across the

resistor is the high-pass filtered output. As with low-pass filters, higher-order high-pass filters can be designed by adding more capacitors (and/or inductors) and resistors.

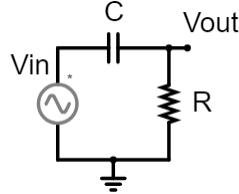


Figure 9.6 A passive electrical implementation of a high-pass filter

Active high-pass filters use op amps. Figure 9.7 shows two possible implementations of a first-order high-pass filter. One is an inverting circuit and the other is non-inverting.

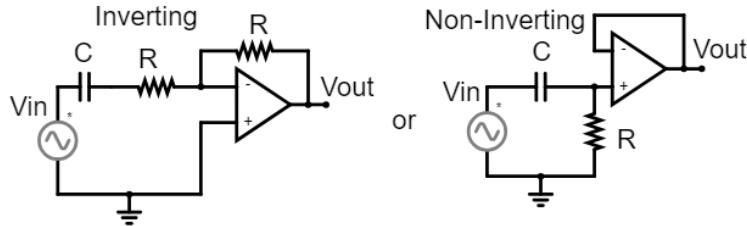


Figure 9.7 Active electrical implementations of high-pass filters

9.1.7 Band-Pass Filters

Band-pass filters are products of high-pass and low-pass filters (compare Filter D of Figure 9.1 with Filters A and B). Therefore, the lowest-order band-pass filter is a second-order filter. The transfer function for a second-order band-pass filter is

$$\frac{Y}{U} = \frac{\tau_H s}{(\tau_H s + 1)(\tau_L s + 1)} \quad (9.9)$$

where τ_H is the time-constant of the high-pass filter (*i.e.*, $\frac{1}{\tau_H}$ is the corner frequency), and τ_L is the time-constant of the low-pass filter (*i.e.*, $\frac{1}{\tau_L}$ is the cutoff frequency). The high-pass filter time-constant τ_H is larger than the low-pass filter time-constant, $\tau_L < \tau_H$. If the objective of the band-pass filter is to focus on a single frequency ω_0 rad/s rather than a band of frequencies, it can be more convenient to write the transfer function as

$$\frac{Y}{U} = \frac{\omega_0 s}{s^2 + 2\zeta\omega_0 s + \omega_0^2} \quad (9.10)$$

A value of $\zeta = 0.5$ is typical for this filter. Figure 9.8 shows an electrical implementation of a band-pass filter. Its transfer function is

$$\frac{Y}{U} = \frac{C_H R_H s}{C_H R_H C_L R_L s^2 + (C_H R_H + C_L R_L + C_H R_L) s + 1} \quad (9.11)$$

9.1 Low-Pass, High-Pass, Band-Pass, and Band-Stop Filters

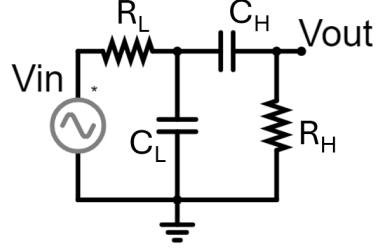


Figure 9.8 Passive electrical implementation of a second-order band-pass filter

If $C_H R_L$ is small so that it is negligible compared to $C_H R_H + C_L R_L$, then Eq. (9.11) is an approximation of Eq. (9.9), where $\tau_H = R_H C_H$ and $\tau_L = R_L C_L$. If there is loading on the filter, an active filter using an op amp is required.

For digital implementation of Eq. (9.9), the output z_k of a low-pass filter with time-constant τ_L could be the input of the high-pass filter with time-constant τ_H . Therefore, using Equations (9.4), (9.8), and (9.7), the backward Euler digital implementation of Eq. (9.9) is

$$y_k = -x_k + z_k \quad (9.12)$$

$$x_{k+1} = \frac{\tau_H}{\Delta t + \tau_H} x_k + \frac{\Delta t}{\Delta t + \tau_H} z_k \quad (9.13)$$

$$z_{k+1} = \frac{\tau_L}{\Delta t + \tau_L} z_k + \frac{\Delta t}{\Delta t + \tau_L} u_k \quad (9.14)$$

where u_k is the unfiltered input signal, and y_k is the band-pass filtered output signal. The iteration timestep is Δt . The digital implementation of Eq. (9.10) requires a canonical conversion of Eq. (9.10) to state-space followed by a numerical solution using a method from Table 3.1.

Higher-order band-pass filters are possible. Like low-pass and high-pass filters, they are multiplication products of high-pass, low-pass, and/or band-pass transfer functions.

9.1.8 Band-Stop or Notch Filters

The lowest-order band-stop (or notch) filter is a second-order filter. The transfer function is

$$\frac{Y}{U} = \left(\frac{\tau_H s}{\tau_H s + 1} + \frac{1}{\tau_L s + 1} \right) = \frac{\tau_L \tau_H s^2 + 2\tau_H s + 1}{\tau_L \tau_H s^2 + (\tau_L + \tau_H) s + 1} \quad (9.15)$$

The backward Euler digital implementation of Eq. (9.15) is the sum of the low-pass filtered signal z_k and the high-pass filtered signal ($u_k - x_k$):

$$y_k = (u_k - x_k) + z_k \quad (9.16)$$

$$x_{k+1} = \frac{\tau_H}{\Delta t + \tau_H} x_k + \frac{\Delta t}{\Delta t + \tau_H} u_k \quad (9.17)$$

$$z_{k+1} = \frac{\tau_L}{\Delta t + \tau_L} z_k + \frac{\Delta t}{\Delta t + \tau_L} u_k \quad (9.18)$$

where u_k is the unfiltered input signal, and y_k is the band-pass filtered output signal. The iteration timestep is Δt .

If the objective is to focus on a single frequency ω_0 or a band of frequencies centered around ω_0 , the second-order band-stop filter can be written as follows:

$$\frac{Y}{U} = \frac{s^2 + \omega_0^2}{s^2 + \omega_Q s + \omega_0^2} \quad (9.19)$$

where ω_Q is the width of the attenuated or stopped frequency band. The digital implementation of Eq. (9.19) requires a canonical conversion of Eq. (9.19) to state-space followed by a numerical solution using a method from Table 3.1.

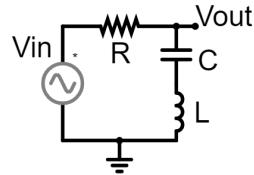


Figure 9.9 Passive electrical implementation of a second-order band-stop filter given by Eq. (9.19)

Figure 9.9 shows an analog circuit that implements Eq. (9.19) with $\omega_0 = \sqrt{1/(CL)}$ and $\omega_Q = R/L$. The filtered output $Y = V_{\text{out}}$ is the voltage drop across the series capacitor-inductor part of the circuit. The unfiltered input signal is $U = V_{\text{in}}$. Notice that this circuit requires large values for C and L to stop lower frequencies. Large capacitors and inductors make this type of band-stop filter less practical than a twin-T notch filter. An active twin-T band-stop filter is the topic of Example 6.7.3.

9.2 State Estimation using a Kalman Filter

9.2.1 The Kalman Filter Algorithm

Model Prediction	Sensor Correction	
$x_p = A_d \hat{x}_k + B_d u_k, \quad (1)$	$S = CP_p C^T + R, \quad (4)$	
$P_p = A_d \hat{P}_k A_d^T + Q, \quad (2)$	$\mathcal{K} = P_p C^T S^{-1}, \quad (5)$	
$y_p = Cx_p + Du_k, \quad (3)$	$\hat{x}_{k+1} = x_p + \mathcal{K}(y_k - y_p), \quad (6)$	
	$\hat{P}_{k+1} = (I_n - \mathcal{K}C) P_p, \quad (7)$	

In addition to removing noise, bias, and unwanted frequencies from a signal, filtering can be used to estimate unknown state-variables and output signals. For Linear-Time-Invariant (LTI) state-space systems such as

$$y = Cx + Du \\ \dot{x} = Ax + Bu$$

9.2 State Estimation using a Kalman Filter

with Gaussian sensor and process noise, the Kalman filter is an optimal state estimator. **State estimators**, which are also called **observers**, calculate an estimate \hat{x} of the state x of a state-space system. The Kalman filter is also called a **sensor fusion** algorithm because it can combine information from multiple sensors and models to provide an optimal prediction \hat{x} of the state x of a state-space system.

The input u and the measured output y are inputs to the Kalman filter. Kalman filter calculates an estimate \hat{x} of the state. Figure 9.10 shows a block-diagram of the Kalman filter.

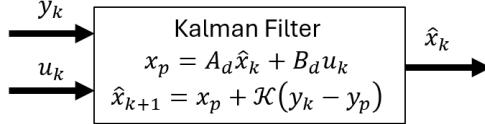


Figure 9.10 Block-diagram of the Kalman filter showing the inputs y_k and u_k and the output \hat{x}_k

Eq. (9.20) is the Kalman filter algorithm for LTI state-space systems. The state-transition matrix A_d and input-transition matrix B_d can be obtained using any of the numerical methods of Table 3.1. The covariance matrix Q is a real-valued, symmetric, **positive definite matrix**, i.e., there exists an invertible matrix G such that $Q = G^T G$, where G^T is the transpose of G . R is the measurement or sensor noise covariance matrix. Like all covariance matrices, R is a symmetric positive definite matrix. In practice, R and Q are often treated as tuning parameters that are determined by trial and error. \hat{x}_k is the Kalman filter's estimate of the state x at the discrete time t_k . An $n \times 1$ vector of zeros is often a sufficient initial condition, where n is the number of state variables. \hat{P}_k is state-error covariance matrix at time t_k . It predicts the covariance in the state estimate. Smaller covariance indicates less uncertainty or greater accuracy in the state estimate \hat{x} . The $n \times n$ identity matrix I_n is often a sufficient initial condition for \hat{P} . The system input signal is u_k . y_k is a $p \times 1$ array of measured output signals, e.g., it consists of sensor measurements. The other variables, x_p , P_p , y_p , S , and K , are intermediate calculations; for example, the Kalman filter algorithm could be written without the intermediate calculations using only two steps instead of seven as follows:

$$\begin{aligned}\hat{x}_{k+1} &= A_d \hat{x}_k + B_d u_k + (A_d \hat{P}_k A_d^T + Q) C^T (C (A_d \hat{P}_k A_d^T + Q) C^T + R)^{-1} (y_k - (C (A_d \hat{x}_k + B_d u_k))) \\ \hat{P}_{k+1} &= (I_n - (A_d \hat{P}_k A_d^T + Q) C^T (C (A_d \hat{P}_k A_d^T + Q) C^T + R)^{-1} C) ((A_d \hat{P}_k A_d^T + Q))\end{aligned}$$

Writing the Kalman filter in seven steps instead of two makes it more readable. The derivation of the Kalman filter uses probability theory of multivariable random processes. This level of math is beyond the expected level of understanding for this book. The derivation, therefore, is not included.

The intuition behind the Kalman filter is that the state-transition equation $x_p = A_d x_k + B_d u_k$ provides a prediction of the state at time t_k . The matrices A_d and B_d are rarely perfect, and the input u_k is usually noisy; therefore the model's prediction x_p of the state x_k is inaccurate. The output equation $y_p = C x_k + D u_k$ predicts the output y_k , but since C and D are rarely perfectly known, and u_k is noisy, the model's prediction y_p of the true output is also inaccurate. Sensors provides an independent measurement of the output y_k . Sensors are usually noisy, so the sensor's prediction of the output is also inaccurate. The Kalman filter, however, calculates an estimate \hat{x}_k of the true state x_k by comparing the model's prediction y_p with the sensor's measurement y_k of the output. The larger the difference, the more correction is applied to the state estimate \hat{x}_{k+1} , as can be seen in Step (6) of Eq. (9.20). The Kalman gain K , which is a

function of the model and sensor covariance, is the correction gain. If the sensor is more certain than the model, the Kalman gain \mathcal{K} will correct the state towards the sensor's prediction. If the model is more certain than the sensor, the Kalman gain \mathcal{K} will correct the state towards the model's prediction x_p .

The Kalman filter is a **recursive** algorithm, meaning that it updates its prediction iteratively as new data arrive. The following example demonstrates how the Kalman filter can be implemented in MATLAB code.

Implementing the Kalman Filter

Example 9.2.1. Implement the Kalman filter

Write MATLAB code to implement the Kalman filter to estimate the unknown states of the following (unknown) state-space system:

$$y = \begin{bmatrix} 1 & 0 \end{bmatrix} x$$

$$\dot{x} = \begin{bmatrix} 0 & 1 \\ -2 & -3 \end{bmatrix} x + \begin{bmatrix} 0 \\ 1 \end{bmatrix} u$$

where the true (but unknown) input is

$$u = 4 \sin(3t)$$

A measurement u_m of the input u is known, but noisy according to the following equation:

$$u_m = u + v$$

where v is zero-mean Gaussian noise with a standard deviation of 0.5. A measurement of the output y is known, but noisy. It is corrupted by zero-mean Gaussian noise with a standard deviation of 0.1. The true (but unknown) initial state is

$$x = \begin{bmatrix} 3 \\ -2 \end{bmatrix}$$

The known but inaccurate model of the state-space system is

$$y_p = \begin{bmatrix} 1 & 0 \end{bmatrix} x$$

$$\dot{x} = \begin{bmatrix} 0.1 & 1 \\ -1.6 & -3.2 \end{bmatrix} x + \begin{bmatrix} -0.1 \\ 1.2 \end{bmatrix} u_m$$

Code a Kalman filter in MATLAB to estimate the state-trajectory and graph the estimated states on the same plot as the true, but unknown, states. Since the true initial condition is unknown, assume zero initial conditions. Use the identity matrix as the initial condition for the state-covariance matrix. For the Kalman filter tuning matrices, use

9.2 State Estimation using a Kalman Filter

$$Q = \begin{bmatrix} 0.01 & 0 \\ 0 & 0.01 \end{bmatrix}$$
$$R = 0.01$$

Solution: The following MATLAB code simulates the true (but unknown) system and implements the Kalman filter to calculate the state-estimates.

```
close all %Close all figures
clear all %Clear all variables and functions
clc %Clear the Command Window

dt = 0.1; %(s) timestep
t = 0:dt:5; %(s) discrete time vector
N = length(t); %Number of simulation timesteps
u = 4*sin(3*t); %(V) true (but unknown) input
um = u + 0.5*randn(size(u)); %noisy (but known) input

%True (but unknown) state-space system
A = [0,1;-2,-3];
B = [0;1];
C = [1,0];
D = 0;
Fd = expm([A*dt,B*dt;zeros(1,3)]);
Ad = Fd(1:2,1:2);
Bd = Fd(1:2,3);

%Inaccurate model of the true state-space system
Am = [0.1, 1; -1.6, -3.2];
Bm = [-0.1;1.2];
Cm = [1,0];
Dm = 0;
Fdm = expm([Am*dt,Bm*dt;zeros(1,3)]);
Adm = Fdm(1:2,1:2);
Bdm = Fdm(1:2,3);

%True (but unknown) initial conditions
x = [3;-2];

%Initial state-prediction
xHat = [0;0];

%Initial state-covariance prediction
PHat = eye(size(Am));

%Kalman filter tuning matrices
Q = diag([0.01,0.01]); %Process noise covariance
R = 0.01; %Sensor noise covariance

%Allocate memory to store the state-variables
xVec = zeros(2,N); %True (but unknown) state variables
```

```

xHatVec = zeros(2,N); %Kalman-filter estimated state variables

%Simulate the true system in the same FOR loop as the Kalman filter
for ii = 1:N
    %Simulate the true (but unknown) system
    xVec(:,ii) = x; %Store the true (but unknown) state
    ym = C*x + 0.1*randn; %Noisy measurement of the output
    x = Ad*x+Bd*u(ii); %Update the true (but unknown) state

    %Run the Kalman filter
    xHatVec(:,ii) = xHat; %Store the Kalman filter's state estimate
    %Model prediction
    xp = Ad*xHat + Bd*u(ii); %Model prediction of the state
    Pp = Ad*PHat*Ad' + Q; %Model prediction of the state-covariance
    yp = Cm*xHat + Dm*u(ii); %Model's prediction of the output
    %Sensor correction
    S = Cm*Pp*Cm' + R;
    sK = Pp*C'*S^(-1); %Kalman gain
    xHat = xp + sK*(ym - yp); %Kalman filter's estimate of the state
    PHat = (eye(size(A))-sK*C)*Pp; %Kalman's estimate of state-covariance
end

%Plot the results
figure
subplot(211)
plot(t, xVec(1,:), t, xHatVec(1,:),'--')
ylabel('x_1')
legend('True', 'Estimate')
title('Kalman Filter State Estimation Results')
subplot(212)
plot(t,xVec(2,:), t, xHatVec(2,:),'--')
ylabel('x_2')
xlabel('Time (s)')

```

The following graph is the result of the above MATLAB code. It shows the performance of the Kalman filter in estimating the true, but unknown, states.

9.3 Luenberger Observer State-Estimation

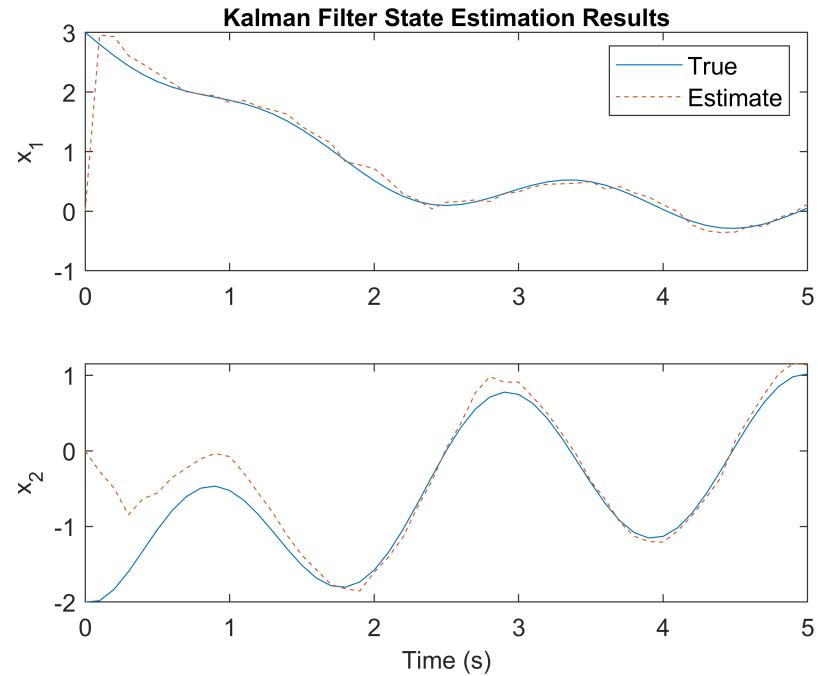


Figure 9.11 Estimates of the state-variables calculated by the Kalman filter

9.3 Luenberger Observer State-Estimation

An observer uses the known control input u and the measured output y_m to estimate the states x as \hat{x} , as shown in the block diagram of Figure 9.12. The block diagram shows two different outputs: y and y_m . The output y_m that is fed into the state-estimator must be measured. A system can have multiple outputs, and any output y not fed back to the observer does not need to be measured.

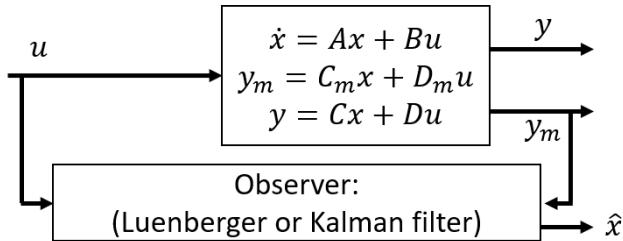


Figure 9.12 State-estimation without feedback control

9.3.1 Observability

Before we can design an observer to estimate the states, we need to determine whether or not the states are observable. To do so we calculate the observability matrix \mathcal{O} .

$$\mathcal{O} = \begin{bmatrix} C_m \\ C_mA \\ \vdots \\ C_mA^{n-1} \end{bmatrix}$$

If the observability matrix is full rank, then the states are observable.

$$\text{rank}(\mathcal{O}) = n$$

One way to simply check that the observability matrix is full rank is to make sure that its determinant is not equal to zero.

$$\det(\mathcal{O}) \neq 0$$

This procedure is similar to calculating the controllability matrix, but instead of using the A and C_m matrices, the controllability matrix used the A and B matrices. If a system's controllability matrix is not full rank, it cannot be controlled. For the given system (A) there are not enough control inputs (through the B matrix) to place all of the states at the desired values. If a system's observability matrix is not full rank, it is not observable, meaning we cannot accurately estimate the states in the vector x , and for the given system (A), the sensors (C_m) are not sufficient to estimate all of the states of the system. If a system is not controllable, then more actuators need to be added. If a system is not observable, more (or better placed) sensors need to be added.

Observability

Example 9.3.1. Observability

Consider the following state-space system:

$$\begin{aligned} \dot{x} &= \begin{bmatrix} 1 & 2 \\ 0 & 3 \end{bmatrix}x + \begin{bmatrix} 5 \\ 1 \end{bmatrix}u \\ y &= \begin{bmatrix} 0 & 1 \end{bmatrix}x \end{aligned}$$

Determine whether the system is observable or not.

Solution: The system is observable if the observability matrix is full rank.

$$\mathcal{O} = \begin{bmatrix} C \\ CA \\ \vdots \\ CA^{n-1} \end{bmatrix} = \left[\begin{bmatrix} 0 & 1 \\ 1 & 2 \end{bmatrix} \begin{bmatrix} 1 & 2 \\ 0 & 3 \end{bmatrix} \right] = \begin{bmatrix} 0 & 1 \\ 0 & 3 \end{bmatrix}$$

$$\det \mathcal{O} = 0$$

The observability matrix is not full rank using a measurement of only the second state! Therefore, the system is not observable.

9.3 Luenberger Observer State-Estimation

9.3.2 The Luenberger Observer

To estimate the states x of the system, we will use an initial guess of the states and use the system model to update the estimated states \hat{x} according to the known feedback input u :

$$\dot{\hat{x}} = A\hat{x} + Bu$$

But, because the initial guess of the states will likely be incorrect, the updated states from the model will still be incorrect. To fix this, we will add a corrector term that will use the measurements y_m to correct the estimated states. The resulting algorithm is called the **Luenberger observer**:

$$\dot{\hat{x}} = A\hat{x} + Bu + L(y_m - (C_m\hat{x} + D_m u)) \quad (9.21)$$

Like the Kalman filter, the Luenberger observer provides an estimate of the state: $\hat{x} \approx x$.

In the observer model, the corrector term $L(y_m - (C_m\hat{x} + D_m u))$ consists of the error between the measurements y_m and the predicted measurements $C_m\hat{x} + D_m u$, which is multiplied by the vector L . If the true measurements are equal to the predicted measurements, the corrector term will go to zero, and the estimated states will equal the true states.

The values in the vector L are called the observer gains. These gains are used to place the poles of the observer to be stable and faster than the controller gains. To place the poles of the observer, we first need to see how to calculate them. If we calculate the observer error e_O as the difference between the true and estimates states:

$$e_O = x - \hat{x}$$

we can take the derivative of the observer error

$$\dot{e}_O = \dot{x} - \dot{\hat{x}}$$

and replace each term on the right-hand side by their state equations.

$$\dot{e}_O = (Ax + Bu) - (A\hat{x} + Bu + L(y_m - (C_m\hat{x} + D_m u)))$$

Plugging in $C_m x + D_m u$ for y_m , the observer error reduces to

$$\dot{e}_O = (A - LC_m)(x - \hat{x}) = (A - LC_m)e_O$$

The poles of the observer are the eigenvalues λ calculated from

$$\det(\lambda I - (A - LC_m)) = 0$$

Because we can pick the observer gains in L , we can place the poles in $A - LC_m$ such that the observer error will be stable and converge to zero faster than the convergence of the controller. The reason we want the poles of the observer to be faster than the poles of the controller is so the observer can estimate the states fast enough that the control input u uses accurate state estimates. And, because the observer is all run in software, we can place the poles to be as fast as we want. Typically, we choose to have the poles about 10 times faster than the controller poles. However, if we choose them to be too fast, the observer will be overly sensitive to model error and sensor noise.

9.3.3 Calculating the Observer Gain L

To calculate the observer gains we follow a process similar to calculating the feedback control gains.

Process for Calculating Luenberger Observer Gains

If using MATLAB, Steps 1-4 below can be replaced with the shortcut command $L = \text{acker}(A', C', o)'$ or $L = \text{place}(A', C', o)'$. These commands calculate the Luenberger observer gain L , where A' is the transpose of the state-matrix, C' is the transpose of the output matrix, and o is a $1 \times n$ vector of desired observer pole locations.

1. Calculate the observability matrix:

$$\mathcal{O} = \begin{bmatrix} C_m \\ C_mA \\ \vdots \\ C_mA^{n-1} \end{bmatrix}$$

and check that it is full rank by checking that its determinant is not equal to zero:

$$\det(\mathcal{O}) \neq 0$$

2. Calculate the characteristic equation of the open loop system:

$$\Delta = \det(sI - A)$$

and arrange it in the form:

$$\Delta = s^n + a_{n-1}s^{n-1} + \cdots + a_1s + a_0$$

Place the coefficients in the row vector:

$$a = [a_{n-1} \ a_{n-2} \ \cdots \ a_1 \ a_0]$$

and use the coefficients to construct the following matrix:

$$\mathcal{A} = \begin{bmatrix} 1 & a_{n-1} & a_{n-2} & \cdots & a_1 \\ 0 & 1 & a_{n-1} & \cdots & a_2 \\ \vdots & \vdots & \ddots & \ddots & \vdots \\ 0 & 0 & \cdots & 1 & a_{n-1} \\ 0 & 0 & 0 & \cdots & 1 \end{bmatrix}$$

3. Using the desired observer pole locations o_1, o_2, \dots, o_n , determine the observer characteristic equation:

$$\Delta_O = (s - o_1)(s - o_2) \cdots (s - o_n)$$

9.3 Luenberger Observer State-Estimation

Factor it into the form:

$$\Delta_O = s^n + \beta_{n-1}s^{n-1} + \cdots + \beta_1s + \beta_0$$

and place the coefficients in the row vector:

$$\beta = [\beta_{n-1} \quad \beta_{n-2} \quad \cdots \quad \beta_1 \quad \beta_0]$$

4. Calculate the observer gains:

$$L = \mathcal{O}^{-1} (\mathcal{A}^T)^{-1} (\beta - a)^T$$

5. To use the observer, you may need to convert it to discrete-time. To calculate the discrete-time observer gain L_d , we use a process similar to finding the input transition matrix B_d :

$$\begin{bmatrix} A_d & L_d \\ \mathbf{0} & \mathbf{I} \end{bmatrix} = \text{expm}\left(\begin{bmatrix} A\Delta t & L\Delta t \\ \mathbf{0} & \mathbf{0} \end{bmatrix}\right)$$

where $\text{expm}()$ is the matrix exponential operator. If the state matrix A is invertible, L_d can also be calculated as follows:

$$L_d = A^{-1} (\text{expm}(A\Delta t) - I) L$$

6. The final step is to implement the observer and calculate the state-estimate \hat{x} . The discrete-time solution is

$$\hat{x}_{k+1} = A_d \hat{x}_k + B_d u_k + L_d (y_{m,k} - (C_m \hat{x}_k + D_m u_k)) \quad (9.22)$$

Because the Luenberger observer is a pole-placement method, the derivation of the observer gain L is nearly identical to the derivation of the pole-placement feedback gain K of Example 13.1.3. The difference in the derivation of the Luenberger gain L is that it uses A^T and C^T instead of A and B . The following example implements a Luenberger observer in Simulink to estimate the states of a simulated system:

Luenberger Observer in Simulink

Example 9.3.2. Implement the Luenberger Observer in Simulink

Use a Luenberger observer in Simulink to estimate the state of the following state-space system:

$$\begin{aligned} \dot{x} &= \begin{bmatrix} 0 & 1 \\ -8 & -5 \end{bmatrix} x + \begin{bmatrix} 0 \\ 5 \end{bmatrix} u \\ y &= \begin{bmatrix} -8 & -5 \end{bmatrix} x + 0.5u \end{aligned}$$

The input signal u is

$$u = 6 \sin(0.5t + 0.3) \quad (9.23)$$

The initial condition is

$$x(0) = \begin{bmatrix} 0.1 \\ -0.5 \end{bmatrix} \quad (9.24)$$

Place the poles of the Luenberger observer at $o_1 = -8$ and $o_2 = -12$.

Solution: To implement the Luenberger observer, we first follow the steps outlined in Section 9.3.3 to find the gain L . Then we can use the Luenberger observer (Eq. (9.21)) in a Simulink block diagram (like Figure 9.12) to calculate the state estimate \hat{x}

The MATLAB command “place” can be used to implement the steps of Section 9.3.3 to find the gain L .

```
%Simulation parameters and State-space Matrices
A = [0,1;-8,-5];
B = [0;5];
C = [-8,-5];
D = 0.5;
x0 = [0.1;-0.5];

%Observer poles
o = [-8,-12];
%Luenberger observer gain
L = place(A',C',o)'
```

Doing so gives the result

$$L = \begin{bmatrix} 5 \\ -11 \end{bmatrix} \quad (9.25)$$

We now use the following Simulink block diagram to implement the Luenberger observer:

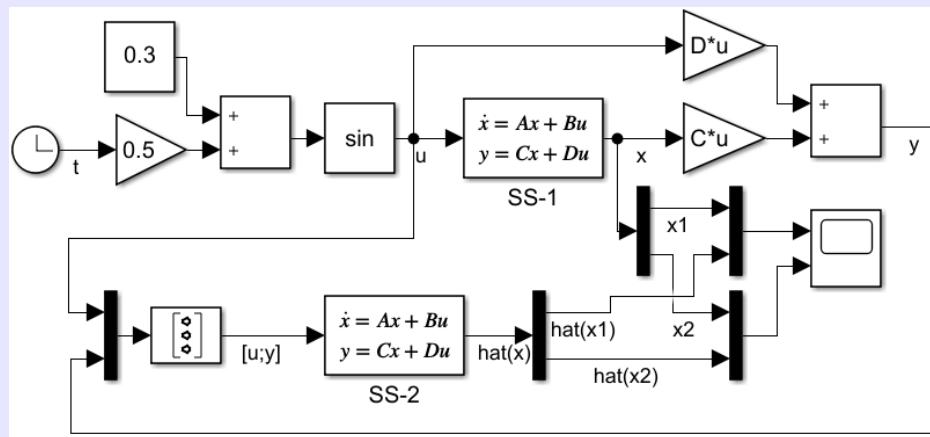


Figure 9.13 Simulink Implementation of the Luenberger State Estimator

The SS-1 block has the following parameters:

9.3 Luenberger Observer State-Estimation

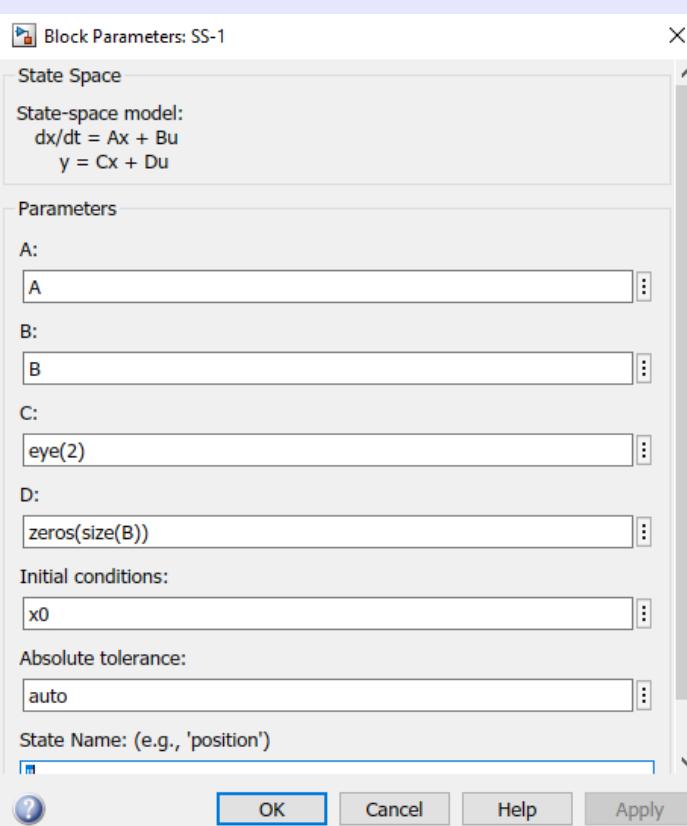


Figure 9.14 Parameters in the SS-1 block

The A and B matrices are those listed in the MATLAB file above. The initial condition was $x_0 = [0.1; -0.5]$.

Another way to write the Luenberger observer of Eq. (9.21) is

$$\dot{\hat{x}} = (A - LC)\hat{x} + \begin{bmatrix} B - LC & L \end{bmatrix} \begin{bmatrix} u \\ y \end{bmatrix} \quad (9.26)$$

The SS-2 block is the Luenberger observer in the form of Eq. (9.26). The A, B, C, D, and L matrices are those listed in the MATLAB file below. The result from running the Simulink model is shown in the figure below. The state estimates \hat{x} converged to the actual state x within about one second.

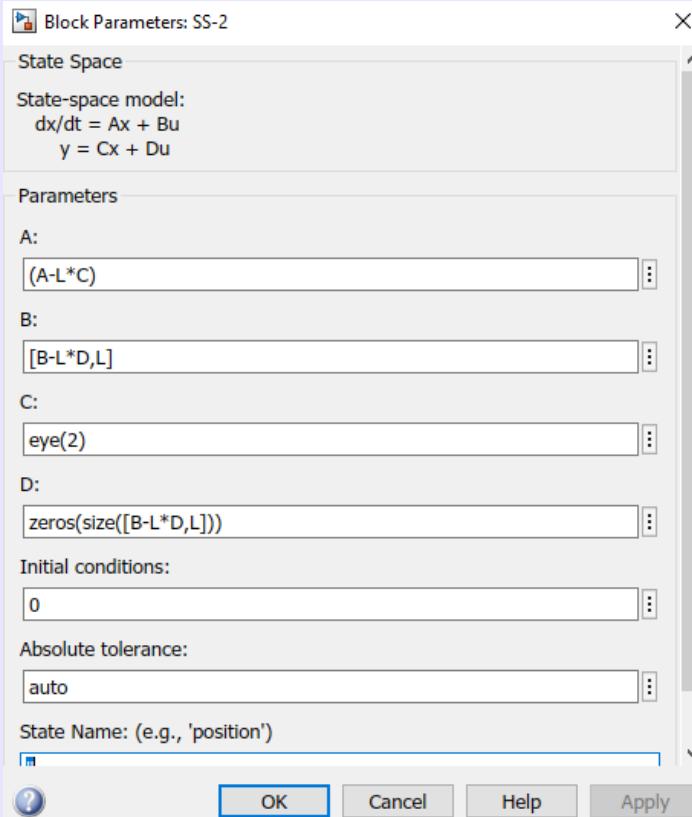


Figure 9.15 Parameters in the SS-2 block

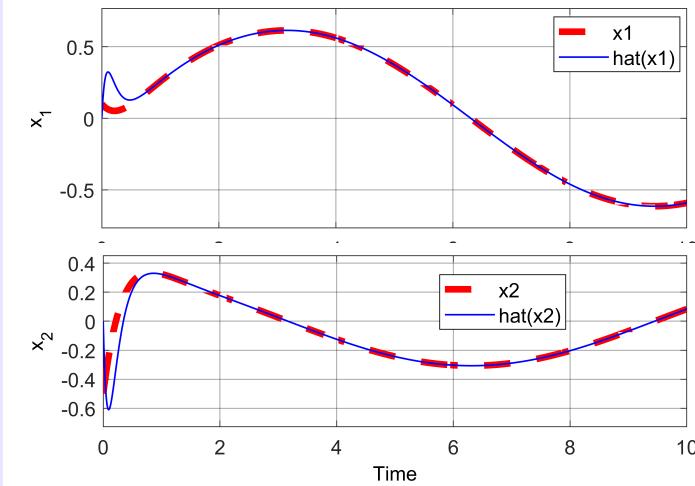


Figure 9.16 Convergence of the Luenberger state estimator

9.4 Complementary Filters and Altitude

9.3.4 Comparing the Luenberger Observer with the Kalman Filter

Recall the Kalman filter equations:

Model Prediction	Sensor Correction
$x_p = A_d \hat{x}_k + B_d u_k, \quad (1)$	$S = CP_p C^T + R, \quad (4)$
$P_p = A_d \hat{P}_k A_d^T + Q, \quad (2)$	$\mathcal{K} = P_p C^T S^{-1}, \quad (5)$
$y_p = Cx_p + Du_k, \quad (3)$	$\hat{x}_{k+1} = x_p + \mathcal{K}(y_k - y_p), \quad (6)$
	$\hat{P}_{k+1} = (I_n - \mathcal{K}C) P_p, \quad (7)$

Consider specifically the state update equation:

$$\hat{x}_{k+1} = x_p + \mathcal{K}(y_k - y_p) \quad (9.27)$$

Let's assume that $D = 0$ such that $y_p = Cx_p$. Then Equation (9.27) can be written as follows:

$$\hat{x}_{k+1} = x_p + \mathcal{K}(y_k - Cx_p) \quad (9.28)$$

Now, instead of using the model's prediction x_p to compare with the measurement y_p , let's use the state update \hat{x}_k from the previous time-step:

$$\hat{x}_{k+1} = x_p + \mathcal{K}(y_k - C\hat{x}_k) \quad (9.29)$$

The final step is to insert the state prediction $x_p = A_d \hat{x}_k + B_d u_k$:

$$\hat{x}_{k+1} = A_d \hat{x}_k + B_d u_k + \mathcal{K}(y_k - C\hat{x}_k) \quad (9.30)$$

Notice the similarity to the discrete implementation of the Luenberger observer:

$$\hat{x}_{k+1} = A_d \hat{x}_k + B_d u_k + L_d(y_k - C\hat{x}_k) \quad (9.31)$$

The only difference between Eqs. (9.30) and (9.31) is L_d instead of \mathcal{K} . If the Kalman gain \mathcal{K} converges to a constant value, using a Luenberger observer with a gain of $L_d = \mathcal{K}$ would produce the optimal state estimate as calculated by the Kalman filter.

9.4 Complementary Filters and Altitude

The altitude signal can be difficult to acquire accurately and precisely. As discussed in Chapter 8, both the GPS sensor and the barometric pressure sensor can determine altitude. Unfortunately, in the absence of differential GPS, which can be expensive and heavy, GPS measurements of altitude are not always reliable or precise. Even without obstructions from GPS satellites, it can take a GPS receiver a few minutes to get a reliable signal. Altitude calculated from pressure measurements can be more precise, but barometric pressure and its prediction of altitude change with the weather. Also, dynamic pressures caused by moving air can cause fluctuations in altitude signals. It is important to position the pressure sensor in a location inside the body of the plane that is isolated from moving air. The pressure sensor is sensitive to changes

in temperature. So if the sensor has been in an air-conditioned location and then moved outdoors, it may take a while to adjust to the temperature change. Any of these issues can cause drift in the pressure sensor's estimate of altitude.

To demonstrate some of these issues, Figure 9.17 shows a graph with two signals. The top signal shows altitude that was acquired using a BMP280 barometric pressure sensor. The bottom signal shows altitude from a GOOUEU TECH GT-U7 GPS sensor. Both sensors were placed together at a constant altitude of 1507 m for the entire 10 minutes and 30 seconds that the data were collected. Temperature changes caused the significant drift in the pressure sensor's measurement of altitude. The test was initiated immediately after the sensors were moved from inside at a temperature of 20°C to the windy and cloudy but unobstructed outdoors at a temperature of -5°C. The GPS signal shows a slowly changing random uncertainty of about ± 10 m in altitude.

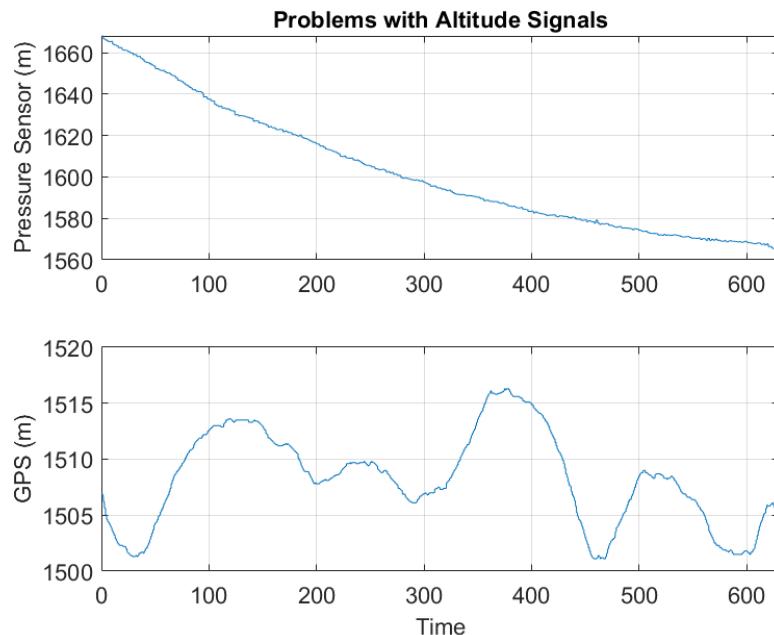


Figure 9.17 Altitude signals from a pressure sensor (top graph) and a GPS sensor (bottom graph). Both signals were measured while the sensors remained at a constant altitude of 1507 m. Temperature transients from 20°C to -5°C caused the pressure sensor's estimate to drift. The changes in the GPS signal highlight the variability and poor tolerances of the GPS signal's estimate of altitude.

Why Temperature Affects Altitude Signals from Pressure Sensors

Local static air pressure P changes with altitude h above sea level according to the hydrostatic pressure equation:

$$P_0 - P = \rho gh \quad (9.32)$$

where P_0 (Pa, often set to be 101325 Pa) is the pressure at sea level, P (Pa) is the pressure at an altitude h (m) above sea level, ρ (kg/m^3) is the air density, and g (m/s^2) is the local gravitational constant. Air density ρ , however, is a function of local pressure P and temperature T (K). By

9.4 Complementary Filters and Altitude

approximating air as an ideal gas, the ideal gas law calculates the air density to be

$$\rho = \frac{P}{RT} \quad (9.33)$$

where $R = 287 \text{ J/(kg K)}$ is the ideal gas constant for air. Combining Eqs. (9.32) and (9.33) results in the following equation for altitude h as a function of temperature T :

$$h = \frac{P_0 - P}{Pg} RT \quad (9.34)$$

This equation and a slowly decreasing temperature from about 20°C to -5°C agrees with the drifting changes in the altitude signal of Figure 9.17. Because of this equation, accurate altitude signals require the pressure sensor to be at the same temperature as its surrounding air.

The pressure sensor should be at the same temperature as its surrounding air. It should also be protected from dynamic pressures caused by moving air. Under these ideal conditions, the pressure sensor can often detect relative altitude changes more precisely than the GPS sensor (unless differential GPS is available). For example, the BMP280 sensor has a relative precision of about $\pm 1 \text{ m}$. The GOOUII TECH GT-U7 GPS sensor that was tested had a relative precision of about $\pm 20 \text{ m}$. The absolute altitude, however, is often more accurately measured by a GPS sensor. For example, in the measurements of Figure 9.17 the GOOUII TECH GT-U7 GPS sensor had an absolute accuracy of about $\pm 20 \text{ m}$. The BMP280 sensor's prediction had a much larger error of about 50 m even after the pressure sensor converged to the surrounding air temperature. This complementary relationship can be leveraged to provide a measurement that has both good relative precision (about $\pm 1 \text{ m}$) and good absolute accuracy (about $\pm 20 \text{ m}$). A complementary filter can accomplish this.

9.4.1 The Complementary Filter

Mathematically, a complementary filter is a weighted average of two or more signals. For two signals, h_1 and h_2 , the complementary filter calculates a weighted average h .

$$h = \kappa h_1 + (1 - \kappa) h_2 \quad (9.35)$$

Although κ could simply be a number, for example, $0 < \kappa < 1$, it could also be an operator. For example, κ could represent a Laplace domain transfer function such as $\kappa = \frac{1}{\tau s + 1}$ where s is the Laplace independent variable and $\tau \geq 0$ is a time-constant. If so, the complementary filter is the combination of a low-pass filtered signal $\frac{1}{\tau s + 1} h_1$ and a high-pass filtered signal $(1 - \frac{1}{\tau s + 1}) h_2$:

$$h = \frac{1}{\tau s + 1} h_1 + \left(1 - \frac{1}{\tau s + 1}\right) h_2 \quad (9.36)$$

$$= \frac{1}{\tau s + 1} h_1 + h_2 - \frac{1}{\tau s + 1} h_2 \quad (9.37)$$

If we define $h_{1,L}$ as the low-pass filtered h_1 signal

$$h_{1,L} = \frac{1}{\tau s + 1} h_1 \quad (9.38)$$

and $h_{2,L}$ as a low-pass filtered h_2 signal

$$h_{2,L} = \frac{1}{\tau s + 1} h_2 \quad (9.39)$$

then Eq. (9.37) becomes

$$h = h_{1,L} + h_2 - h_{2,L} \quad (9.40)$$

When κ is a Laplace transfer function, we may need to use a numerical method to implement the complementary filter. The next section uses Backwards Euler integration to solve the low-pass filters $h_{1,L}$ and $h_{2,L}$ of Eqs. (9.38) and (9.39).

9.4.2 Backward Euler Implementation of a Low-Pass Filter

A Laplace domain representation of a first-order low-pass filter, $h_{1,L} = \frac{1}{\tau s + 1} h_1$, can be written in state-space form:

$$\dot{h}_{1,L} = -\frac{1}{\tau} h_{1,L} + \frac{1}{\tau} h_1 \quad (9.41)$$

The backward Euler implementation of this differential equation is

$$\frac{h_{1,L,k+1} - h_{1,L,k}}{\Delta t} = -\frac{1}{\tau} h_{1,L,k+1} + \frac{1}{\tau} h_{1,k} \quad (9.42)$$

and its solution is

$$h_{1,L,k+1} = \frac{\tau}{\tau + \Delta t} h_{1,L,k} + \frac{\Delta t}{\tau + \Delta t} h_{1,k} \quad (9.43)$$

where $\Delta t = t_{k+1} - t_k$ is the integration time-step, $h_{1,k}$ is the value of the signal h_1 at timestep t_k , $h_{1,L,k}$ is the value of $h_{1,L}$ at timestep t_k , and $h_{1,L,k+1}$ is the value of $h_{1,L}$ at timestep t_{k+1} .

Following the same process, the solution for the low-pass filter signal $h_{2,L}$ is

$$h_{2,L,k+1} = \frac{\tau}{\tau + \Delta t} h_{2,L,k} + \frac{\Delta t}{\tau + \Delta t} h_{2,k} \quad (9.44)$$

9.4.3 Complementary Filter For Altitude Sensor Fusion

Averaged over a long duration, the GPS altitude signal h_{GPS} has good absolute accuracy. The GPS altitude signal, therefore, will be the low-pass filtered signal $h_1 = h_{GPS}$ in the complementary filter of Eq. (9.37). The pressure altitude signal h_P has an offset bias, but it has good relative precision; therefore, it will be the high-pass filtered signal $h_2 = h_P$. Combining Eqs. (9.43) and (9.44) with Eq. (9.40) results in a sensor fusion algorithm for improved altitude estimation.

Complementary Filter Algorithm for Altitude Sensor Fusion

At any timestep t_k the values of $h_{1,L,k}$ and $h_{2,L,k}$ must be known. Therefore, these signals require initial conditions. One approach is to set $h_{1,L,0}$ equal to the first altitude value measured by the GPS sensor h_{GPS} . If the actual altitude is known, it should be used instead. The initial condition $h_{2,L,0}$ can be set to the first altitude value measured by the pressure sensor h_P .

The values at the next time instance t_{k+1} can be calculated recursively:

$$h_k = h_{1,L,k} + h_{P,k} - h_{2,L,k} \quad (9.45)$$

where $h_{P,k}$ is the pressure sensor's measurement of altitude at time t_k .

9.4 Complementary Filters and Altitude

The signal $h_{1,L,k}$ transitions as follows:

$$h_{1,L,k+1} = \frac{\tau}{\tau + \Delta t} h_{1,L,k} + \frac{\Delta t}{\tau + \Delta t} h_{GPS,k} \quad (9.46)$$

where $h_{GPS,k}$ is the GPS sensor's measurement of altitude at time t_k .

The signal $h_{2,L,k}$ transitions as follows:

$$h_{2,L,k+1} = \frac{\tau}{\tau + \Delta t} h_{2,L,k} + \frac{\Delta t}{\tau + \Delta t} h_{P,k} \quad (9.47)$$

9.4.4 Altitude Complementary Filter Experimental Results

Altitude measurements from a GPS sensor were collected into a vector h_{GPS} . Corresponding altitude measurements from a pressured sensor were collected into a vector h_P . The experimental test included collecting data for 150 seconds while waiting at the bottom of a small 8 m hill. The sensors were then moved to the top of the hill over a period of about 50 seconds. Then the sensors were held on the top of the hill for 25 seconds before being returned to the bottom of the hill over a period of about 15 second. The results are shown in Figure 9.18. The following MATLAB code was used to implement the complementary filter:

```
N = length(t);
h1L = hGPS(1);
h2L = hP(1);
h = zeros(size(hP));
h(1) = hGPS(1);
tau = 100;
for k = 2:N
    h(k) = h1L + hP(k) - h2L;
    dt = t(k)-t(k-1);
    h1L = tau/(tau+dt)*h1L+dt/(tau+dt)*hGPS(k);
    h2L = tau/(tau+dt)*h2L+dt/(tau+dt)*hP(k);
end

figure
plot(t,hGPS,t,hP, t, h)
legend('h_GPS', 'h_P', 'h', "Location", "best")
xlabel('Time (s)')
ylabel('Altitude (m)')
grid on
title('Complementary Filter Results')
```

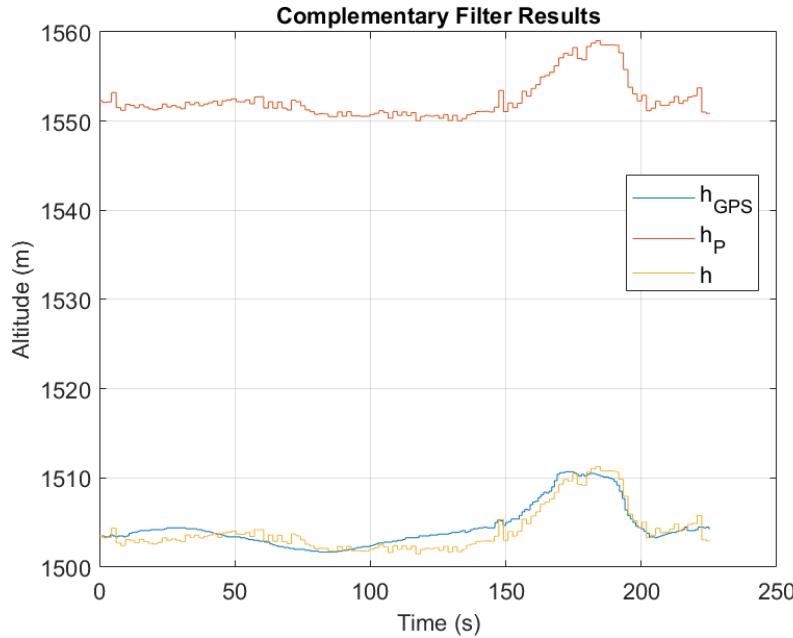


Figure 9.18 The filtered altitude signal h has the absolute accuracy of the GPS signal h_{GPS} and the relative precision of the pressure signal h_P .

9.5 Estimating Gravity

The following sections are optional because orientation can be estimated with a simpler, less accurate estimate of gravity given in Eq. (11.8). However, these sections present algorithms that can calculate better estimates of the gravity vector. A better gravity signal improves the orientation estimation algorithms presented in other chapters of this book.

9.5.1 Deriving the Gravity Estimation State-Equations

Because of the motion of the airplane, accurately estimating gravity from accelerometer measurements is one of the most challenging parts of orientation estimation. In Section 10.5, Eq. (10.21) presented an equation relating gravity to the accelerometer measurement. The equation can be written in vector form:

$$\dot{V} = -\omega \times V + g - a \quad (9.48)$$

where \dot{V} (m/s^2) is the time-derivative of the body-frame velocity vector $V = [u \ v \ w]^T$ (m/s), $\omega = [\omega_x \ \omega_y \ \omega_z]^T$ (rad/s) is the angular velocity vector, g (m/s^2) is the body-frame gravity vector, and a (m/s^2) is the accelerometer signal. The variables u , v , and w (m/s) are the velocities in the bod-fixed x, y, and z-axes respectively. By definition of the cross product, Eq. (9.48) can also be written as

$$\dot{V} = -\Omega V + g - a \quad (9.49)$$

where the matrix Ω is

9.5 Estimating Gravity

$$\Omega = \begin{bmatrix} 0 & -\omega_z & \omega_y \\ \omega_z & 0 & -\omega_x \\ -\omega_y & \omega_x & 0 \end{bmatrix} \quad (9.50)$$

A variety of discrete-time numerical methods could be used to solve Eq. (9.49). Although explicit methods, such as forward Euler, are computationally simpler, they are more susceptible to instability issues than implicit methods, such as backward Euler numerical integration. These numerical instabilities could lead to significant errors. The backward Euler form of Eq. (9.49) is

$$\frac{V_k - V_{k-1}}{\Delta t} = -\Omega_k V_k + g_{k-1} - a_k \quad (9.51)$$

which is solved for the velocity V_k :

$$V_k = (I + \Delta t \Omega_k)^{-1} V_{k-1} + \Delta t (I + \Delta t \Omega_k)^{-1} g_{k-1} - \Delta t (I + \Delta t \Omega_k)^{-1} a_k \quad (9.52)$$

The subscript k relates to the present iteration of the microcontroller. The gravity vector g_k (m/s^2) in the body-frame can be written as a function of the rotation matrix $R_{I,k}^b$, which is the inertial-to-body rotation matrix (see Eq. (10.15)):

$$g_k = R_{I,k}^b \begin{bmatrix} 0 \\ 0 \\ 9.8 \end{bmatrix} \quad (9.53)$$

where Eq. (9.53) assumes that the local gravitational acceleration is 9.8 (m/s^2). Section 11.2 introduced the Rodrigues rotation matrix δR :

$$\delta R \approx I - \Delta t \Omega_k \quad (9.54)$$

Section 11.2 showed that the present rotation matrix $R_{I,k}^b$ can be written as a product of the Rodrigues matrix and the previous rotation matrix $R_{I,k-1}^b$:

$$\begin{aligned} R_{I,k}^b &= \delta R R_{I,k-1}^b \\ &= (I - \Delta t \Omega_k) R_{I,k-1}^b \end{aligned} \quad (9.55)$$

Substituting Eq. (9.55) into Eq. (9.53) gives

$$g_k = \delta R R_{I,k-1}^b \begin{bmatrix} 0 \\ 0 \\ 9.8 \end{bmatrix} \quad (9.56)$$

Recognizing that the previous gravity vector is $g_{k-1} = R_{I,k-1}^b \begin{bmatrix} 0 \\ 0 \\ 9.8 \end{bmatrix}$, Eq. (9.56) can be rewritten as

$$\begin{aligned} g_k &= \delta R g_{k-1} \\ &= (I - \Delta t \Omega_k) g_{k-1} \end{aligned} \quad (9.57)$$

which is the forward Euler numerical approximation of

$$\dot{g} = -\omega \times g \quad (9.58)$$

In fact, notice that in the limit as $\Delta t \rightarrow 0$, Eq. (9.57) becomes Eq. (9.58). For improved numerical stability, the algorithm of this chapter uses the following backward Euler solution to Eq. (9.58) instead of the forward Euler solution given in Eq. (9.57):

$$g_k = (I + \Delta t \Omega_k)^{-1} g_{k-1} \quad (9.59)$$

Eqs. (9.52) and (9.59) are the state equations for a Kalman filter that provides an improved estimate of gravity. A better gravity signal improves estimates of quaternion orientation. The state equations Eqs. (9.52) and (9.59) are combined in a block-matrix format as follows:

$$\begin{bmatrix} V_k \\ g_k \end{bmatrix} = \begin{bmatrix} (I + \Delta t \Omega_k)^{-1} & \Delta t (I + \Delta t \Omega_k)^{-1} \\ 0 & (I + \Delta t \Omega_k)^{-1} \end{bmatrix} \begin{bmatrix} V_{k-1} \\ g_{k-1} \end{bmatrix} + \begin{bmatrix} -\Delta t (I + \Delta t \Omega_k)^{-1} \\ 0 \end{bmatrix} a_k \quad (9.60)$$

From these state-equations, we will define the state-transition (A_d) and input-transition (B_d) matrices, which will be used in a Kalman filter in Section 9.6:

$$A_d = \begin{bmatrix} (I + \Delta t \Omega_k)^{-1} & \Delta t (I + \Delta t \Omega_k)^{-1} \\ 0 & (I + \Delta t \Omega_k)^{-1} \end{bmatrix} \quad (9.61)$$

$$B_d = \begin{bmatrix} -\Delta t (I + \Delta t \Omega_k)^{-1} \\ 0 \end{bmatrix} \quad (9.62)$$

9.5.2 Sensor Measurements for Gravity Estimation

Accelerometers measure the acceleration signal a_k (m/s²), which is the input to the state equations, Eq. (9.60). GPS sensors measure latitude ϕ_k (rad), longitude λ_k (rad), and altitude h_{GPS} (m). An altimeter (pressure sensor) also measures altitude h_P (m). The methods of Section 9.4 combine the signals h_{GPS} and h_P for an improved estimate of altitude h_k (m). Latitude, longitude, and altitude are converted to inertial velocities, $V_{I,k}$ (m/s), using Eqs. (8.17) and (8.18). It can be used to approximate an estimate \hat{V}_k (m/s) of the body-fixed velocity vector.

$$\hat{V}_k = \begin{bmatrix} \|V_{I,k}\| \\ 0 \\ 0 \end{bmatrix} \quad (9.63)$$

The gyrometer measures the angular velocity signal ω_k (rad/s). An estimate $\hat{g}_{a,k}$ (m/s²) of the gravity vector g_k can be calculated as follows:

$$\hat{g}_{a,k} = 9.8 \frac{a_k + \omega_k \times \hat{V}_k}{\|a_k + \omega_k \times \hat{V}_k\|} \quad (9.64)$$

9.6 Gravity Estimation using a Kalman Filter

9.6 Gravity Estimation using a Kalman Filter

The discrete-state space form of the accelerometer equations Eqs. (9.60) and (9.61) provides an enabling framework for Kalman filtering. The state variables, \hat{x}_k , estimated by the Kalman filter are the body-frame velocities V_k (m/s) and the gravity vector g_k in the body-frame:

$$\hat{x}_k = \begin{bmatrix} V_k \\ g_k \end{bmatrix} \quad (9.65)$$

The Kalman filtering algorithm consists of the following sequential equations:

$$x_p = A_d \hat{x}_{k-1} + B_d a_k \quad (9.66)$$

$$P_p = A_d \hat{P}_{k-1} A_d^T + Q \quad (9.67)$$

$$C = \begin{cases} I_6 & \text{if new GPS data arrive} \\ [0_{3 \times 3} \ I_3] & \text{Otherwise} \end{cases} \quad (9.68)$$

$$S = \begin{cases} CP_p C^T + R_6 & \text{if new GPS data arrive} \\ CP_p C^T + R_3 & \text{Otherwise} \end{cases} \quad (9.69)$$

$$\mathcal{K} = P_p C^T (S)^{-1} \quad (9.70)$$

$$\hat{x}_k = \begin{cases} x_p + \mathcal{K} \left(\begin{bmatrix} \hat{V}_k \\ \hat{g}_{a,k} \end{bmatrix} - C x_p \right) & \text{if new GPS data arrive} \\ x_p + \mathcal{K} (\hat{g}_{a,k} - C x_p) & \text{Otherwise} \end{cases} \quad (9.71)$$

$$\hat{P}_k = (I_6 - \mathcal{K} C) P_p \quad (9.72)$$

The intermediate calculation x_p is the model's prediction of the state variables. The updated prediction \hat{x}_k is the Kalman filter's estimate of the states V_k (m/s) and g_k at the k^{th} time-iteration. P_p is the model's prediction of the state noise covariance. \hat{P}_k is the Kalman filter's estimate of the state noise covariance. The matrix S is an intermediate step in calculating the Kalman gain \mathcal{K} . The matrix I_6 is the 6×6 identity matrix and I_3 is the 3×3 identity matrix. The matrix $0_{3 \times 3}$ is the 3×3 matrix of zeros. The gravity estimate $\hat{g}_{a,k}$ is calculated using Eq. (9.64), and \hat{V}_k is from Eq. (9.63). It is important to note that the acceleration vector a_k (m/s^2) is measured using the accelerometer, the inertial velocities vector $V_{I,k}$ is calculated using Eqs. (8.17) and (8.18), and the rotation matrix \hat{R}_I^b is from Eq. (11.2). The Kalman filter state \hat{x}_k includes estimates of the velocities u, v, w (m/s) and the body-fixed gravity signals g_x, g_y , and g_z (m/s^2).

The Kalman filter requires the process noise covariance matrix Q and sensor noise covariance R , both of which are positive definite matrices. When new GPS data is available, the sensor noise covariance matrix is a 6×6 matrix R_6 , which includes the variances of both the GPS velocities and the accelerometer signals. If new GPS data is not available, the sensor noise covariance matrix is a 3×3 matrix R_3 , which includes only the variance of the accelerometer signals. These covariance matrices are treated as tuning parameters for the Kalman filter. Approximating the state variables as being independent results in Q being a diagonal matrix. If the accelerometer measurements are independent, R is a diagonal matrix. For example, with a sampling time of about $\Delta t = 0.006$ seconds, the following covariance matrices are suggested as a starting point for tuning:

$$Q = \begin{bmatrix} 0.1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0.1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0.1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0.003 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0.003 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0.003 \end{bmatrix} \quad (9.73)$$

$$R_6 = \begin{bmatrix} 0.01 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0.01 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0.01 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \quad (9.74)$$

$$R_3 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad (9.75)$$

9.6.1 Simulation Example

Data collected from a flight simulator was processed using the Kalman filtering algorithm. The resulting estimates of the velocities are shown in Figure 9.19. Figure 9.20 shows the Kalman filter's estimate of the gravity components. The Kalman filter approach of this chapter significantly improves the gravity estimates compared to the prediction based on Eq. (10.23), especially in the body-fixed y and z axes.

9.6.2 Kalman Filtering Results: Actual Flight Data

The Kalman filtering approach to estimating the gravity signals can help reduce noise caused by propeller vibrations as well. Figure 9.22 compares the filtered gravity estimates from actual airplane data versus estimates calculated by Eq. (10.23). The effect of propeller vibrations is significantly reduced in the filtered estimates. This reduction is especially evident in the g_x graph. Similar to the simulation results of Figure 9.20, the Kalman filtering algorithms for the g_y and g_z graphs predict larger banking and pitching angles than estimates based on Eq. (10.23).

Figure 9.21 shows the Kalman filter estimates of body-frame velocities.

9.6 Gravity Estimation using a Kalman Filter

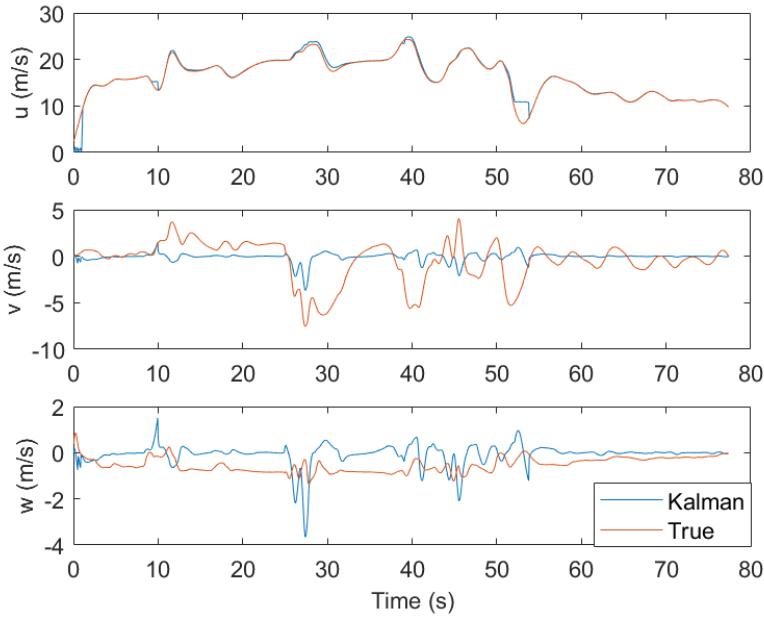


Figure 9.19 Results of the Kalman filter in estimating the body-fixed velocities from a flight simulator. The Kalman filter estimates are labeled ‘Kalman’. The simulated velocities are labeled ‘True’.

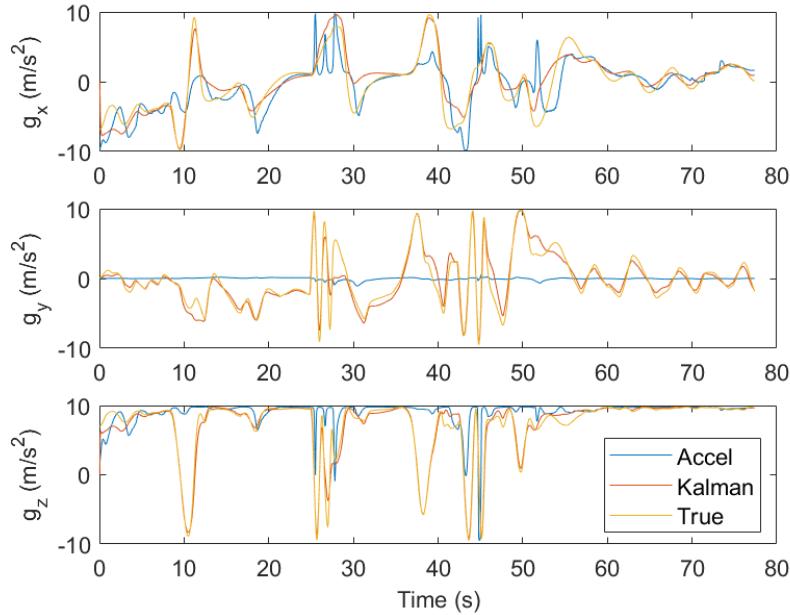


Figure 9.20 Results of the Kalman filter in estimating the body-fixed gravitational accelerations. The Kalman filter estimates are labeled ‘Kalman’. The actual simulator results are labeled ‘True’. The signals labeled ‘Accel’ are estimates of gravity based on Eq. (10.23).

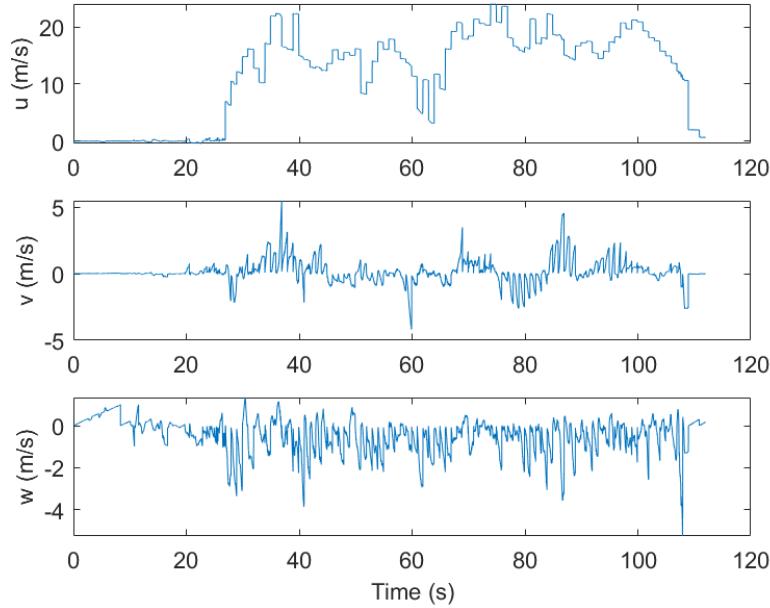


Figure 9.21 The Kalman filter estimates the body-frame velocities of an actual flight.

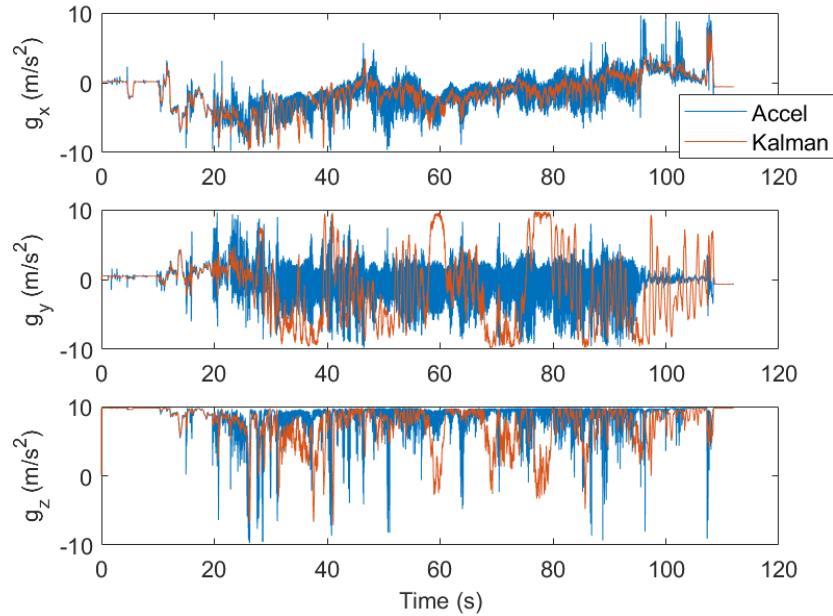


Figure 9.22 The Kalman filter decreases the effect of propeller vibrations in estimating the body-fixed gravitational accelerations and predicts larger banking and pitching angles. The Kalman filter estimates are labeled 'Kalman'. The signals labeled 'Accel' are estimates of gravity based on Eq. (10.23).

9.7 Gravity Estimation Using a Low Pass Filter

9.7 Gravity Estimation Using a Low Pass Filter

Using a Kalman filter to estimate the gravity signal is effective, but also computationally expensive. This section derives a gravity estimation algorithm that uses a computationally simpler approach. It uses a low pass filter to remove high-frequency acceleration from the accelerometer signal. High frequency terms cause large rates of change in velocity making \dot{V} have large values. Low-pass filtering the accelerometer signal makes the \dot{V} term in Eq. (9.48) less important. As an approximation, then, \dot{V} is neglected, and Eq. (9.48) can be approximated as

$$0 \approx -\omega \times V + g - a_{LP} \quad (9.76)$$

where a_{LP} is the low-pass filtered accelerometer signal. Using the derivation of Section 9.4.2, the backwards Euler equation for a_{LP} is

$$a_{LP,k+1} = \frac{\tau}{\tau + \Delta t} a_{LP,k} + \frac{\Delta t}{\tau + \Delta t} a_k \quad (9.77)$$

Where τ (s) is the low-pass filter time constant. A value of $\tau = 0.2$ s is a good starting point. We rearrange Eq. (9.76) to solve for gravity g :

$$g \approx a_{LP} + \omega \times V \quad (9.78)$$

The body-frame velocity vector V is approximated as discussed in Eq. (9.63). The gravity signal is normalized because it should always have a magnitude of 1 g. The complete gravity estimation algorithm is provided below.

Gravity Estimation using a Low Pass Filter

An accelerometer measures the body-frame x-, y-, and z-axis accelerations as the vector a_k at timestep t_k . Latitude, longitude, and altitude from a GPS sensor are converted to inertial velocities, $V_{I,k}$ (m/s), using Eqs. (8.17) and (8.18). The estimate \hat{V}_k (m/s) of the body-fixed velocity vector is approximated as

$$\hat{V}_k = \begin{bmatrix} \|V_{I,k}\| \\ 0 \\ 0 \end{bmatrix} \quad (9.79)$$

The low-pass filter equation needs an initial condition, and so it is set to zero: $a_{LP,0} = 0$. The accelerometer signal a_k is low-pass filtered using a backwards Euler algorithm. The low-pass filtered acceleration signal $a_{LP,k}$ is updated recursively as follows:

$$a_{LP,k+1} = \frac{\tau}{\tau + \Delta t} a_{LP,k} + \frac{\Delta t}{\tau + \Delta t} a_k \quad (9.80)$$

The gyrometer measures the x-, y-, and z-axis angular velocities (rad/s) and stores them in the vector ω_k . The accelerometer's estimate of gravity $\hat{g}_{a,k}$ is the low pass filtered accelerometer signal $a_{LP,k}$ added to the cross-product of the gyrometer signal ω_k and the body frame velocity estimate \hat{V}_k :

$$\hat{g} = a_{LP,k} + \omega_k \times \hat{V}_k \quad (9.81)$$

The gravity estimate is normalized to have a magnitude of 1 g:

$$\hat{g}_{a,k} = \frac{\hat{g}}{\|\hat{g}\|} \quad (9.82)$$

The gravity estimate $\hat{g}_{a,k}$ replaces the gravity estimate $g_{a,k} = a_k / \|a_k\|$ in the quaternion estimation algorithm.

9.7.1 Simulation and Experimental Results

The low pass filtering algorithm for gravity estimation was applied to the same simulation and experiments as Sections 9.6.1 and 9.6.2. The simulator results are shown in Figures 9.23 and 9.24. The experimental results are shown in Figures 9.25 and 9.26.

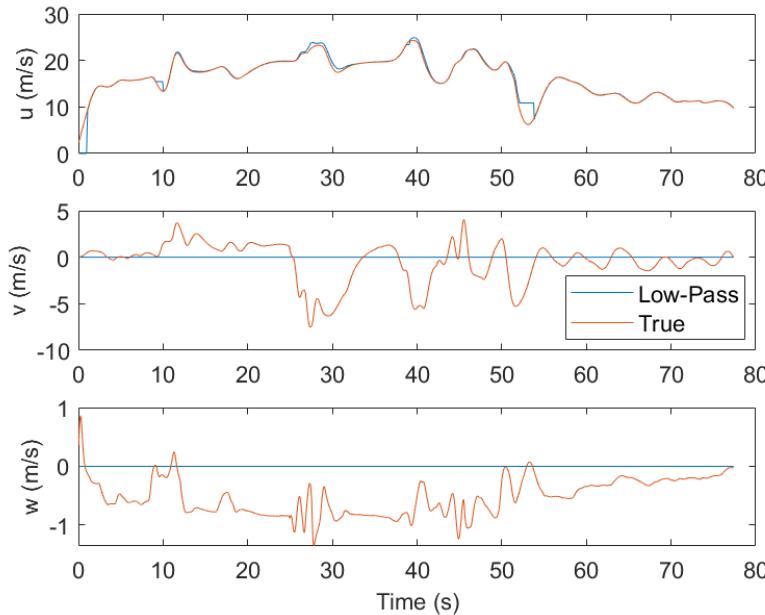


Figure 9.23 Results of the low-pass filter in estimating the body-fixed velocities from a flight simulator. The low-pass filter estimates are labeled ‘Low-Pass’. The simulated velocities are labeled ‘True’.

9.7 Gravity Estimation Using a Low Pass Filter

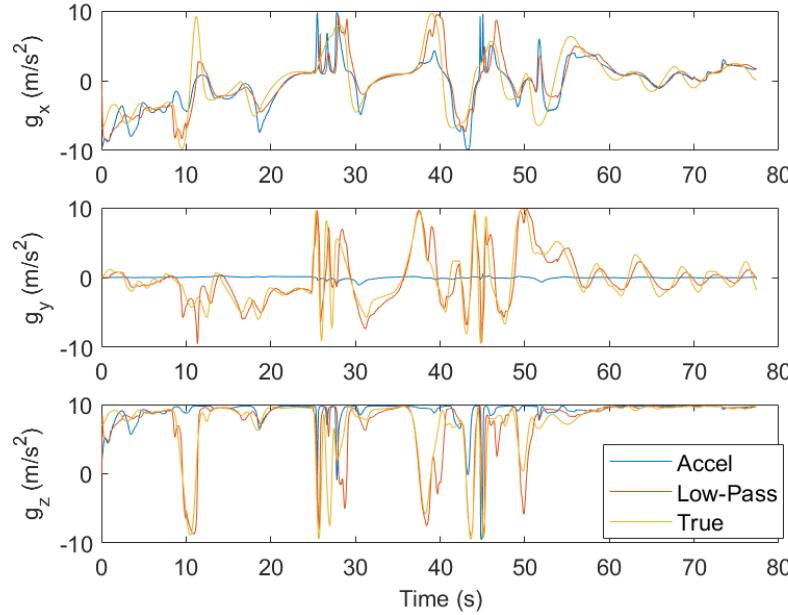


Figure 9.24 Results of the low-pass filter in estimating the body-fixed gravitational accelerations. The low-pass filter estimates are labeled ‘Low-Pass’. The actual simulator results are labeled ‘True’. The signals labeled ‘Accel’ are estimates of gravity based on Eq. (10.23).

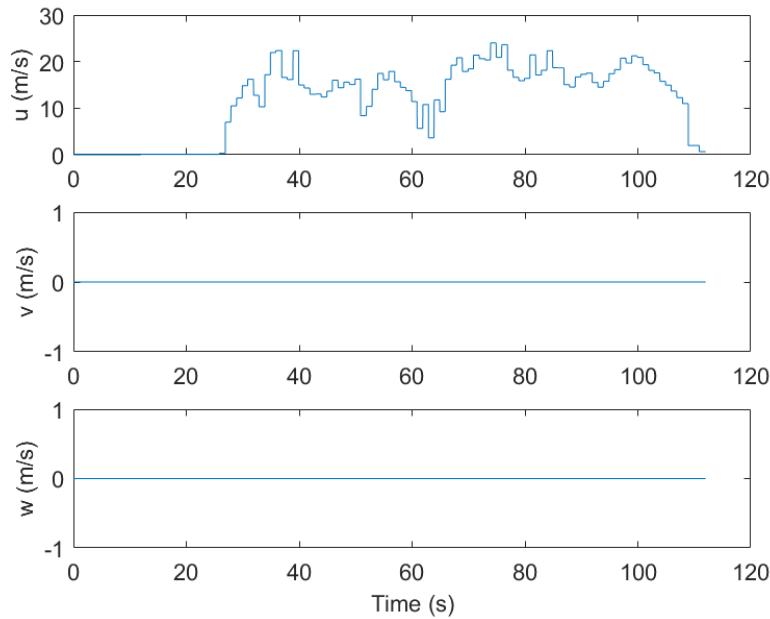


Figure 9.25 The low-pass filter estimates the body-frame velocities of an actual flight.

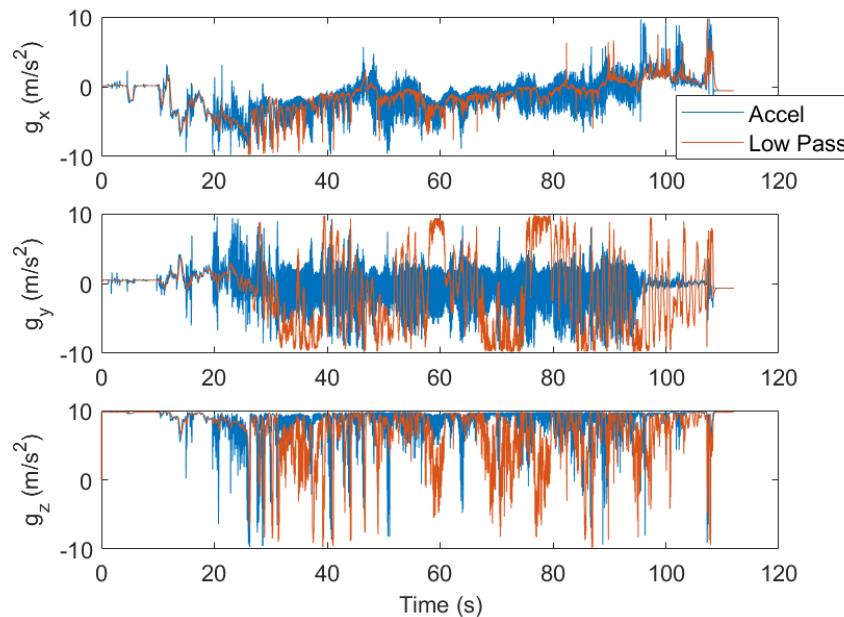


Figure 9.26 The low-pass filter decreases the effect of propeller vibrations in estimating the body-fixed gravitational accelerations and predicts larger banking and pitching angles. The low-pass filter estimates are labeled ‘Low Pass’. The signals labeled ‘Accel’ are estimates of gravity based on Eq. (10.23).

Chapter 10

Background for Estimating Orientation

Contents

10.1 Prediction: Quaternion Prediction with a Gyrometer	326
10.2 Prediction: Gyrometer Prediction of the Rotation Matrix	328
10.3 Prediction: Gyrometer Prediction of Gravity	328
10.4 Prediction: Gyrometer Prediction of Geomagnetic Vector	329
10.5 Measurement: Accelerometer Measurement of Gravity	330
10.6 Measurement: Magnetometer Measurement of Geomagnetic Vector	331
10.7 Update: Quaternion from Gravity and Geomagnetic Vector	332
10.7.1 Quaternion Orientation from a Rotation Matrix	333

This chapter discusses how to estimate quaternion orientation using a 9 Degree Of Freedom (9dof) sensor, *e.g.*, Figure 10.1. The 9dof sensor includes a 3-axis magnetometer, a 3-axis gyrometer (angular rate sensor), and a 3-axis accelerometer (for measuring gravity). Algorithms of this type are sometimes referred to as MARG (Magnetometer, Angular Rate, and Gravity) algorithms or AHRS (Attitude and Heading Reference Systems) filters.



Figure 10.1 A 9dof sensor (FXOS8700 + FXAS21002) by Adafruit

Gyrometer data is typically less noisy than gravity data from an accelerometer. Integrating gyrometer data provides an estimate of orientation, but it is susceptible to long term integration drift. On the other hand, the combination of magnetometer and gravity data provides an independent estimate of orientation that is relatively noisy, but it does not suffer from integration drift. MARG algorithms fuse the information from these three sensors together to obtain a cleaner estimate of orientation that does not drift.

Various algorithms have been derived to estimate orientation. Generally, these algorithms can be categorized into one of three groups: (1) complementary filters, (2) extended or unscented Kalman filters, and (3) error state (indirect) Kalman filters.

Of these three groups, the complementary filters are the easiest to understand, easiest to program, easiest to tune, and require the least resources computationally. They can be robust and provide accurate estimates of orientation. A particularly simple and robust algorithm presented in this chapter is an adaptation of the algorithm developed by Kok and Schön¹.

The extended and unscented Kalman filters are easier to understand than the error state Kalman filters. They can also be easier to tune. The unscented Kalman filters provide an especially accurate, but computationally expensive solution. These algorithms have significantly more tuning parameters than the complementary filters. The unscented Kalman filters are more capable of handling the large nonlinearities of the quaternion state equations than the extended Kalman filters, and are therefore more accurate.

The error state Kalman filters are the most challenging conceptually because they operate on the errors of the dynamic equations rather than the dynamic equations themselves. The main benefit is that the error dynamics are more linear. Error state filters can be less computationally expensive than the unscented Kalman filters. They are the most difficult to tune, however, because they have the largest number of tuning parameters of all the algorithms. For MARG algorithms, the error state Kalman filter requires 12 by 12 process covariance matrices and 6 by 6 sensor covariance matrices that need to be parameterized.

This chapter provides a background for Chapter 11, which will explain how to estimate orientation from 9dof sensors. In orientation algorithms, the gyrometer data is integrated to predict the quaternion orientation. Then gravity and magnetometer data is used to correct the predictions.

10.1 Prediction: Quaternion Prediction with a Gyrometer

A gyrometer, *i.e.* gyroscope, is a sensor that measures angular velocities p , q , and r . The variables p , q , and r are the angular velocities about the body-fixed x , y , and z axes respectively. There are at least two ways that a gyrometer can predict quaternion orientation. The first is to numerically integrate the quaternion state-equation which includes the sensed angular velocities. The quaternion state-equation was derived in a previous chapter. It is repeated here for the North-East-Down (NED) coordinate system.

$$\begin{bmatrix} \dot{e}_0 \\ \dot{e}_1 \\ \dot{e}_2 \\ \dot{e}_3 \end{bmatrix} = \frac{1}{2} \begin{bmatrix} -e_1 & -e_2 & -e_3 \\ e_0 & -e_3 & e_2 \\ e_3 & e_0 & -e_1 \\ -e_2 & e_1 & e_0 \end{bmatrix} \begin{bmatrix} p \\ q \\ r \end{bmatrix} \quad (10.1)$$

The second way is similar in that it also numerically integrates the angular velocities p , q , and r over one time-step Δt from t_k to t_{k+1} . Numerical integration produces incremental angle changes:

¹ Manon Kok and Thomas B. Schön, “A Fast and Robust Algorithm for Orientation Estimation using Inertial Sensors”, IEEE Signal Processing Letters, 2019. DOI: 10.1109/LSP.2019.2943995

10.1 Prediction: Quaternion Prediction with a Gyrometer

$$\Delta\theta_x = \int_{t_k}^{t_{k+1}} p dt \quad (10.2)$$

$$\Delta\theta_y = \int_{t_k}^{t_{k+1}} q dt \quad (10.3)$$

$$\Delta\theta_z = \int_{t_k}^{t_{k+1}} r dt \quad (10.4)$$

If forward Euler numerical integration is used with a small enough time-step Δt , these can be simplified to the following approximations:

$$\Delta\theta_x \approx p\Delta t \quad (10.5)$$

$$\Delta\theta_y \approx q\Delta t \quad (10.6)$$

$$\Delta\theta_z \approx r\Delta t \quad (10.7)$$

$\Delta\theta_x$, $\Delta\theta_y$, and $\Delta\theta_z$ are the x , y , and z components of the incremental changes in the orientation. The magnitude $\|\Delta\theta\|$ of the incremental change is the square root of the sum of the squares of the components:

$$\|\Delta\theta\| = \sqrt{\Delta\theta_x^2 + \Delta\theta_y^2 + \Delta\theta_z^2} \quad (10.8)$$

A quaternion has both scalar and vector parts. The scalar part is the cosine of half the rotation angle. Therefore, the incremental change in the scalar part of the quaternion is

$$\Delta e_0 = \cos\left(\frac{\|\Delta\theta\|}{2}\right) \quad (10.9)$$

The vector parts are the sine of half the rotation angle multiplied by a unit vector pointing in the same direction as the vector formed by the components of the incremental change in orientation (which are in the same direction as the components of the angular velocities p , q , and r):

$$\Delta e_1 = \frac{\Delta\theta_x}{\|\Delta\theta\|} \sin\left(\frac{\|\Delta\theta\|}{2}\right) \quad (10.10)$$

$$\Delta e_2 = \frac{\Delta\theta_y}{\|\Delta\theta\|} \sin\left(\frac{\|\Delta\theta\|}{2}\right) \quad (10.11)$$

$$\Delta e_3 = \frac{\Delta\theta_z}{\|\Delta\theta\|} \sin\left(\frac{\|\Delta\theta\|}{2}\right) \quad (10.12)$$

The variables Δe_0 , Δe_1 , Δe_2 , and Δe_3 are the four components of the incremental change in the quaternion during the time-step Δt from t_k to t_{k+1} . To predict the new quaternion at time t_{k+1} , we need to add the incremental change. This is done by the quaternion product:

$$\begin{bmatrix} e_{0,k+1} \\ e_{1,k+1} \\ e_{2,k+1} \\ e_{3,k+1} \end{bmatrix} = \begin{bmatrix} e_{0,k} & -e_{1,k} & -e_{2,k} & -e_{3,k} \\ e_{1,k} & e_{0,k} & -e_{3,k} & e_{2,k} \\ e_{2,k} & e_{3,k} & e_{0,k} & -e_{1,k} \\ e_{3,k} & -e_{2,k} & e_{1,k} & e_{0,k} \end{bmatrix} \begin{bmatrix} \Delta e_0 \\ \Delta e_1 \\ \Delta e_2 \\ \Delta e_3 \end{bmatrix} \quad (10.13)$$

The components $e_{0,k+1}$, $e_{1,k+1}$, $e_{2,k+1}$, and $e_{3,k+1}$ are the prediction of the quaternion from the gyrometer.

10.2 Prediction: Gyrometer Prediction of the Rotation Matrix

The previous section described how the gyrometer measurement predicts the quaternion. Using the quaternion, the rotation matrix from the body to inertial frame in any coordinate system is

$$R_b^I = \begin{bmatrix} e_0^2 + e_1^2 - e_2^2 - e_3^2 & 2(e_1 e_2 - e_0 e_3) & 2(e_0 e_2 + e_1 e_3) \\ 2(e_0 e_3 + e_1 e_2) & e_0^2 - e_1^2 + e_2^2 - e_3^2 & 2(e_2 e_3 - e_0 e_1) \\ 2(e_1 e_3 - e_0 e_2) & 2(e_0 e_1 + e_2 e_3) & e_0^2 - e_1^2 - e_2^2 + e_3^2 \end{bmatrix} \quad (10.14)$$

The rotation matrix from the inertial frame to the body frame is the transpose of R_b^I :

$$R_I^b = \begin{bmatrix} e_0^2 + e_1^2 - e_2^2 - e_3^2 & 2(e_0 e_3 + e_1 e_2) & 2(e_1 e_3 - e_0 e_2) \\ 2(e_1 e_2 - e_0 e_3) & e_0^2 - e_1^2 + e_2^2 - e_3^2 & 2(e_0 e_1 + e_2 e_3) \\ 2(e_0 e_2 + e_1 e_3) & 2(e_2 e_3 - e_0 e_1) & e_0^2 - e_1^2 - e_2^2 + e_3^2 \end{bmatrix} \quad (10.15)$$

10.3 Prediction: Gyrometer Prediction of Gravity

Using the rotation matrix R_I^b that was presented in the previous section, the gyrometer readings can be used to predict the direction of gravity in the body frame. If g is the acceleration of gravity, the gravitational vector in the inertial NED coordinate system only has a component in the positive z direction:

$$\begin{bmatrix} 0 \\ 0 \\ g \end{bmatrix} \quad (10.16)$$

In the body frame, the gravity vector is the rotation matrix R_I^b multiplied by the NED gravity vector:

$$\begin{bmatrix} g_x \\ g_y \\ g_z \end{bmatrix} = \begin{bmatrix} e_0^2 + e_1^2 - e_2^2 - e_3^2 & 2(e_0 e_3 + e_1 e_2) & 2(e_1 e_3 - e_0 e_2) \\ 2(e_1 e_2 - e_0 e_3) & e_0^2 - e_1^2 + e_2^2 - e_3^2 & 2(e_0 e_1 + e_2 e_3) \\ 2(e_0 e_2 + e_1 e_3) & 2(e_2 e_3 - e_0 e_1) & e_0^2 - e_1^2 - e_2^2 + e_3^2 \end{bmatrix} \begin{bmatrix} 0 \\ 0 \\ g \end{bmatrix} \quad (10.17)$$

$$= \begin{bmatrix} 2(e_1 e_3 - e_0 e_2)g \\ 2(e_0 e_1 + e_2 e_3)g \\ (e_0^2 - e_1^2 - e_2^2 + e_3^2)g \end{bmatrix} \quad (10.18)$$

If acceleration has units of gravity (g), i.e., $g = 1g$, the body-frame gravity vector is simply the last column of the inertial to body frame rotation matrix R_I^b .

10.4 Prediction: Gyrometer Prediction of Geomagnetic Vector

Unfortunately, even on earth, the geomagnetic vector does not only have a component in the magnetic North direction, it also has a component in the Down direction. The inclination angle δ , *i.e.*, the angle at which the geomagnetic field is tilted with respect to the earth's surface, varies from 90 degrees at the earth's poles to 0 degrees at its magnetic equator. For example, the geomagnetic inclination angle δ in Rexburg, Idaho is approximately 67 degrees. This means that the geomagnetic field in Rexburg points more towards the center of the earth than towards the north, see Figure 10.2. The strength B of the magnetic field in Rexburg is approximately 53 μT .

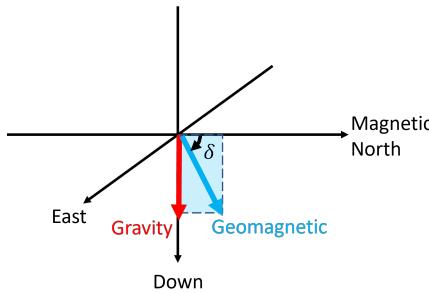


Figure 10.2 Gravitational and geomagnetic vectors in the NED reference frame

The geomagnetic vector is constant in the inertial frame. However, in the body frame, it is the rotation matrix R_I^b multiplied by the inertial geomagnetic vector:

$$\begin{bmatrix} M_x \\ M_y \\ M_z \end{bmatrix} = \begin{bmatrix} e_0^2 + e_1^2 - e_2^2 - e_3^2 & 2(e_0e_3 + e_1e_2) & 2(e_1e_3 - e_0e_2) \\ 2(e_1e_2 - e_0e_3) & e_0^2 - e_1^2 + e_2^2 - e_3^2 & 2(e_0e_1 + e_2e_3) \\ 2(e_0e_2 + e_1e_3) & 2(e_2e_3 - e_0e_1) & e_0^2 - e_1^2 - e_2^2 + e_3^2 \end{bmatrix} \begin{bmatrix} B \cos(\delta) \\ 0 \\ B \sin(\delta) \end{bmatrix} \quad (10.19)$$

In Eq. (10.19), M_x , M_y , and M_z are the gyrometer's predictions of the components of the geomagnetic field in the body frame. They have units of μT . A more useful prediction is the normalized unitless geomagnetic components m_x , m_y , and m_z which are predicted by the gyrometer as follows:

$$\begin{bmatrix} m_x \\ m_y \\ m_z \end{bmatrix} = \begin{bmatrix} e_0^2 + e_1^2 - e_2^2 - e_3^2 & 2(e_0e_3 + e_1e_2) & 2(e_1e_3 - e_0e_2) \\ 2(e_1e_2 - e_0e_3) & e_0^2 - e_1^2 + e_2^2 - e_3^2 & 2(e_0e_1 + e_2e_3) \\ 2(e_0e_2 + e_1e_3) & 2(e_2e_3 - e_0e_1) & e_0^2 - e_1^2 - e_2^2 + e_3^2 \end{bmatrix} \begin{bmatrix} \cos(\delta) \\ 0 \\ \sin(\delta) \end{bmatrix} \quad (10.20)$$

The only difference between Eq. (10.20) and Eq. (10.19) is the elimination of the field strength B from the prediction. The variables m_x , m_y , and m_z form the components of a unit vector pointing in the direction of the geomagnetic vector from the perspective of the body frame.

10.5 Measurement: Accelerometer Measurement of Gravity

Accelerometers typically measure the acceleration of gravity in addition to other linear accelerations². Extracting the gravitational signals from accelerometer measurements is challenging but essential for orientation estimation. From the perspective of this paper, accelerometers measure gravity as positive and accelerations as negative:

$$\begin{bmatrix} a_x \\ a_y \\ a_z \end{bmatrix} = \begin{bmatrix} -(\dot{u} + qw - rv) + g_x \\ -(\dot{v} + ru - pw) + g_y \\ -(\dot{w} + pv - qu) + g_z \end{bmatrix} \quad (10.21)$$

For example, consider the x-component of the accelerometer signal:

$$a_x = -(\dot{u} + qw - rv) + g_x \quad (10.22)$$

The terms in the parentheses ($\dot{u} + qw - rv$) form the total acceleration in the x-axis of the body frame. The total acceleration consists of the linear part \dot{u} and the parts caused by the rotation of the body $qw - rv$. In addition to the total acceleration, the accelerometer also includes an x-axis component of gravity g_x . The goal is to extract the gravity component.

To extract the gravity component, the total acceleration ($\dot{u} + qw - rv$) must be filtered out or removed. Fortunately, the gravity component can be extracted by using a low-pass filter with a sufficiently large time-constant. The low-pass filtered accelerometer signal provides an estimate of the gravity vector from the perspective of the body frame:

$$\begin{bmatrix} a_{x,LP} \\ a_{y,LP} \\ a_{z,LP} \end{bmatrix} \approx \begin{bmatrix} g_x \\ g_y \\ g_z \end{bmatrix} \quad (10.23)$$

In Eq. (10.23), $a_{x,LP}$, $a_{y,LP}$, and $a_{z,LP}$ are the low-pass filtered accelerometer signals in the body-fixed x, y, and z axes respectively. In sensor fusion orientation algorithms, Eq. (10.23) is compared with the gyrometer estimate of gravity Eq. (10.18) for updated estimates of orientation.

To understand why a low-pass filter can extract the gravity component from an accelerometer signal, consider what happens when the accelerometer signal is converted to the inertial frame via the rotation matrix R_b^I from the body to inertial frame:

$$R_b^I \begin{bmatrix} a_x \\ a_y \\ a_z \end{bmatrix} = \begin{bmatrix} -a_{x,I,total} \\ -a_{y,I,total} \\ -a_{z,I,total} + g_{z,I} \end{bmatrix} \quad (10.24)$$

²Piezoelectric accelerometers are an exception. They do not measure constant accelerations. However, these accelerometers are not commonly found on most inertial measurement units (IMUs).

10.6 Measurement: Magnetometer Measurement of Geomagnetic Vector

Only the inertial z-axis (in a NED reference frame) has a nonzero component of gravity $g_{z,I}$. Gravity is constant, but all the other accelerations are transient. A low-pass filter with a sufficiently long time-constant removes the transient parts of the signals. Therefore only the constant gravitational part remains.

In practice, a low-pass filter with a very large time-constant introduces a delay that causes inaccuracies in orientation algorithms. Therefore, the low-pass filter time-constant becomes an important calibration parameter that requires careful tuning. The transient accelerations are rarely entirely removed from the accelerometer signals, but are only partially attenuated; this introduces error. Fortunately, the magnetometer sensor measurement and gyrometer predictions discussed in other sections partially compensate for the gravity error when used as part of a sensor fusion algorithm. If a more accurate estimate of gravity is needed, Kalman filtering approaches, see Chapter 9.5, can provide better estimates of gravity.

10.6 Measurement: Magnetometer Measurement of Geomagnetic Vector

A magnetometer sensor measures earth's geomagnetic vector. As discussed in Section 10.4, the geomagnetic vector does not only have a component in the magnetic North direction, it also has a component in the Down direction, see Figure 10.2. The inclination angle δ , *i.e.*, the angle at which the geomagnetic field is tilted with respect to the earth's surface, varies from 90 degrees at the earth's poles to 0 degrees at its magnetic equator.

The magnetometer sensor is fixed in the body-frame. It therefore provides a reading of the geomagnetic vector from the perspective of the body-frame:

$$\begin{bmatrix} M_x \\ M_y \\ M_z \end{bmatrix} \quad (10.25)$$

This signal can be normalized by dividing each component by the magnitude, *i.e.*, geomagnetic field strength $B = \sqrt{M_x^2 + M_y^2 + M_z^2}$:

$$u_m = \begin{bmatrix} m_x \\ m_y \\ m_z \end{bmatrix} = \begin{bmatrix} \frac{M_x}{B} \\ \frac{M_y}{B} \\ \frac{M_z}{B} \end{bmatrix} \quad (10.26)$$

The variables m_x , m_y , and m_z form the components of a unit vector that point in the direction of the geomagnetic vector. The magnetometer readings of Eq. (10.26) can be compared with the gyrometer estimate of the geomagnetic unit vector in Eq. (10.20) for an updated estimate of orientation.

Magnetometer signals can be distorted by local magnetic fields, such as a motor coil or inductor, or nearby ferromagnetic materials. These cause “hard iron” and “soft iron” disturbances that bias the magnetometer readings. Magnetometer calibration is required in most cases to correct or minimize the effect of these disturbances (see Section 8.3.1).

10.7 Update: Quaternion from Gravity and Geomagnetic Vector

Section 10.5 discussed how to extract gravity from an accelerometer measurement (see Eq. (10.23)), and Section 10.6 discussed accurately measuring the geomagnetic unit vector (see Eqs. (10.26) and (8.5)). A unit vector u_g in the body frame pointing towards the center of the earth (downward) can be found by normalizing the gravity vector Eq. (10.23):

$$u_g = \begin{bmatrix} u_{g,x} \\ u_{g,y} \\ u_{g,z} \end{bmatrix} = \begin{bmatrix} \frac{g_x}{g} \\ \frac{g_y}{g} \\ \frac{g_z}{g} \end{bmatrix} \quad (10.27)$$

where $g = \sqrt{g_x^2 + g_y^2 + g_z^2}$ is the norm of the gravity vector.

The unit vector u_g is in the body frame and points in the positive z_I direction of the inertial frame. Therefore, it forms the last column of the rotation matrix from the inertial to the body frame R_I^b .

The geomagnetic unit vector Eq. (10.26) and gravity unit vector Eq. (10.27) lie in the plane between the positive z_I (Down) and positive x_I (Magnetic North) axes of the inertial frame, see Figure 10.2. Their normalized cross-product ($u_g \otimes u_m$) results in a unit vector u_E in the positive y_I (Magnetic East) direction.

$$u_E = \frac{u_g \otimes u_m}{\|u_g \otimes u_m\|} \quad (10.28)$$

More explicitly,

$$\begin{bmatrix} u_{E,x} \\ u_{E,y} \\ u_{E,z} \end{bmatrix} = \frac{1}{\sqrt{(u_{g,y}m_z - u_{g,z}m_y)^2 + (u_{g,z}m_x - u_{g,x}m_z)^2 + (u_{g,x}m_y - u_{g,y}m_x)^2}} \begin{bmatrix} u_{g,y}m_z - u_{g,z}m_y \\ u_{g,z}m_x - u_{g,x}m_z \\ u_{g,x}m_y - u_{g,y}m_x \end{bmatrix} \quad (10.29)$$

This unit vector u_E forms the second column of the rotation matrix from the inertial to the body frame R_I^b .

The normalized cross product $u_N = u_E \otimes u_g$ results in a unit vector pointing in the positive x_I (Magnetic North) axis of the inertial frame.

$$u_N = \begin{bmatrix} u_{N,x} \\ u_{N,y} \\ u_{N,z} \end{bmatrix} = \begin{bmatrix} u_{E,y}u_{g,z} - u_{E,z}u_{g,y} \\ u_{E,z}u_{g,x} - u_{E,x}u_{g,z} \\ u_{E,x}u_{g,y} - u_{E,y}u_{g,x} \end{bmatrix} \quad (10.30)$$

This unit vector u_N forms the first column of the rotation matrix from the inertial to the body frame R_I^b . The complete rotation matrix is

$$R_I^b = \begin{bmatrix} u_{N,x} & u_{E,x} & u_{g,x} \\ u_{N,y} & u_{E,y} & u_{g,y} \\ u_{N,z} & u_{E,z} & u_{g,z} \end{bmatrix} \quad (10.31)$$

10.7 Update: Quaternion from Gravity and Geomagnetic Vector

If we compare Eq. (10.31) with Eq. (10.15), we get the equation

$$\begin{bmatrix} u_{N,x} & u_{E,x} & u_{g,x} \\ u_{N,y} & u_{E,y} & u_{g,y} \\ u_{N,z} & u_{E,z} & u_{g,z} \end{bmatrix} = \begin{bmatrix} e_0^2 + e_1^2 - e_2^2 - e_3^2 & 2(e_0 e_3 + e_1 e_2) & 2(e_1 e_3 - e_0 e_2) \\ 2(e_1 e_2 - e_0 e_3) & e_0^2 - e_1^2 + e_2^2 - e_3^2 & 2(e_0 e_1 + e_2 e_3) \\ 2(e_0 e_2 + e_1 e_3) & 2(e_2 e_3 - e_0 e_1) & e_0^2 - e_1^2 - e_2^2 + e_3^2 \end{bmatrix} \quad (10.32)$$

This relationship provides a way to calculate quaternion orientation from a rotation matrix.

10.7.1 Quaternion Orientation from a Rotation Matrix

The goal of this section is to calculate the quaternion orientation e_0 , e_1 , e_2 , and e_3 from a known rotation matrix R . The variables e_0 , e_1 , e_2 , and e_3 are the four parts of a unit quaternion. The first part, e_0 is the scalar part: $e_0 = \cos(\eta)$, where η is twice the rotation angle. The remaining parts e_1 , e_2 , and e_3 are the vector parts of the quaternion. Let the rotation matrix R from the body to inertial frame be defined as

$$R = \begin{bmatrix} r_{11} & r_{12} & r_{13} \\ r_{21} & r_{22} & r_{23} \\ r_{31} & r_{32} & r_{33} \end{bmatrix} \quad (10.33)$$

Then using Eq. (10.15), we get

$$\begin{bmatrix} r_{11} & r_{12} & r_{13} \\ r_{21} & r_{22} & r_{23} \\ r_{31} & r_{32} & r_{33} \end{bmatrix} = \begin{bmatrix} e_0^2 + e_1^2 - e_2^2 - e_3^2 & 2(e_0 e_3 + e_1 e_2) & 2(e_1 e_3 - e_0 e_2) \\ 2(e_1 e_2 - e_0 e_3) & e_0^2 - e_1^2 + e_2^2 - e_3^2 & 2(e_0 e_1 + e_2 e_3) \\ 2(e_0 e_2 + e_1 e_3) & 2(e_2 e_3 - e_0 e_1) & e_0^2 - e_1^2 - e_2^2 + e_3^2 \end{bmatrix} \quad (10.34)$$

The trace of each matrix produces the equation

$$r_{11} + r_{22} + r_{33} = (e_0^2 + e_1^2 - e_2^2 - e_3^2) + (e_0^2 - e_1^2 + e_2^2 - e_3^2) + (e_0^2 - e_1^2 - e_2^2 + e_3^2) \quad (10.35)$$

$$= 3e_0^2 - (e_1^2 + e_2^2 + e_3^2) \quad (10.36)$$

$$= 4e_0^2 - (e_0^2 + e_1^2 + e_2^2 + e_3^2) \quad (10.37)$$

$$= 4e_0^2 - 1 \quad (10.38)$$

The last equation, Eq. (10.38), results from the fact that the quaternion is a unit quaternion with magnitude equal to one. If $1 + r_{11} + r_{22} + r_{33} > 0$, we can find the value of e_0 by solving Eq. (10.38):

$$e_0 = \frac{\sqrt{1 + r_{11} + r_{22} + r_{33}}}{2} \quad (10.39)$$

If the sum $1 + r_{11} + r_{22} + r_{33}$ is negative, Eq. (10.39) would result in a complex number with imaginary parts. An algorithm in the following section will prevent this possibility.

Subtracting r_{32} from r_{23} and dividing by $4e_0$ (from Eq. (10.39)) yields an equation for e_1 :

$$e_1 = \frac{r_{23} - r_{32}}{4e_0} \quad (10.40)$$

Similarly, e_2 and e_3 are

$$e_2 = \frac{r_{31} - r_{13}}{4e_0} \quad (10.41)$$

$$e_3 = \frac{r_{12} - r_{21}}{4e_0} \quad (10.42)$$

The problem with this approach is that Eq. (10.39) could result in imaginary numbers. However, a variation of the ideas presented in this section leads to an algorithm for finding the quaternion orientation e_0 , e_1 , e_2 , and e_3 from a known rotation matrix R .

Algorithm: Rotation Matrix to Quaternion

The following MATLAB algorithm calculates the e_0 , e_1 , e_2 , and e_3 from a known inertial to body frame rotation matrix R .

```

function q = function_RI2b_to_Quaternion(R)
% q = function_RI2b_to_Quaternion(R)
% This function calculates the quaternion orientation
% e0, e1, e2, and e3 from a known inertial to body
% rotation matrix R from the inertial to body frame

%calculate the trace of the rotation matrix
t = R(1,1) + R(2,2) + R(3,3);
if t > 0 %the trace is positive, no chance of imaginary numbers
    s = sqrt(t+1)*2; % calculate an intermediate parameter
    e0 = 0.25 * s; %get e0
    e1 = (R(2,3)-R(3,2))/s; %get e1
    e2 = (R(3,1)-R(1,3))/s; %get e2
    e3 = (R(1,2)-R(2,1))/s; %get e3
elseif R(1,1) > max(R(2,2), R(3,3))
    %The trace is negative...Avoid imaginary numbers
    s = sqrt(1+R(1,1)-R(2,2)-R(3,3))*2;
    e0 = (R(2,3)-R(3,2))/s; %get e0
    e1 = 0.25 * s; %get e1
    e2 = (R(1,2)+R(2,1))/s; %get e2
    e3 = (R(1,3)+R(3,1))/s; %get e3
elseif R(2,2) > max(R(1,1), R(3,3))
    %The trace is negative...Avoid imaginary numbers
    s = sqrt(1-R(1,1)+R(2,2)-R(3,3))*2;
    e0 = (R(3,1)-R(1,3))/s; %get e0
    e1 = (R(1,2)+R(2,1))/s; %get e1
    e2 = 0.25 * s; %get e2
    e3 = (R(2,3)+R(3,2))/s; %get e3
else
    %The trace is negative...Avoid imaginary numbers
    s = sqrt(1-R(1,1)-R(2,2)+R(3,3))*2;
    e0 = (R(1,2)-R(2,1))/s; %get e0
    e1 = (R(1,3)+R(3,1))/s; %get e1
    e2 = (R(2,3)+R(3,2))/s; %get e2
    e3 = 0.25 * s; %get e3
end
q = [e0;e1;e2;e3];

```

10.7 Update: Quaternion from Gravity and Geomagnetic Vector

end

Chapter 11

Quaternion Orientation Estimation

Contents

11.1 The Quaternion Estimation Algorithm	337
11.1.1 Testing the Quaternion Estimation Algorithm	341
11.2 Background and Derivation of the Kok Schön Algorithm	342
11.3 Estimating Position and Orientation	344

This chapter presents an algorithm for estimating the quaternion orientation of a plane during flight. Although there are many orientation algorithms, this one uses a combination of the Madgwick¹ and Kok Schön² algorithms. The Kok Schön algorithm is the computationally simplest orientation algorithm, yet it maintains accuracy. It requires the fewest lines of code, has the least number of tuning parameters, and requires the least amount of computer memory. An advantage of the Madwick algorithm, however, is that it does not require a known geomagnetic inclination angle. The quaternion estimation algorithm of this chapter combines the advantages of each algorithm for an algorithm that is computationally simple and easy to tune.

This chapter's quaternion estimation algorithm is a type of complementary filter. It numerically integrates the gyrometer measurements to predict the quaternion orientation. This integration is susceptible to drift, so the noisy but unbiased magnetometer and accelerometer measurements correct the effect of drift. The algorithm cleverly calculates from magnetometer and accelerometer measurements an angular velocity correction. Applying the correction removes the gyrometer drift errors caused by the angular velocity integration.

11.1 The Quaternion Estimation Algorithm

At timestep t_k , the estimate of the quaternion is $\hat{q}_k = [\hat{e}_0 \quad \hat{e}_1 \quad \hat{e}_2 \quad \hat{e}_3]^T$. For the first iteration $k = 0$ at time t_0 , it requires an initial condition, e.g., $\hat{q}_0 = [1 \quad 0 \quad 0 \quad 0]^T$. A 3-axis accelerometer sensor measures

¹ Madgwick, Harrison, Vaidyanathan, "Estimation of IMU and MARG orientation using a gradient descent algorithm", July 2011, 2011 IEEE International Conference on Rehabilitation Robotics

² Manon Kok and Thomas B. Schön, "A Fast and Robust Algorithm for Orientation Estimation using Inertial Sensors", IEEE Signal Processing Letters, 2019. DOI: 10.1109/LSP.2019.2943995

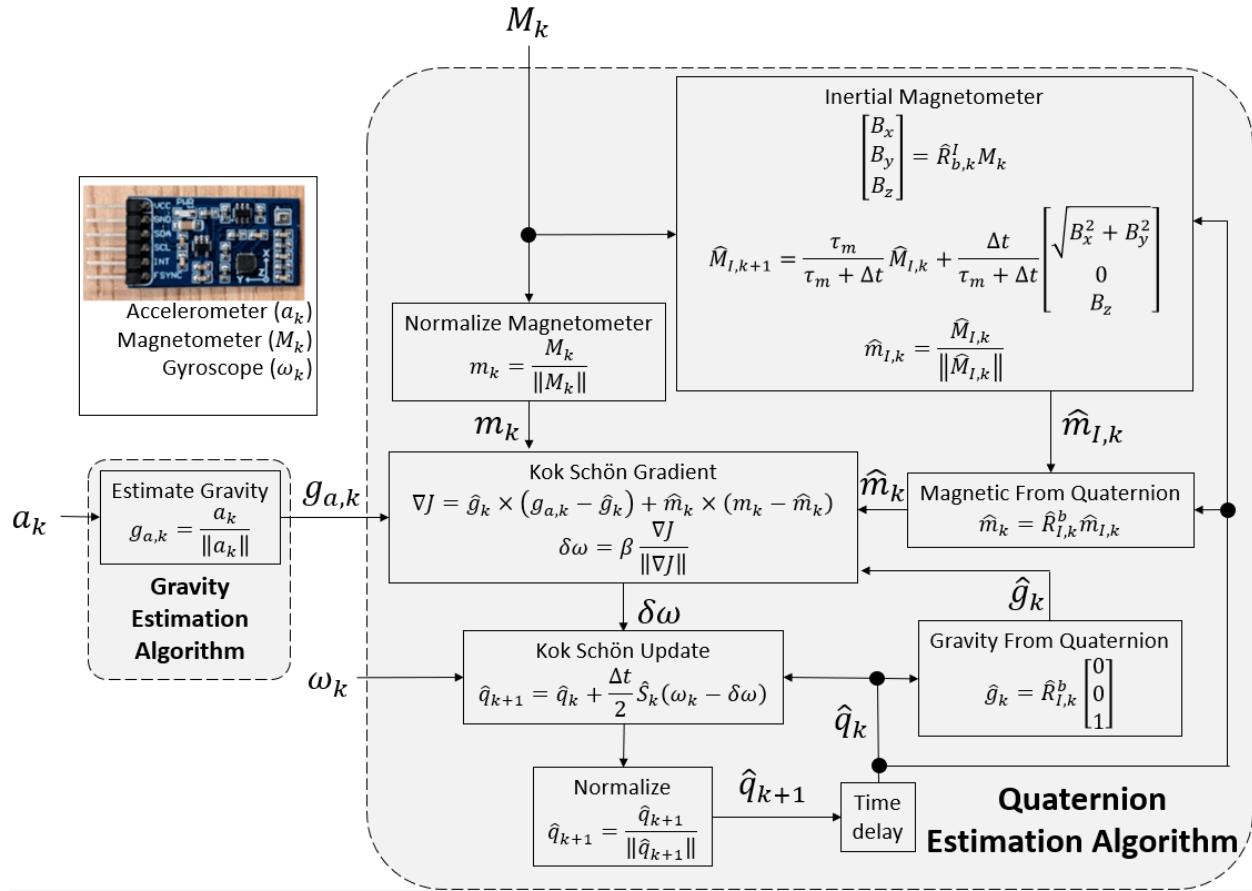


Figure 11.1 Block Diagram of the quaternion estimation algorithm.

gravity and acceleration. The accelerometer signal is a vector $a_k = [a_x \ a_y \ a_z]^T$ (m/s^2). The subscripts x , y , and z correspond to the x , y , and z -axes respectively. The 3-axis gyrometer measurement is $\omega_k = [\omega_x \ \omega_y \ \omega_z]^T$ (rad/s). The 3-axis magnetometer measurement is $M_k = [M_x \ M_y \ M_z]^T$ (μT). Each sensor has been calibrated to eliminate as much bias and noise as possible (see Chapter 8). The algorithm also requires an initial condition for the inertial geomagnetic vector $M_{I,k}$. If the local inclination angle δ is known, the initial condition should be set to $M_{I,k} = [\cos(\delta) \ 0 \ \sin(\delta)]^T$. The normalized magnetometer signal m_k is

$$m_k = \frac{M_k}{\|M_k\|} \quad (11.1)$$

The inertial-to-body rotation matrix is

$$\hat{R}_I^b = \begin{bmatrix} \hat{e}_0^2 + \hat{e}_1^2 - \hat{e}_2^2 - \hat{e}_3^2 & 2(\hat{e}_0\hat{e}_3 + \hat{e}_1\hat{e}_2) & 2(\hat{e}_1\hat{e}_3 - \hat{e}_0\hat{e}_2) \\ 2(\hat{e}_1\hat{e}_2 - \hat{e}_0\hat{e}_3) & \hat{e}_0^2 - \hat{e}_1^2 + \hat{e}_2^2 - \hat{e}_3^2 & 2(\hat{e}_0\hat{e}_1 + \hat{e}_2\hat{e}_3) \\ 2(\hat{e}_0\hat{e}_2 + \hat{e}_1\hat{e}_3) & 2(\hat{e}_2\hat{e}_3 - \hat{e}_0\hat{e}_1) & \hat{e}_0^2 - \hat{e}_1^2 - \hat{e}_2^2 + \hat{e}_3^2 \end{bmatrix} \quad (11.2)$$

11.1 The Quaternion Estimation Algorithm

An estimate of the geomagnetic vector M_I is calculated as follows:

$$\begin{bmatrix} B_x \\ B_y \\ B_z \end{bmatrix} = (\hat{R}_I^b)^T M_k \quad (11.3)$$

However, the geomagnetic vector has no component in the east direction, *i.e.*, B_y should be zero, so a better estimate $\hat{M}_{I,k}$ of the geomagnetic vector M_I is obtained by augmenting B_x and low-pass filtering as follows:

$$\hat{M}_{I,k+1} = \frac{\tau_m}{\tau_m + \Delta t} \hat{M}_{I,k} + \frac{\Delta t}{\tau_m + \Delta t} \begin{bmatrix} \sqrt{B_x^2 + B_y^2} \\ 0 \\ B_z \end{bmatrix} \quad (11.4)$$

The geomagnetic vector should be constant for a specific location on earth, so the low-pass filter time constant τ_m should be fairly large, for example $\tau_m > 5$ seconds. $\hat{M}_{I,k+1}$ is the estimate of the inertial frame geomagnetic vector. Normalizing it results in the magnetometer's estimate $\hat{m}_{I,k}$ of the inertial-frame geomagnetic unit vector $[\cos(\delta) \ 0 \ \sin(\delta)]^T$.

$$\hat{m}_{I,k} = \frac{M_{I,k}}{\|\hat{M}_{I,k}\|} \quad (11.5)$$

The rotation matrix \hat{R}_I^b converts it to the body frame, yielding the algorithm's estimate of body-frame geomagnetic unit vector, \hat{m}_k .

$$\hat{m}_k = \hat{R}_I^b \hat{m}_{I,k} \quad (11.6)$$

The normalized estimate \hat{g}_k of gravity in the body frame is

$$\hat{g}_k = \begin{bmatrix} 2(\hat{e}_1 \hat{e}_3 - \hat{e}_0 \hat{e}_2) \\ 2(\hat{e}_0 \hat{e}_1 + \hat{e}_2 \hat{e}_3) \\ \hat{e}_0^2 - \hat{e}_1^2 - \hat{e}_2^2 + \hat{e}_3^2 \end{bmatrix} \quad (11.7)$$

which is just the last column of the inertial-to-body rotation matrix \hat{R}_I^b .

The gravity unit vector $g_{a,k}$, the geomagnetic vector M_k (or its unit vector m_k), and the angular velocity vector ω_k are sensor measurements fed to the quaternion estimation algorithm. The gravity unit vector $g_{a,k}$ is one of the most difficult signals to measure accurately, because the airplane is often accelerating. Chapter 9.5 presents techniques that may improve the estimate of gravity if GPS signals are available. Eq. (10.23) of Section 10.5 presents another way to estimate the accelerometer's measurement of gravity. The simplest way to estimate the gravity unit vector is to approximate it as the normalized accelerometer signal:

$$g_{a,k} = \frac{a_k}{\|a_k\|} \quad (11.8)$$

where $a_k = [a_x \ a_y \ a_z]^T$ is the accelerometer measurement at the present time step t_k . If a more accurate estimate of gravity is needed, the methods of Chapter 9.5 may be required.

The first step in the algorithm is to calculate a steepest descent gradient ∇J in the direction that minimizes the difference between the estimated and measured magnetometer / accelerometer estimates of gravity and geomagnetic vectors. The steepest descent gradient ∇J is calculated by the Kok Schön gradient ∇J :

$$\nabla J = \hat{g}_k \times (g_{a,k} - \hat{g}_k) + \hat{m}_k \times (m_k - \hat{m}_k) \quad (11.9)$$

where \times is the cross product operator. The second step is to calculate an angular velocity correction $\delta\omega_k$ based on the steepest descent gradient:

$$\delta\omega_k = \beta \frac{\nabla J}{\|\nabla J\|} \quad (11.10)$$

where β is a small, positive, real-valued scalar, e.g., $\beta = 0.3$.

The final step is to apply the correction to the gyrometer's measurement ω_k and numerically integrate to get the next estimate of the quaternion:

$$\hat{q}_{k+1} = \hat{q}_k + \frac{\Delta t}{2} \hat{S}_k (\omega_k - \delta\omega_k) \quad (11.11)$$

where

$$\hat{q}_k = \begin{bmatrix} \hat{e}_0 \\ \hat{e}_1 \\ \hat{e}_2 \\ \hat{e}_3 \end{bmatrix} \quad (11.12)$$

is the estimate of the quaternion at time t_k from the complementary filter,

$$\hat{S}_k = \begin{bmatrix} -\hat{e}_1 & -\hat{e}_2 & -\hat{e}_3 \\ \hat{e}_0 & -\hat{e}_3 & \hat{e}_2 \\ \hat{e}_3 & \hat{e}_0 & -\hat{e}_1 \\ -\hat{e}_2 & \hat{e}_1 & \hat{e}_0 \end{bmatrix} \quad (11.13)$$

and

$$\omega_k = \begin{bmatrix} p_k \\ q_k \\ r_k \end{bmatrix} \quad (11.14)$$

11.1 The Quaternion Estimation Algorithm

is the 3-axis gyrometer measurement of the angular velocity (rad/s) at the present time t_k . The estimated quaternion is normalized at each iteration as follows:

$$\hat{q}_{k+1} = \frac{\hat{q}_{k+1}}{\|\hat{q}_{k+1}\|} \quad (11.15)$$

11.1.1 Testing the Quaternion Estimation Algorithm

You can use the following inputs and outputs to check that your quaternion estimator function is working correctly. If you give the quaternion estimator the following inputs:

$$q_k = \begin{bmatrix} -0.41229239244802518005172942139325 \\ -0.0808817005705984443109102244307 \\ -0.20896118240416539091341974199167 \\ 0.88306758398808005150470989974565 \end{bmatrix}$$

$$\hat{M}_{I,k} = \begin{bmatrix} 20.7 \\ 0 \\ 48.8 \end{bmatrix}$$

$$\omega_k = \begin{bmatrix} -0.29613161100000001679788397268567 \\ -0.00933158199999999724182409011064 \\ -0.1861926469999998891044583615439 \end{bmatrix}$$

$$a_k = \begin{bmatrix} -0.933579445000000077594108915946 \\ -0.490412711999999728783279806521 \\ 6.5710458760000003408663360460196 \end{bmatrix}$$

$$M_k = \begin{bmatrix} -34.409744263000000330521288560703 \\ 2.7896525859999998786520336579997 \\ 40.214271545000002561209839768708 \end{bmatrix}$$

$$\Delta t = 0.06805$$

$$\beta = 0.3$$

$$\tau_m = 5$$

Your quaternion estimator should return the following output:

$$q_{k+1} = \begin{bmatrix} -0.408361 \\ -0.0777655 \\ -0.208554 \\ 0.885268 \end{bmatrix}$$

$$\hat{M}_{I,k+1} = \begin{bmatrix} 20.7713 \\ 0 \\ 48.7648 \end{bmatrix}$$

11.2 Background and Derivation of the Kok Schön Algorithm

Eq. (11.11) is the Kok Schön algorithm. It estimates the quaternion orientation. Without the correction term, *i.e.*, if $\delta\omega_k = 0$, the quaternion estimate would entirely depend on integrating the gyrometer's angular velocity measurement ω_k . It would completely disregard the accelerometer and magnetometer signals $g_{a,k}$ and m_k respectively. This would result in integration drift.

To address the drift problem, the Kok Schön algorithm uses a correction term $\delta\omega_k$. The correction term represents a small angular velocity step in the direction that minimizes the error between the estimated and measured gravity and geomagnetic vectors. To do so, it uses a steepest-descent optimization approach. The function to be minimized is

$$J = \frac{1}{2} \|g_{a,k} - \hat{g}_k\|^2 + \frac{1}{2} \|m_k - \hat{m}_k\|^2 \quad (11.16)$$

which can be written as

$$J = \frac{1}{2} \left\| g_{a,k} - \hat{R}_{I,k}^b \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} \right\|^2 + \frac{1}{2} \left\| m_k - \hat{R}_{I,k}^b \begin{bmatrix} \cos(\delta) \\ 0 \\ \sin(\delta) \end{bmatrix} \right\|^2 \quad (11.17)$$

where $\hat{R}_{I,k}^b$ is the inertial-to-body rotation matrix using the quaternion estimate \hat{q}_k .

The inertial-to-body rotation matrix $\hat{R}_{I,k}^b$ can be written as the product of two rotation matrices:

$$\hat{R}_{I,k}^b = \delta R \hat{R}_{I,k-1}^b \quad (11.18)$$

where $\hat{R}_{I,k-1}^b$ was the rotation matrix at the previous time-step t_{k-1} and δR is the rotation from the orientation at time t_{k-1} to the present orientation at t_k . The Rodrigues form of any rotation matrix δR is

$$\delta R = \begin{bmatrix} e_x^2 + (1 - e_x^2) \cos(\eta) & e_x e_y (1 - \cos(\eta)) + e_z \sin(\eta) & e_x e_z (1 - \cos(\eta)) - e_y \sin(\eta) \\ e_x e_y (1 - \cos(\eta)) - e_z \sin(\eta) & e_y^2 + (1 - e_y^2) \cos(\eta) & e_y e_z (1 - \cos(\eta)) + e_x \sin(\eta) \\ e_x e_z (1 - \cos(\eta)) + e_y \sin(\eta) & e_y e_z (1 - \cos(\eta)) - e_x \sin(\eta) & e_z^2 + (1 - e_z^2) \cos(\eta) \end{bmatrix} \quad (11.19)$$

where $[e_x \ e_y \ e_z]^T$ is the normalized axis of rotation, and η is the rotation angle.

11.2 Background and Derivation of the Kok Schön Algorithm

Assuming a sufficiently high sampling rate such that the rotation from t_{k-1} to t_k is small, i.e., η is small, we can use the small angle approximation. The small angle approximation of the sine function is $\sin(\eta) \approx \eta$ for η in radians. The small angle approximation of the cosine function is $\cos(\eta) \approx 1$. Therefore, the small angle approximation of the Rodrigues rotation matrix is

$$\delta R \approx \begin{bmatrix} 1 & e_z\eta & -e_y\eta \\ -e_z\eta & 1 & e_x\eta \\ e_y\eta & -e_x\eta & 1 \end{bmatrix} \quad (11.20)$$

If the sampling rate is sufficiently high such that the angular velocity is constant from t_{k-1} to t_k , then

$$\begin{bmatrix} e_x \\ e_y \\ e_z \end{bmatrix} = \frac{\omega_k}{\|\omega_k\|} \quad (11.21)$$

where $\omega_k = [p_k \quad q_k \quad r_k]^T$ is the angular velocity. Also, the rotation angle η is

$$\eta = \|\omega_k\| \Delta t \quad (11.22)$$

where $\Delta t = t_k - t_{k-1}$ is the time-step. Therefore, the Rodrigues rotation matrix becomes

$$\delta R \approx \begin{bmatrix} 1 & r_k \Delta t & -q_k \Delta t \\ -r_k \Delta t & 1 & p_k \Delta t \\ q_k \Delta t & -p_k \Delta t & 1 \end{bmatrix} \quad (11.23)$$

and the function to be minimized is

$$J = \frac{1}{2} \left\| g_{a,k} - \begin{bmatrix} 1 & r_k \Delta t & -q_k \Delta t \\ -r_k \Delta t & 1 & p_k \Delta t \\ q_k \Delta t & -p_k \Delta t & 1 \end{bmatrix} \hat{g}_{k-1} \right\|^2 + \frac{1}{2} \left\| m_k - \begin{bmatrix} 1 & r_k \Delta t & -q_k \Delta t \\ -r_k \Delta t & 1 & p_k \Delta t \\ q_k \Delta t & -p_k \Delta t & 1 \end{bmatrix} \hat{m}_{k-1} \right\|^2 \quad (11.24)$$

where

$$\hat{g}_{k-1} = \hat{R}_{I,k-1}^b \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} \quad (11.25)$$

is the previous estimate of the normalized gravity vector and

$$\hat{m}_{k-1} = \hat{R}_{I,k-1}^b \begin{bmatrix} \cos(\delta) \\ 0 \\ \sin(\delta) \end{bmatrix} \quad (11.26)$$

is the previous estimate of the normalized geomagnetic vector.

The angular velocity correction term $\delta\omega_k$ is a vector that points in the steepest descent direction of the function J of Eq. (11.24). The gradient ∇J also points in the direction of steepest descent. It is found by taking partial derivatives of J with respect to ω_k to be

$$\nabla J = \hat{g}_{k-1} \times (g_{a,k} - \hat{g}_{k-1}) + \hat{m}_{k-1} \times (m_k - \hat{m}_{k-1}) \quad (11.27)$$

Note that in Section 11.1, prior estimates \hat{g}_{k-1} and \hat{m}_{k-1} were labeled \hat{g}_k and \hat{m}_k respectively to reduce notation. The correction term $\delta\omega_k$ is the normalized gradient multiplied by a small positive step-size β :

$$\delta\omega_k = \beta \frac{\nabla J}{\|\nabla J\|} \quad (11.28)$$

The Kok Schöön algorithm, Eq. (11.11), is the result of correcting the angular velocity ω_k by the correction term $\delta\omega_k$ and integrating³ to get the quaternion orientation.

11.3 Estimating Position and Orientation

Section 8.7 presented a way to determine the inertial location of the airplane, and Section 11.1 presented a way to estimate its orientation. Figure 11.2 combines both algorithms to estimate the airplane's orientation and its 3D displacement from a known origin. The origin is defined by the latitude ϕ_0 (rad), longitude λ_0 (rad), and altitude h_0 (m).

³see Basile Graf, “Quaternions and Dynamics”, 18 Nov. 2008, Cornell University, Online (accessed March 2021): <https://arxiv.org/pdf/0811.2889.pdf>

11.3 Estimating Position and Orientation

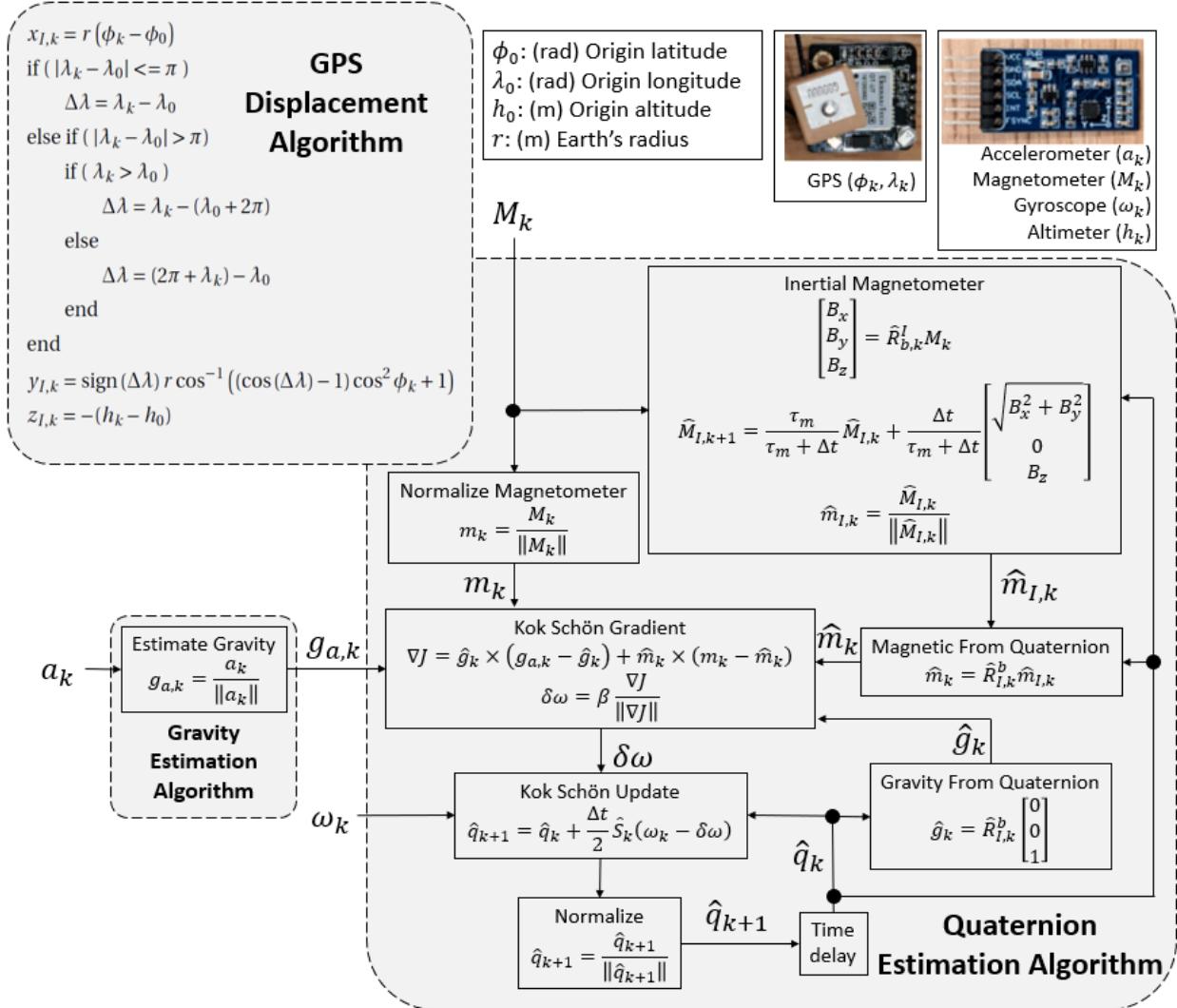


Figure 11.2 GPS displacement algorithm and the quaternion estimation algorithm.

Chapter 12

Machine Learning and System Identification

Contents

12.1	System ID in the Time-Domain	348
12.1.1	Singular Value Decomposition	353
12.2	Nonlinear System Identification	363
12.2.1	Linear System ID with Nonlinear Input Functions	363
12.2.2	System ID with Input Neural Networks	365
12.2.3	System ID with Nonlinear Output Functions	367
12.3	System ID in the Frequency Domain	370
12.3.1	Discrete Fourier Transform (DFT)	371
12.3.2	Fast Fourier Transform (FFT)	376
12.3.3	Bode Plots from Experimental Data	377
12.3.4	Transfer Function Models from Bode Plots	382
12.3.5	Laplace Transfer Functions from Experimental Data	382
12.4	Model Order Reduction and Deduction	382

This chapter presents system identification (ID) and machine learning of dynamic systems. It describes how to use acquired experimental data to derive transfer functions and state-space models of dynamic systems. The focus is on linear system ID, but it also addresses some system nonlinearities. It describes time-domain and frequency domain system ID methods. The time-domain approach uses linear regression to fit parameters of time-series or difference equations. The difference equations can be converted to state-space models. The frequency domain approach constructs Bode plots from experimental data and uses pole-zero mapping to identify transfer functions. This chapter also introduces visual and numerical tools for assessing how well the models fit the experimental data.

12.1 System ID in the Time-Domain

Summary: Time-Domain System ID

Linear dynamic systems can be written as difference equations in the following vector-product form:

$$y_k = \begin{bmatrix} -y_{k-1} & -y_{k-2} & \dots & -y_{k-n} & u_k & u_{k-1} & \dots & u_{k-m} \end{bmatrix} \begin{bmatrix} a_1 \\ a_2 \\ \vdots \\ a_n \\ b_0 \\ b_1 \\ \vdots \\ b_m \end{bmatrix} \quad (12.1)$$

The inputs u_k and outputs y_k are known. The coefficients $a_i, i = 1, \dots, n$ and $b_i, i = 0, 1, \dots, m$ are unknown, and the numbers n and m of these coefficients are also potentially unknown. The goal of machine learning is to determine the coefficients a_i and b_i . If the time-series of the inputs u_k and outputs y_k are collected and stored beginning at time t_1 and ending at t_N , we can write the time-series as

$$\begin{bmatrix} y_{p+1} \\ y_{p+2} \\ \vdots \\ y_k \\ \vdots \\ y_N \end{bmatrix} = \begin{bmatrix} -y_p & -y_{p-1} & \dots & -y_{p-n+1} & u_{p+1} & \dots & u_{p-m+1} \\ -y_{p+1} & -y_p & \dots & -y_{p-n+2} & u_{p+2} & \dots & u_{p-m+2} \\ \vdots & \vdots & & \vdots & & & \vdots \\ -y_{k-1} & -y_{k-2} & \dots & -y_{k-n} & u_k & \dots & u_{k-m} \\ \vdots & \vdots & & \vdots & & & \vdots \\ -y_{N-1} & -y_{N-2} & \dots & -y_{N-n} & u_N & \dots & u_{N-m} \end{bmatrix} \begin{bmatrix} a_1 \\ a_2 \\ \vdots \\ a_n \\ b_0 \\ b_1 \\ \vdots \\ b_m \end{bmatrix} \quad (12.2)$$

or more compactly

$$Y = \Phi\theta \quad (12.3)$$

where $p = \max(n, m)$. Because the coefficients a_i and b_i are in θ , the goal is to find θ . Before we can find θ , however, we need to determine the number n of the a_i coefficients (and hence the sizes of the Y , Φ , and θ matrices). Determining n and m is a combination of engineering judgment and trial and error. The model-order-reduction techniques of Section 12.4 provide a systematic approach using engineering judgment to determine n and m . With the appropriately constructed Φ matrix, we can calculate θ using the pseudo-inverse Φ^\dagger

12.1 System ID in the Time-Domain

$$\theta = \Phi^\dagger Y \quad (12.4)$$

If using MATLAB to solve the pseudo inverse problem, the backslash (\) operator can be used:

$$\theta = \Phi \setminus Y \quad (12.5)$$

Alternatively, the Moore-Penrose pseudo-inverse $\Phi^\dagger = (\Phi^T \Phi)^{-1} \Phi^T$ could be used. Another option for the pseudo-inverse is calculated by inverting the singular value decomposition of Φ , as described in Section 12.1.1.

$$\Phi^\dagger = V S^\dagger U^T \quad (12.6)$$

and

$$S^\dagger = \begin{bmatrix} \frac{1}{s_1} & 0 & \dots & 0 & \dots & 0 \\ 0 & \frac{1}{s_2} & \dots & 0 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots & \dots & 0 \\ 0 & 0 & \dots & \frac{1}{s_p} & \dots & 0 \end{bmatrix} \quad (12.7)$$

where s_1, s_2, \dots, s_p are the singular values, V is the matrix of eigenvectors of $\Phi^T \Phi$, and U is the matrix of eigenvectors of $\Phi \Phi^T$. By solving for θ , we can now extract the a_i and b_i coefficients. By observing which b_i values are zero, or nearly zero relative to the others, we can eliminate those b_i values and determine the correct value for m . Because m has changed, we must again rebuild the Φ matrix, calculate its pseudo-inverse, and solve for θ . Knowing θ , we can extract the a_i and b_i values, and our machine learning process is complete. The result is a difference equation (Eq. (12.1)) that models the dynamic behavior of the system. Section 3.1.6 showed that the controllable canonical form can convert the difference equation to state-space. The coefficients a_1, \dots, a_N and b_0, \dots, b_N from the θ vector are placed into the controllable canonical form matrices:

$$q_{k+1} = \begin{bmatrix} 0 & 1 & 0 & \dots & 0 \\ 0 & 0 & 1 & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \dots & 1 \\ -a_N & -a_{N-1} & -a_{N-2} & \dots & -a_1 \end{bmatrix} q_k + \begin{bmatrix} 0 \\ 0 \\ \vdots \\ 0 \\ 1 \end{bmatrix} u_k \quad (12.8)$$

$$y_k = [(b_N - a_N b_0) \quad \dots \quad (b_2 - a_2 b_0) \quad (b_1 - a_1 b_0)] q_k + b_0 u_k \quad (12.9)$$

The a_i and b_i coefficients from the θ matrix could also be used to get the Z-domain transfer function:

$$\frac{Y}{U} = \frac{b_0 + b_1 z^{-1} + \dots + b_m z^{-m}}{1 + a_1 z^{-1} + \dots + a_n z^{-n}} \quad (12.10)$$

which, using the Tustin approximation $z \approx \frac{1+s\Delta t/2}{1-s\Delta t/2}$, could be converted to a Laplace transfer function. Frequency domain tools, such as Bode or frequency response plots, could be used by substituting the relationship $z = \exp(j\omega\Delta t)$ into Eq. (12.10), where ω is a array of frequencies. Converting the result to state-space or a transfer function allows us to use the tools of Section 3.1.7 to analyze the stability of the learned model. The following MATLAB function is an example implementation of the linear system ID described above:

```

function [Ad,Bd,C,D,yL,fHz,BodeMag,BodePhase] = ...
    function_LinearSystemID(uT,yT,n,m,dt)
%[Ad,Bd,C,D,yL,BodeMag,BodePhase] = function_LinearSystemID(uT,yT,n,m,dt)
%
%OUTPUTS:
%Ad: Controllable canonical state-transition matrix of the learned system
%Bd: Controllable canonical input-transition matrix of the learned system
%C: Controllable canonical output matrix of the learned system
%D: Controllable canonical feedthrough matrix of the learned system
%yL: estimated output signal (to be compared with the true signal yT)
%fHz: (Hz) frequency vector in Hz for Bode magnitude and phase
%BodeMag: (dB) Bode plot magnitude of the learned model
%BodePhase: (deg) Bode plot phase angle of the learned model
%
%INPUTS:
%uT: training data input signal
%yT: training data output signal
%n: number of poles
%m: number of zeros
%dt: (s) timestep

%Get the number N of data points
N = length(uT);

%Determine p
p=max(n,m);

Y = yT(p+1:end); %Y vector in Y=Phi*theta
Phi = zeros(N-p,n+m+1);%Phi vector in Y=Phi*theta
for ii = p+1:N
    %form the Phi matrix
    for jj = 1:n
        Phi(ii-p,jj) = -yT(ii-jj);
    end
    for jj = 0:m
        Phi(ii-p,jj+1+n) = uT(ii-jj);
    end
end
%Solve the equation Y = Phi*theta for theta
theta = Phi \ Y;

%Adjust for the case in which n and m are different
theta_copy = theta;

```

12.1 System ID in the Time-Domain

```
theta = [theta_copy(1:n);...
          zeros(max(0,p-n),1);...
          theta_copy(n+1:end);...
          zeros(max(0,p-m),1)];
b = theta(p+1:end)'; %Numerator coefficients
a = theta(1:p)'; %Denominator coefficients

%Form the controllable canonical discrete state-space matrices
Ad = [zeros(p-1,1),eye(p-1);...
       -fliplr(a)];
Bd = [zeros(p-1,1);1];
C = fliplr(b(2:end)) - b(1)*fliplr(a);
D = b(1);

% Run the training simulation
yL = zeros(N,1); %Allocate memory to store the training results
x = zeros(p,1); %Initial condition for the state
for ii = 1:N
    yL(ii) = C*x+D*uT(ii); %Output equation
    x = Ad*x+Bd*uT(ii); %State-transition equation
end

%get the Z-domain transfer function
num = b;
den = [1, theta(1:p)'];
%Displaying the transfer function requires the Control Systems Toolbox
try tf(num,den,dt,'Variable','z^-1'), catch, end

%Get the Bode plot data
fHz = 1/(dt*N):1/(dt*N):floor(N/2-1)/(dt*N); %(Hz) frequency vector
z = exp(1i*2*pi*fHz*dt);
TF = zeros(size(fHz));
for ii = 1:length(fHz)
    TF(ii) = C*((z(ii)*eye(p)-Ad)^(-1)*Bd)+D;
end
BodeMag = 20*log10(sqrt(real(TF).^2+imag(TF).^2)); %(dB) model mag
BodePhase = atan2d(imag(TF), real(TF)); %(deg) model phase angle
%Force the phase angle to be less than 0
for ii = 1:length(BodePhase)
    if (BodePhase(ii) > 0)
        BodePhase(ii) = BodePhase(ii)-360;
    end
end
end
```

The boxed introduction above summarized time-domain system ID of linear systems. The remainder of this section provides background derivations and examples for the boxed introduction.

Chapter 1 presented continuous-time systems. It showed how linear, continuous-time, state-space systems can be converted to discrete-time systems (see Eqs. (1.22) and (1.23)). Section 3.1.5 demonstrated how to convert a discrete-time state-space system to a difference equation. Therefore, we can convert

both continuous and discrete systems to difference equations in the following form:

$$y_k + a_1 y_{k-1} + \cdots + a_{n-1} y_{k-n+1} + a_n y_{k-n} = b_0 u_k + b_1 u_{k-1} + \cdots + b_m u_{k-m} \quad (12.11)$$

In this chapter, it is more convenient to write Eq. (12.11) in terms of vector products:

$$y_k = \begin{bmatrix} -y_{k-1} & -y_{k-2} & \dots & -y_{k-n} & u_k & u_{k-1} & \dots & u_{k-m} \end{bmatrix} \begin{bmatrix} a_1 \\ a_2 \\ \vdots \\ a_n \\ b_0 \\ b_1 \\ \vdots \\ b_m \end{bmatrix} \quad (12.12)$$

or, more compactly

$$y_k = \phi_k^T \theta \quad (12.13)$$

where

$$\phi_k^T = \begin{bmatrix} -y_{k-1} & -y_{k-2} & \dots & -y_{k-n} & u_k & u_{k-1} & \dots & u_{k-m} \end{bmatrix} \quad (12.14)$$

and

$$\theta^T = [a_1 \ a_2 \ \dots \ a_n \ b_0 \ b_1 \ \dots \ b_m] \quad (12.15)$$

The goal of linear system ID is to use known input u_k and output y_k data to learn the value of θ , i.e., the coefficients a_1, \dots, a_n and b_0, \dots, b_m . Beginning at time t_1 , the series of data y_1, y_2, \dots, y_N and u_1, u_2, \dots, u_N are collected and stored. We can use Eq. (12.12) to write the y_k time-series as

$$\begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_k \\ \vdots \\ y_N \end{bmatrix} = \begin{bmatrix} -y_0 & -y_{-1} & \dots & -y_{1-n} & u_1 & u_0 & \dots & u_{1-m} \\ -y_1 & -y_0 & \dots & -y_{2-n} & u_2 & u_1 & \dots & u_{2-m} \\ \vdots & & & & & & & \\ -y_{k-1} & -y_{k-2} & \dots & -y_{k-n} & u_k & u_{k-1} & \dots & u_{k-m} \\ \vdots & & & & & & & \\ -y_{N-1} & -y_{N-2} & \dots & -y_{N-n} & u_N & u_{N-1} & \dots & u_{N-m} \end{bmatrix} \begin{bmatrix} a_1 \\ a_2 \\ \vdots \\ a_n \\ b_0 \\ b_1 \\ \vdots \\ b_m \end{bmatrix} \quad (12.16)$$

12.1 System ID in the Time-Domain

Unfortunately, since we began collecting and storing data at time t_1 , the values of $y_0, y_{-1}, \dots, y_{1-n}$ and $u_0, u_{-1}, \dots, u_{1-m}$ are unknown. We must discard the rows in Eq. (12.16) which contain this data. Defining $p = \max(n, m)$, Eq. (12.16) becomes

$$\begin{bmatrix} y_{p+1} \\ y_{p+2} \\ \vdots \\ y_k \\ \vdots \\ y_N \end{bmatrix} = \begin{bmatrix} -y_p & -y_{p-1} & \dots & -y_{p-n+1} & u_{p+1} & u_p & \dots & u_{p-m+1} \\ -y_{p+1} & -y_p & \dots & -y_{p-n+2} & u_{p+2} & u_{p+1} & \dots & u_{p-m+2} \\ \vdots & \vdots & & \vdots & & & & \\ -y_{k-1} & -y_{k-2} & \dots & -y_{k-n} & u_k & u_{k-1} & \dots & u_{k-m} \\ \vdots & \vdots & & \vdots & & & & \\ -y_{N-1} & -y_{N-2} & \dots & -y_{N-n} & u_N & u_{N-1} & \dots & u_{N-m} \end{bmatrix} \begin{bmatrix} a_1 \\ a_2 \\ \vdots \\ a_n \\ b_0 \\ b_1 \\ \vdots \\ b_m \end{bmatrix} \quad (12.17)$$

or more compactly

$$Y = \Phi\theta \quad (12.18)$$

Eqs. (12.13) and (12.18) are called the **regressor form**. Φ is the **regressor matrix**, and ϕ_k is the **regressor vector**. The **unknown parameter vector** is θ . We can use batch or recursive estimation methods to find θ . This enables either offline or real-time machine learning of linear dynamic systems.

The size of the matrix Φ is $(N - p) \times (n + m + 1)$. To solve Eq. (12.18), the batch estimation solution requires that $(N - p) \geq (n + m + 1)$. MATLAB's solution to the batch estimation problem of Eq. (12.18) uses the backslash (\) operator:

$$\theta = \Phi \backslash Y \quad (12.19)$$

An alternate solution is to use the Moore-Penrose pseudo-inverse $(\Phi^T \Phi)^{-1} \Phi^T$ to calculate θ by

$$\theta = (\Phi^T \Phi)^{-1} \Phi^T Y \quad (12.20)$$

If the regressor matrix Φ is poorly conditioned, however, MATLAB's backslash operator and the Moore-Penrose pseudo-inverse can face singularity issues. Using singular value decomposition to solve Eq. (12.18) can offer a better approach. The next section presents the Singular Value Decomposition (SVD) solution. It uses examples to show possible reasons why inverting Φ can face singularity and poor conditioning issues. It also presents a way to overcome these issues.

12.1.1 Singular Value Decomposition

The Singular Value Decomposition (SVD) of a real-valued $L \times P$ matrix Φ is given by

$$\Phi = USV^T \quad (12.21)$$

where U is an $L \times L$ unitary matrix, V is a $P \times P$ unitary matrix, and S is an $L \times P$ rectangular diagonal matrix with non-negative real numbers on the diagonal. Φ is a tall matrix, *i.e.*, $L \geq P$. The (T) operator denotes the transpose.

A real **unitary matrix** U has the property that its transpose is also its inverse:

$$U^T U = U U^T = I \quad (12.22)$$

i.e.,

$$U^{-1} = U^T \quad (12.23)$$

Using these properties, we find that the matrix product $\Phi^T \Phi$ is

$$\Phi^T \Phi = V S^T U^T U S V^T = V (S^T S) V^T \quad (12.24)$$

and the matrix product $\Phi \Phi^T$ is

$$\Phi \Phi^T = U S V^T V S^T U^T = U (S S^T) U^T \quad (12.25)$$

We will use eigenvalue decomposition to prove that the columns of U are the eigenvectors of the matrix $S S^T$ and the columns of V are the eigenvectors of the matrix $S^T S$.

Eigenvalue Decomposition and Eigenvectors

By definition, a non-zero column vector v is an eigenvalue of a square matrix A if the following condition is satisfied:

$$A v = \lambda v \quad (12.26)$$

where λ is a complex scalar (called an eigenvalue). If A is $n \times n$ with n linearly independent, normalized, eigenvectors v_1, \dots, v_n corresponding respectively to the n eigenvalues $\lambda_1, \dots, \lambda_n$, then we can write

$$\begin{bmatrix} A v_1 & A v_2 & \dots & A v_n \end{bmatrix} = \begin{bmatrix} v_1 \lambda_1 & v_2 \lambda_2 & \dots & v_n \lambda_n \end{bmatrix} \quad (12.27)$$

or more compactly

$$A Q = Q \Lambda \quad (12.28)$$

where Q is a matrix with v_1, \dots, v_n as the n columns. The matrix Λ is a diagonal matrix whose i^{th} diagonal element is λ_i corresponding to the eigenvector in the i^{th} column of Q . Since the n eigenvectors are linearly independent, we can write

$$A = Q \Lambda Q^{-1} \quad (12.29)$$

Comparing Eq. (12.29) with Eqs. (12.24) and (12.25), and noting that $U^T = U^{-1}$ and $V^T = V^{-1}$, we see that **U and V are eigenvectors of $\Phi \Phi^T$ and $\Phi^T \Phi$ respectively**. Therefore, one way to solve for the U and V matrices is to solve for the eigenvectors of $\Phi \Phi^T$ and $\Phi^T \Phi$ respectively. Some numerical methods perform the singular value decomposition by performing each eigendecomposition instead. The singular values in the S matrix are found by taking the square root of the eigenvalues in the eigendecomposition.

12.1 System ID in the Time-Domain

Machine Learning using Singular Value Decomposition

Machine learning requires signal richness. Signal richness is related to the number and range of frequencies in the input u_k and output y_k signals. For example, if the signals are constant and do not change, the Y and Φ matrices will not contain enough information to determine θ . Φ will be ill-conditioned, and no algorithm will be able to successfully calculate its pseudo-inverse.

Another important challenge in machine learning of linear dynamic systems is knowing the correct numbers n and m of coefficients a_i and b_i to use in the difference equation Eq. (12.17). Selecting appropriate values for n and m uses engineering judgment and trial and error. The model-order-reduction techniques of Section 12.4 provide a systematic approach to determine n and m . To do so, n and m are chosen to be larger than required. Then, fast-response poles outside of a frequency-range-of-interest are eliminated from the identified transfer function to reduce the model order. Reducing the model order also reduces n and m to appropriate values. Choosing n or m too small (too few a_i or b_i coefficients) can result in model inaccuracies because the model order is incorrect. Choosing n or m too large (too many a_i or b_i coefficients) can lead to matrix singularities, over-fitting, and poor conditioning.

Although less effective than the model-order-reduction techniques of Section 12.4, analyzing the singular values of Φ , especially for simulated systems, can help determine the appropriate number n of a_i coefficients. In the singular value decomposition, $\Phi = USV^T$, the singular values are the diagonal elements of S . Choosing n too large will result in some singular values being zero (or nearly zero if noise is present) for perfect systems. Thus we can evaluate the singular values to determine the number n of a_i coefficients.

If the signals u_k and y_k are sufficiently rich, and the correct value of n has been selected, the singular values of Φ will be nonzero. As a result, the (pseudo) inverse of S exists, and we can therefore calculate the pseudo-inverse of Φ . Recall that S is an $L \times P$ rectangular diagonal matrix. The P diagonal elements are the non-negative singular values. We will denote the singular values as s_1, s_2, \dots, s_P where $s_1 \geq s_2 \geq \dots \geq s_P \geq 0$. Since S is a rectangular diagonal matrix, the inverse matrix product $(S^T S)^{-1}$ is

$$(S^T S)^{-1} = \begin{bmatrix} \frac{1}{s_1^2} & 0 & \dots & 0 \\ 0 & \frac{1}{s_2^2} & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & \frac{1}{s_P^2} \end{bmatrix} \quad (12.30)$$

The pseudo-inverse S^\dagger of S is

$$S^\dagger = (S^T S)^{-1} S^T \quad (12.31)$$

$$= \begin{bmatrix} \frac{1}{s_1} & 0 & \dots & 0 & \dots & 0 \\ 0 & \frac{1}{s_2} & \dots & 0 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots & \dots & 0 \\ 0 & 0 & \dots & \frac{1}{s_P} & \dots & 0 \end{bmatrix} \quad (12.32)$$

Therefore, the pseudo-inverse S^\dagger exists if and only if no singular value is zero. Using the pseudo-inverse S^\dagger , the pseudo-inverse of Φ is

$$\Phi^\dagger = VS^\dagger U^T \quad (12.33)$$

and the solution to Eq. (12.18) is

$$\theta = \Phi^\dagger Y \quad (12.34)$$

Just as the relative magnitudes of the singular values help determine the number n of a_i coefficients, the magnitude of the elements of θ can help determine the number m of b_i coefficients. Be careful, however, not to compare the a_i coefficients of θ to the b_i coefficients of θ .

Machine learning can automatically determine the mathematical models that describe the behavior of linear dynamic systems. The physical behavior of linear dynamic systems can be modeled by difference equations. The time-series of inputs and outputs, if sufficiently rich, can help determine the number of coefficients in the difference equations and their values. The procedure for machine learning of linear dynamic systems is illustrated in the following example.

Time-Domain Linear System ID

Example 12.1.1. Linear System Identification in the Time-Domain

Input and output data from a mass-spring-damper system was collected at a time-step of $\Delta t = 0.1$ s. The acquired training and validation data is listed in Table 12.1. The first two columns were training data, and the third and fourth columns were for validation. The first column was the input force (u_T) to the mass-spring-damper system. The second column was the output response (y_T). The third column was the validation input force (u_V) and the fourth column was the validation output (y_V). All data was acquired at a sampling time-step of $\Delta t = 0.1$ s.

Table 12.1 Table of Training and Validation Data

Training Data		Validation Data	
Input (u_T)	Output (y_T)	Input (u_V)	Output (y_V)
0	0	-1.4473	0
2.8472	0	0.9766	-0.0072
-1.4554	0.0142	4.9172	-0.0182
6.6574	0.0382	-3.7789	0.0013
-6.9617	0.0865	-6.0564	0.0313
-6.0622	0.1397	6.2539	0.0082
-4.1145	0.1203	8.2884	-0.0199
1.4554	0.0441	-4.0653	0.0312
5.2020	-0.0491	-3.8741	0.1114
-4.1145	-0.1069	3.3056	0.1472
6.0622	-0.1537	0.8256	0.1761

12.1 System ID in the Time-Domain

2.8472	-0.1946	0.1892	0.2286
6.6574	-0.1845	5.6336	0.2867
-5.2020	-0.1240	-6.9775	0.3735
-6.9617	-0.0498	-7.8652	0.4586
0.0000	-0.0421	7.3539	0.4617
-2.8472	-0.0762	6.0424	0.4541
1.4554	-0.1243	-5.9268	0.5207
-6.6574	-0.1818	-2.9238	0.5933
6.9617	-0.2635	4.2568	0.6149
6.0622	-0.3497	-0.3959	0.6399
4.1145	-0.3631	2.8185	0.6879
-1.4554	-0.3193	3.6379	0.7470
-5.2020	-0.2582	-5.2039	0.8405
4.1145	-0.2320	-5.7774	0.9289
-6.0622	-0.2166	5.7663	0.9561
-2.8472	-0.2067	7.2071	0.9768
-6.6574	-0.2474	-4.4609	1.0676
5.2020	-0.3382		
6.9617	-0.4421		
-0.0000	-0.4792		

Use the training data to form the Y and Φ matrices according to Eqs. (12.17) and (12.18). Use MATLAB's SVD command to calculate the singular value decomposition of Φ . Show that if we incorrectly choose $n = 4$ instead of the correct value of $n = 2$ that two of the singular values will be nearly zero relative to the others. Also show that if we choose $m = 5$ instead of the correct value of $m = 2$ that all but two of the b_i coefficients will be nearly zero. Show the singular values and b_i values on log-scale bar graphs. With the correct values of $n = 2$ and $m = 2$, use the validation data to show that the learned model can successfully predict the behavior of the mass-spring-damper system.

Solution: The following MATLAB code forms the Y and Φ matrices using the incorrect values $n = 4$ and $m = 5$. It also plots the singular values on a log-scale bar graph, which shows that two of the singular values are nearly zero relative to the others. They are around 4 orders of magnitude smaller than the other singular values.

```
% Put the training and validation data into arrays
uT = [0,2.8472,-1.4554,6.6574,-6.9617,-6.0622,-4.1145,1.4554,....
5.2020,-4.1145,6.0622,2.8472,6.6574,-5.2020,-6.9617,0.0000,....
-2.8472,1.4554,-6.6574,6.9617,6.0622,4.1145,-1.4554,-5.2020,....
4.1145,-6.0622,-2.8472,-6.6574,5.2020,6.9617,-0.0000]';
yT = [0,0, 0.0142, 0.0382, 0.0865, 0.1397, 0.1203, 0.0441,....
-0.0491,-0.1069,-0.1537,-0.1946,-0.1845,-0.1240,-0.0498,-0.0421,....
-0.0762,-0.1243,-0.1818,-0.2635,-0.3497,-0.3631,-0.3193,-0.2582,....
-0.2320,-0.2166,-0.2067,-0.2474,-0.3382,-0.4421,-0.4792]';
uV = [-1.4473,0.9766,4.9172,-3.7789,-6.0564,6.2539,8.2884,-4.0653,....
-3.8741, 3.3056, 0.8256, 0.1892, 5.6336,-6.9775,-7.8652, 7.3539,....
6.0424,-5.9268,-2.9238, 4.2568,-0.3959, 2.8185, 3.6379,-5.2039,....
-5.7774,5.7663,7.2071,-4.4609]';
yV = [0,-0.0072,-0.0182, 0.0013, 0.0313, 0.0082,-0.0199, 0.0312,....
0.1114, 0.1472, 0.1761, 0.2286, 0.2867, 0.3735, 0.4586, 0.4617,....
0.4541, 0.5207, 0.5933, 0.6149, 0.6399, 0.6879, 0.7470, 0.8405,....
0.9289,0.9561,0.9768,1.0676]';

%set the timestep and time vector for the training data
dt = 0.1; %(s) sampling time-step
N = length(uT); %Length of the sampled training data
t = 0:dt:dt*(N-1);%(s) time vector for sampled training data

%Now that we have the inputs (uT) and outputs (yT),
% we can use machine learning to estimate the unknown
% a_i and b_i coefficients
n = 4; %Incorrect guess of number of a_i coeffs
m = 5; %Guess of number of b_i coeffs
p = max(n,m);
Y = yT(p+1:end); %Y vector in Y=Phi*theta
Phi = zeros(N-p,n+m+1);%Phi vector in Y=Phi*theta
for ii = p+1:N
    %form the Phi matrix
    for jj = 1:n
        Phi(ii-p,jj) = -yT(ii-jj);
    end
    for jj = n+1:n+m
        Phi(ii-p,jj) = uT(ii-jj);
    end
end
```

12.1 System ID in the Time-Domain

```
    end
    for jj = 0:m
        Phi(ii-p,jj+1+n) = uT(ii-jj);
    end
end

%Calculate the Singular Value Decomposition
[U,S,V] = svd(Phi,0);
figure, bar(diag(S)) %Plot the singular values
set(gca,'YScale','log') %Make it log-scale
title('Singular Values of \Phi')
xlabel('Singular Value')
```

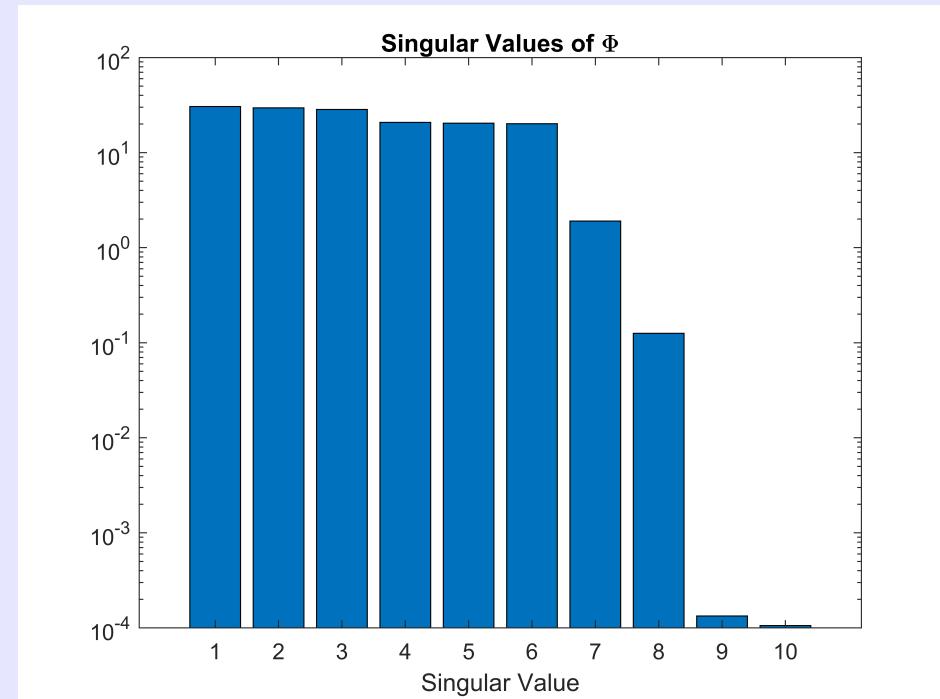


Figure 12.1 Two singular values are nearly zero

Because we have identified that two of the singular values are nearly zero, we can reduce our guess of n by two to $n = 2$. We run the code again with the new value of $n = 2$. The following code also uses the singular-value-decomposition pseudo-inverse to calculate θ and the b_i coefficients. It plots the b_i coefficients on a log-scale bar graph, which shows that all but two b_i coefficients, b_1 and b_2 , are nearly zero relative to the others.

```
uT = [0,2.8472,-1.4554,6.6574,-6.9617,-6.0622,-4.1145,1.4554,....
5.2020,-4.1145,6.0622,2.8472,6.6574,-5.2020,-6.9617,0.0000,....
-2.8472,1.4554,-6.6574,6.9617,6.0622,4.1145,-1.4554,-5.2020,....
4.1145,-6.0622,-2.8472,-6.6574,5.2020,6.9617,-0.0000]';
```

```

yT = [0,0, 0.0142, 0.0382, 0.0865, 0.1397, 0.1203, 0.0441, ...
-0.0491,-0.1069,-0.1537,-0.1946,-0.1845,-0.1240,-0.0498,-0.0421, ...
-0.0762,-0.1243,-0.1818,-0.2635,-0.3497,-0.3631,-0.3193,-0.2582, ...
-0.2320,-0.2166,-0.2067,-0.2474,-0.3382,-0.4421,-0.4792]';
uV = [-1.4473,0.9766,4.9172,-3.7789,-6.0564,6.2539,8.2884,-4.0653, ...
-3.8741, 3.3056, 0.8256, 0.1892, 5.6336,-6.9775,-7.8652, 7.3539, ...
6.0424,-5.9268,-2.9238, 4.2568,-0.3959, 2.8185, 3.6379,-5.2039, ...
-5.7774,5.7663,7.2071,-4.4609]';
yV = [0,-0.0072,-0.0182, 0.0013, 0.0313, 0.0082,-0.0199, 0.0312, ...
0.1114, 0.1472, 0.1761, 0.2286, 0.2867, 0.3735, 0.4586, 0.4617, ...
0.4541, 0.5207, 0.5933, 0.6149, 0.6399, 0.6879, 0.7470, 0.8405, ...
0.9289,0.9561,0.9768,1.0676]';

%set the timestep and time vector for the training data
dt = 0.1; %(s) sampling time-step
N = length(uT); %Length of the sampled training data
t = 0:dt:dt*(N-1);%(s) time vector for sampled training data

%Now that we have the inputs (uT) and outputs (yT),
% we can use machine learning to estimate the unknown
% a_i and b_i coefficients
n = 2; %Incorrect guess of number of a_i coeffs
m = 5; %Guess of number of b_i coeffs
p = max(n,m);
Y = yT(p+1:end); %Y vector in Y=Phi*theta
Phi = zeros(N-p,n+m+1);%Phi vector in Y=Phi*theta
for ii = p+1:N
    %form the Phi matrix
    for jj = 1:n
        Phi(ii-p,jj) = -yT(ii-jj);
    end
    for jj = 0:m
        Phi(ii-p,jj+1+n) = uT(ii-jj);
    end
end
%Calculate the Singular Value Decomposition
[U,S,V] = svd(Phi,0);
%Calculate theta using the Pseudo-inverse
theta = V*S^(-1)*U'* Y;
%Check the b_i coefficients
b = theta(n+1:end);
figure, bar(abs(b)) %Plot the b_i coeffs
set(gca,'YScale','log') %Make it log-scale
%Make the x-axis tick labels the b_i coeffs
tickLabel = cell(1,m);
for ii = 0:m
    tickLabel{ii+1} = ['b_',num2str(ii)];
end
set(gca,'XTickLabel',tickLabel)
title('b_i Values')
xlabel('b_i Coefficient')

```

12.1 System ID in the Time-Domain

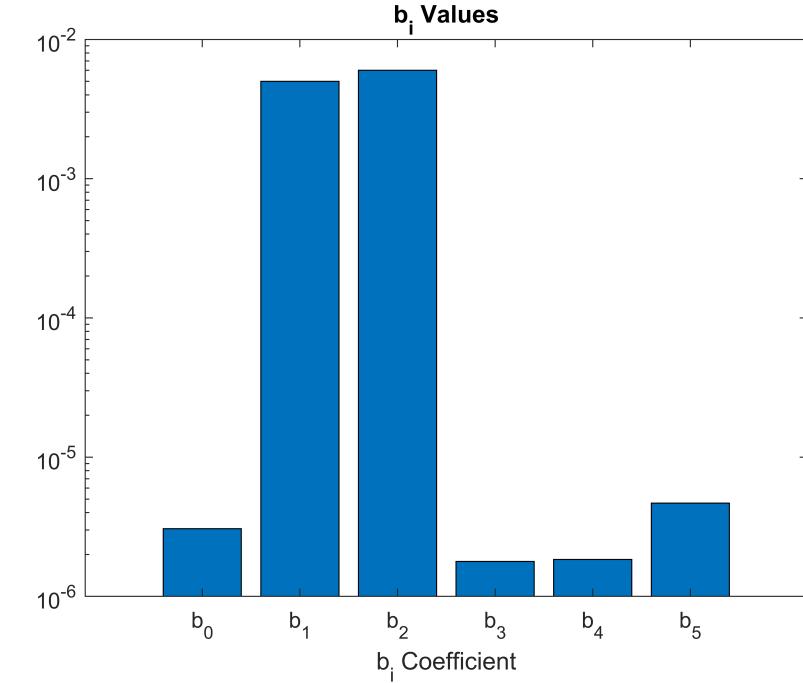


Figure 12.2 All but two b_i coefficients are nearly zero

Now that we have identified that only b_1 and b_2 are nonzero, we can adjust the learned model to have the following form:

$$y_k = -a_1 y_{k-1} - a_2 y_{k-2} + b_1 u_{k-1} + b_2 u_{k-2}$$

The following code uses the updated values of $n = 2$ and $m = 2$. It uses the pseudo-inverse of Φ to calculate θ , i.e., the a_i and b_i coefficients. It then simulates the learned model with the validation input (uV). The learned model ($yEst$) predicts the dynamic response of the true system (yV). The results of both the true (yV) and learned ($yEst$) systems are shown on a graph. The match between the learned model ($yEst$) and the validation data (yV) is close (within 2% absolute error) but not perfect. This suggests that the algorithm may benefit from a larger training data set or that there is noise in the data or the model.

```

uT = [0, 2.8472, -1.4554, 6.6574, -6.9617, -6.0622, -4.1145, 1.4554, ...
5.2020, -4.1145, 6.0622, 2.8472, 6.6574, -5.2020, -6.9617, 0.0000, ...
-2.8472, 1.4554, -6.6574, 6.9617, 6.0622, 4.1145, -1.4554, -5.2020, ...
4.1145, -6.0622, -2.8472, -6.6574, 5.2020, 6.9617, -0.0000]';
yT = [0, 0, 0.0142, 0.0382, 0.0865, 0.1397, 0.1203, 0.0441, ...
-0.0491, -0.1069, -0.1537, -0.1946, -0.1845, -0.1240, -0.0498, -0.0421, ...
-0.0762, -0.1243, -0.1818, -0.2635, -0.3497, -0.3631, -0.3193, -0.2582, ...
-0.2320, -0.2166, -0.2067, -0.2474, -0.3382, -0.4421, -0.4792]';
uV = [-1.4473, 0.9766, 4.9172, -3.7789, -6.0564, 6.2539, 8.2884, -4.0653, ...
-3.8741, 3.3056, 0.8256, 0.1892, 5.6336, -6.9775, -7.8652, 7.3539, ...

```

```

6.0424,-5.9268,-2.9238, 4.2568,-0.3959, 2.8185, 3.6379,-5.2039, ...
-5.7774,5.7663,7.2071,-4.4609]';
yV = [0,-0.0072,-0.0182, 0.0013, 0.0313, 0.0082,-0.0199, 0.0312, ...
0.1114, 0.1472, 0.1761, 0.2286, 0.2867, 0.3735, 0.4586, 0.4617, ...
0.4541, 0.5207, 0.5933, 0.6149, 0.6399, 0.6879, 0.7470, 0.8405, ...
0.9289,0.9561,0.9768,1.0676]';

%set the timestep and time vector for the training data
dt = 0.1; %(s) sampling time-step
N = length(uT); %Length of the sampled training data
t = 0:dt:dt*(N-1);%(s) time vector for sampled training data

%Now that we have the inputs (uT) and outputs (yT),
% we can use machine learning to estimate the unknown
% a_i and b_i coefficients
n = 2; %Incorrect guess of number of a_i coeffs
m = 2; %Guess of number of b_i coeffs
p = max(n,m);
Y = yT(p+1:end); %Y vector in Y=Phi*theta
Phi = zeros(N-p,n+m+1);%Phi vector in Y=Phi*theta
for ii = p+1:N
    %form the Phi matrix
    for jj = 1:n
        Phi(ii-p,jj) = -yT(ii-jj);
    end
    for jj = 0:m
        Phi(ii-p,jj+1+n) = uT(ii-jj);
    end
end

%Calculate the Singular Value Decomposition
[U,S,V] = svd(Phi,0);

%Calculate theta using the Pseudo-inverse
theta = V*S^(-1)*U'* Y;
%Check the b_i coefficients
b = theta(n+1:end);

%Run a simulation using the validation
% input (uV) to calculate a prediction (yEst)
% of the validation output (yV)
NV = length(uV); %Length of the validation data
tV = 0:dt:dt*(NV-1); %(s) validation time vector
%Allocate memory to store the prediction
yEst = zeros(NV,1);
for ii = p+1:NV
    %use the learned difference equation to predict the output
    yEst(ii) = [-yEst(ii-1:-1:ii-n)',uV(ii-1:-1:ii-m)']*...
    theta([1:2,4:end]);
end
figure
plot(tV, yV, tV, yEst,'--','LineWidth',2)
legend('True (yV)', 'Learned (yEst)')

```

12.2 Nonlinear System Identification

```
xlabel('Time (s)')  
ylabel('Displacement (m)')  
title('Displacement of the Mass')  
grid on
```

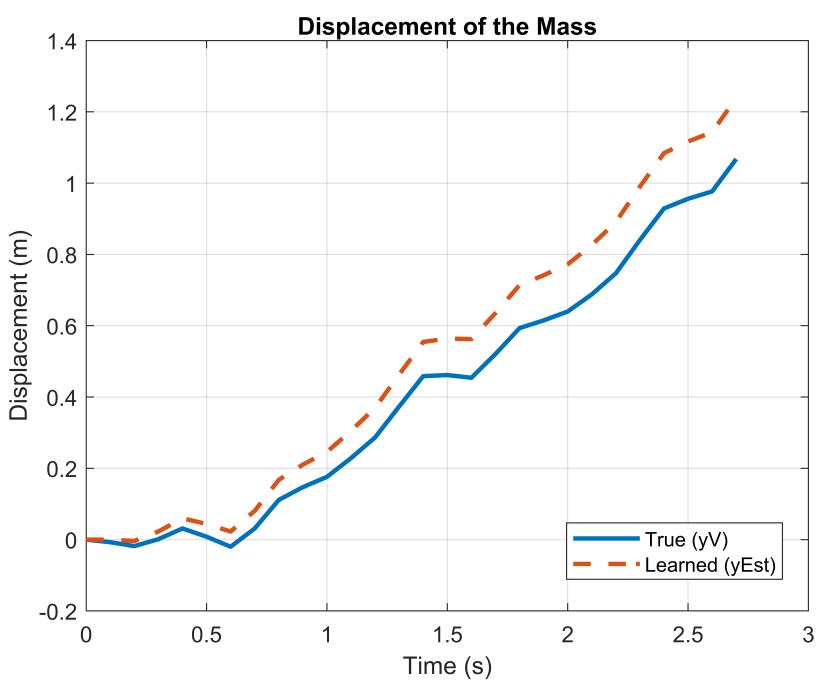


Figure 12.3 The learned system (y_{Est}) predicts the response of the true system (y_V)

12.2 Nonlinear System Identification

Nonlinear machine learning and system ID is a broad topic that could fill an entire textbook or volume of books rather than one section in a chapter. This section will provide only an introduction to a limited scope of topics in nonlinear system ID. Analyzing the stability of nonlinear systems is challenging. Therefore, it is rare for nonlinear machine learning techniques to produce results that can be guaranteed to be stable for all inputs. This chapter will focus on a few exceptions. The first models the dynamic behavior of the system as linear but uses a nonlinear input function. Because the dynamics are linear, the stability tools of Chapter 3 apply to the models generated by this approach.

12.2.1 Linear System ID with Nonlinear Input Functions

Linear system ID with nonlinear input functions transforms Eq. (12.1) by adding a nonlinear input mapping function $f(u)$ to all inputs u_i :

$$y_k = \begin{bmatrix} -y_{k-1} & -y_{k-2} & \dots & -y_{k-n} & f(u_k) & f(u_{k-1}) & \dots & f(u_{k-m}) \end{bmatrix} \begin{bmatrix} a_1 \\ a_2 \\ \vdots \\ a_n \\ b_0 \\ b_1 \\ \vdots \\ b_m \end{bmatrix} \quad (12.35)$$

The input function $f(u)$ could be any static (not dynamic) function. If we define $w_k = f(u_k)$, then Eq. (12.35) becomes

$$y_k = \begin{bmatrix} -y_{k-1} & -y_{k-2} & \dots & -y_{k-n} & w_k & w_{k-1} & \dots & w_{k-m} \end{bmatrix} \begin{bmatrix} a_1 \\ a_2 \\ \vdots \\ a_n \\ b_0 \\ b_1 \\ \vdots \\ b_m \end{bmatrix} \quad (12.36)$$

and the system ID problem for finding the a_i and b_i coefficients is exactly as described in the previous section, Section 12.1. The remaining challenge, however, is to determine the nonlinear (but static) input function $f(u)$. Since an infinite number of trial functions exist, determining $f(u)$ is not trivial. Sometimes engineering judgment can aid the selection of $f(u)$. For example, if it is known that actuator saturation causes input nonlinearities, the following sigmoidal function may be a good choice:

$$w = f(u) \quad (12.37)$$

$$= h \left(-0.5 + \frac{1}{1 + \exp(-mu)} \right) \quad (12.38)$$

where the constant h determines the saturation limits $\pm \frac{h}{2}$, and m adjusts the slope. A graph of a sigmoidal function with $h = 2$ and $m = 9$ is shown in Figure 12.4.

Other potential trial functions could consist of polynomials, exponential functions, or piece-wise linear mappings. If non-integer power expressions are used, care must be taken to avoid imaginary numbers. For example, to avoid imaginary numbers, it would be better to use the function $w = \text{sign}(u) (|u|)^{\frac{3}{7}}$ rather than $w = u^{\frac{3}{7}}$.

12.2 Nonlinear System Identification

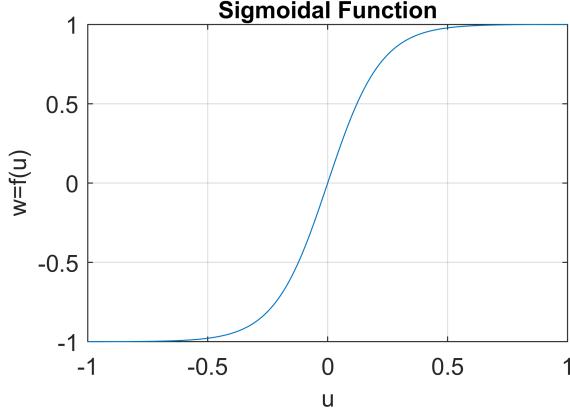


Figure 12.4 The sigmoidal function of Eq. (12.38) with $h = 2$ and $m = 9$

12.2.2 System ID with Input Neural Networks

When there is no prior information to aid engineering judgment, one option is to make the input functions $f(u)$ a linear combination of basis functions: $f(u) = \sum w_i \rho_i(u)$. The functions $\rho_i(u)$ are the basis functions for a radial basis network, spline function, set of orthonormal polynomials, sigmoidal neural network, or other neural network. A machine learning algorithm calculates the optimal weights w_i to fit the training data. Eq. (12.1) is changed as follows:

$$y_k = - \sum_{i_a=1}^n a_{i_a} y_{k-i_a} + \sum_{i=1}^{n_c} w_{0,i} \rho_i(u_k) + \sum_{j=1}^{n_c} w_{1,j} \rho_j(u_{k-1}) + \cdots + \sum_{l=1}^{n_c} w_{m,l} \rho_l(u_{k-m}) \quad (12.39)$$

which can be written in regressor form $y_k = \phi_k^T \theta$, where

$$\phi_k^T = \begin{bmatrix} -y_{k-1} & \cdots & \rho_1(u_k) & \cdots & \rho_{n_c}(u_k) & \rho_1(u_{k-1}) & \cdots & \rho_{n_c}(u_{k-m}) \end{bmatrix} \quad (12.40)$$

and

$$\theta^T = \begin{bmatrix} -a_1 & \cdots & w_{0,1} & \cdots & w_{0,n_c} & w_{1,1} & \cdots & w_{m,n_c} \end{bmatrix} \quad (12.41)$$

The basis functions ρ_i are deterministic, and are therefore completely known. For example, if ρ_i are normalized radial basis functions, they can be written as follows:

$$\rho_i(u) = \frac{\exp(-\beta(u - c_i))}{\sum_{j=1}^{n_c} \exp(-\beta(u - c_j))} \quad (12.42)$$

where $\beta > 0$ is a scalar tuning parameter, and c_i are the radial basis function centers. For example, if it is known that the input u is in the range $u \in [u_{\min}, u_{\max}]$, and there are n_c basis functions, then the n_c centers could be chosen to be uniformly distributed across the range of inputs, e.g., $c_i = u_{\min} + \frac{i-1}{n_c-1} (u_{\max} - u_{\min})$. Since the input u is known at any given timestep, the basis function $\rho_i(u)$ is known. Since every basis function ρ_i is known, and only the weights w_i are unknown, the same process described in Section 12.1 can be used to calculate the unknown parameters of the vector θ , i.e., $\theta = \Phi \setminus Y$. Once θ is known, the

nonlinear input functions are known. Graphical tools can plot the input functions, and sometimes simpler input functions can be fit to the data for more computationally feasible solutions. Such graphical tools are demonstrated in the next section for fitting output nonlinearities. The following MATLAB function is one example of how to calculate the unknown parameter in the θ vector:

Radial Basis Network for Nonlinear Input Functions

Example 12.2.1. System ID with Input Radial Basis Networks

Write a program to implement system ID with an input radial basis network. The function should calculate the unknown parameter vector θ from Eq. (12.41).

Solution: The following MATLAB code provides a function to calculate the unknown parameter vector θ .

```

function theta = function_RadialBasisSystemID(uT,yT,n,m,c,beta)
% theta = function_RadialBasisSystemID(uT,yT,n,m,c,beta)
%
%OUTPUTS:
%theta: vector of regressor coefficients and basis weights
%
%INPUTS:
%uT: Array of input training data
%yT: Array of output training data
%n: Number of past outputs in the time-series
%m: Number of past inputs in the time-series
%c: vector of radial basis function centers, e.g., c_i =
%   u_min+(i-1)/(nc-1)*(u_max-u_min)
%beta > 0: scalar tuning parameter for the radial basis functions

N = length(uT); %Length of the training input data
nc = length(c); %Number of radial basis functions and weights
p = max(n,m);
Y = yT(p+1:end); %Y vector in Y = Phi*theta
Phi = zeros(N-p,n); %Initialize part of the Phi matrix for Y = Phi*theta
%create the Phi matrix
for ii = p+1:N
    %Form the Phi matrix
    for jj = 1:n
        Phi(ii-p,jj) = -yT(ii-jj);
    end
end
for ii = 0:m
    %Get the radial basis functions
    rho = function_RadialBasisNet1D(uT(p-ii+1:N-ii), c, beta);
    %Put the radial basis functions into the Phi matrix
    Phi = [Phi,rho];
end
%Calculate the unknown parameters in the theta vector
theta = Phi \ Y;
end

%Function to calculate the radial basis functions

```

12.2 Nonlinear System Identification

```

function rho = function_RadialBasisNet1D(u, c, beta)
Nc = length(c); %Number of centers = number of basis functions
N = length(u); %Number of inputs
rho = zeros(N,Nc); %Initialize the radial basis functions
for jj = 1:N
    den = 0; %Denominator to normalize the basis functions
    for ii = 1:Nc
        %Calculate the un-normalized basis function
        rho(jj,ii) = exp(-beta * (u(jj)-c(ii))^2);
        den = den + rho(jj,ii); %Calculate the denominator
    end

    %Normalize the basis functions
    rho(jj,:) = rho(jj,:) / den;
end
end

```

12.2.3 System ID with Nonlinear Output Functions

In system ID with nonlinear output functions, the output y_k is treated as a nonlinear mapping $g(z_k)$ of a linear output z_k , *i.e.*, $y_k = g(z_k)$. The inverse mapping $z_k = g^{-1}(y_k)$ is also required so that the regressor equation Eq. (12.1) is replaced with

$$z_k = \begin{bmatrix} -z_{k-1} & -z_{k-2} & \dots & -z_{k-n} & u_k & u_{k-1} & \dots & u_{k-m} \end{bmatrix} \begin{bmatrix} a_1 \\ a_2 \\ \vdots \\ a_n \\ b_0 \\ b_1 \\ \vdots \\ b_m \end{bmatrix} \quad (12.43)$$

and the system ID problem for finding the a_i and b_i coefficients is exactly as described in Section 12.1. The remaining challenge, however, is to determine the nonlinear (but static) output mapping function $y = g(z)$ and its inverse $z = g^{-1}(y)$. Fortunately, graphical tools can assist in generating the nonlinear output mapping function $y = g(z)$. This is described in the following example.

Graphical Aids for Finding the Output Mapping Function

Example 12.2.2. Use graphs to help construct the output mapping function

Roll data was collected from a small, flying wing airplane. The aileron command was the input signal u . The output signal was the gyrometer's measurement of the roll rate ω_x in units of

rad/s. The following validation graph was the result of applying the linear system ID method of Section 12.1 (without any input mapping $w = f(u)$ or output mapping $y = g(z)$).

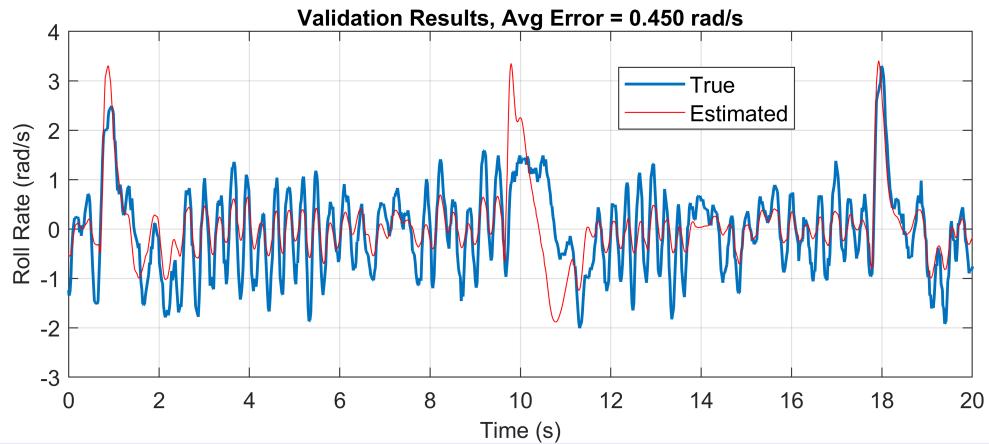


Figure 12.5 Linear system ID results without output mapping

Use graphical aids to help design an output mapping function that can improve the performance of the system ID.

Solution: To design an output mapping function, it helps to create a graph of the true versus the estimated output. The x-axis is the estimated output and the y-axis is the true output.

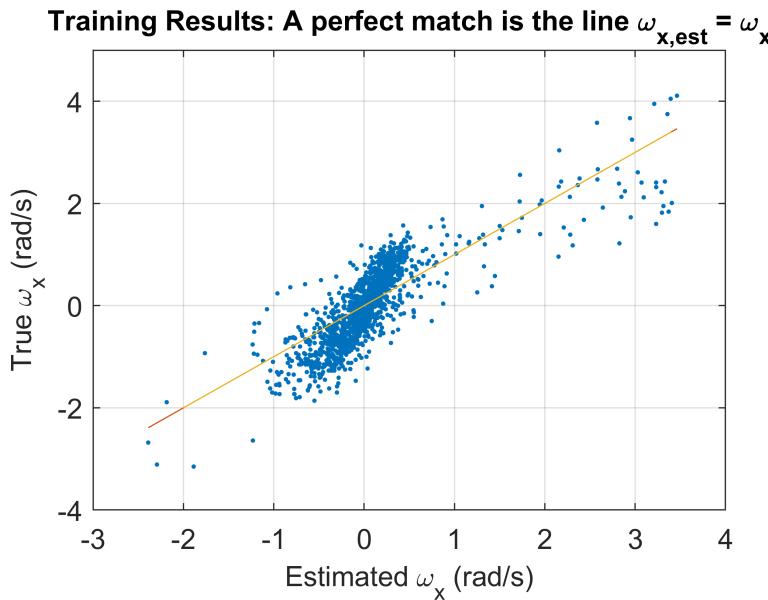


Figure 12.6 Linear system ID results without output mapping. The estimated output is the x-axis; the true output is the y-axis.

If the estimated output is perfect, the result will be a straight line with a slope of one. Inspecting Figure 12.6 shows that a portion of the graph, from -0.4 to 0.4 rad/s, has a significant amount of

12.2 Nonlinear System Identification

data with a different slope. In fact, it appears that a better function to fit the data would be the following piecewise linear function: $y = 0.54x - 0.884$ for $x < -0.4$, $y = 2.75x$ for $-0.4 \leq x \leq 0.4$, and $y = 0.54x + 0.884$ for $y > 0.4$. This piecewise linear function will be used as the nonlinear mapping function $y = g(x)$ and is shown in the following graph:

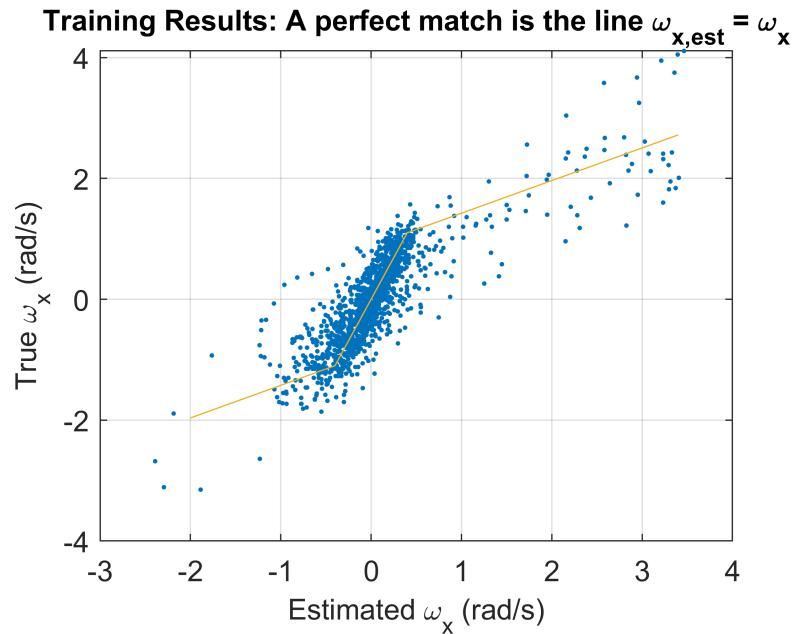


Figure 12.7 A piecewise function may fit the data better than a straight line.

The inverse $x = g^{-1}(y)$ is $x = \frac{y+0.884}{0.54}$ for $y < -1.1$, $x = \frac{y}{2.75}$ for $-1.1 \leq y \leq 1.1$, and $x = \frac{y-0.884}{0.54}$ for $y > 1.1$. These functions can be implemented in MATLAB. $y = g(x)$ is

```
function y = g(x)
if x < 0.4
    y = 0.54*x-0.884;
elseif x <= 0.4
    y = 2.75*x;
else
    y = 0.54*x+0.884;
end
```

and $x = g^{-1}(y)$ is

```
function x = gInv(y)
if y < -1.1
    x = (y+0.884)/0.54;
elseif y <= 1.1
    x = y/2.75;
else
```

```

x = (y - 0.884) / 0.54;
end
    
```

With the inverse mapping function, we can use Eq. (12.43) to replace Eq. (12.1) and repeat the process of Section 12.1. The validation results are shown in the graph below. The average error decreased from 0.45 rad/s without the output map to 0.398 rad/s with it.

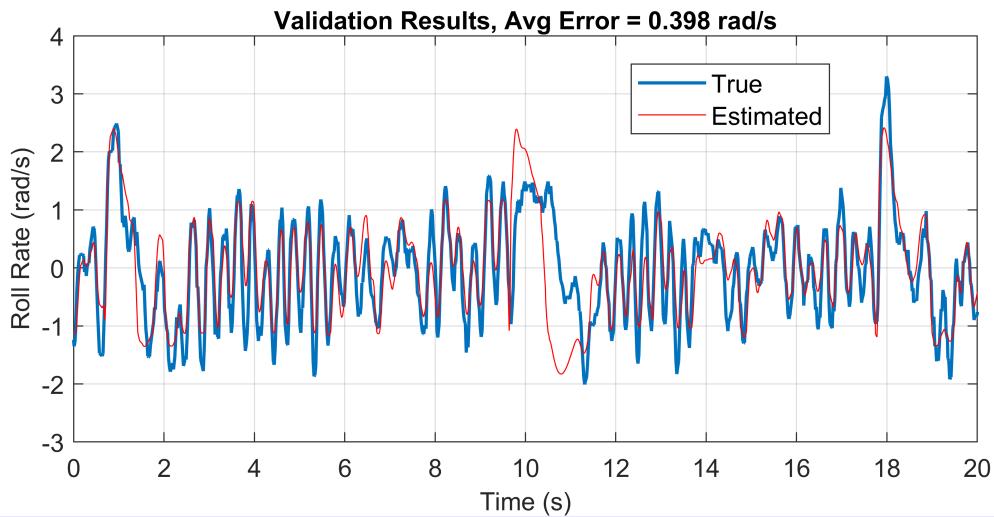


Figure 12.8 Linear system ID results with output mapping

As a side note, it was expected that the nonlinearity was caused by input saturation. When the input map of Figure 12.4 (see Eq. (12.38)) was used instead of the output map of this example, the average error decreased even further to 0.386 rad/s.

12.3 System ID in the Frequency Domain

Bode plots and frequency response plots can be constructed using recorded time-domain input and output data. There are multiple ways of constructing Bode plots from data. One algorithm has already been introduced in the “Summary: Time-Domain System ID” box of Section 12.1. Can you identify it? The function at the end of the summary calculates fHz, BodeMag, and BodePhase arrays for the magnitude and phase of a Bode plot. That function first calculated the Z-domain transfer function, then used $z = \exp(j\omega\Delta t)$ to calculate frequency domain information for an array of frequencies ω . It calculated the transfer function model first, then created frequency domain graphs. The methods of this section do the opposite. They calculate the frequency domain graphs first and then extract the transfer functions and time-domain models.

12.3 System ID in the Frequency Domain

12.3.1 Discrete Fourier Transform (DFT)

The discrete Fourier transform calculates the frequency content of a signal. It works by comparing the measured time-domain signal against sinusoids of different frequencies. The comparison is performed one frequency at a time, over a range of frequencies. When the frequency content of the measured signal and comparison signal match, the summation of their element-wise product is large. When the frequency contents do not match, the summation of their product is small. Most textbooks define the Fourier transform mathematically as follows:

$$\begin{aligned} x_n &= \sum_{k=0}^{N-1} x_k e^{-j\omega_n t_k} \\ &= \sum_{k=0}^{N-1} x_k e^{-j(2\pi f_n)(\Delta t k)} \\ &= \sum_{k=0}^{N-1} x_k e^{-j(2\pi \frac{n}{\Delta t N})(\Delta t k)} \\ &= \sum_{k=0}^{N-1} x_k e^{-j2\pi \frac{n}{N} k} \end{aligned} \quad n = 0, \dots, N-1$$

Euler's formula states that $e^{-j\omega k} = \cos(\omega k) - j \sin(\omega k)$. Using it, the following definition of the DFT separates the real and imaginary parts:

$$x_n = \left(\sum_{k=0}^{N-1} x_k \cos\left(2\pi \frac{n}{N} k\right) \right) - j \left(\sum_{k=0}^{N-1} x_k \sin\left(2\pi \frac{n}{N} k\right) \right) \quad n = 0, \dots, N-1$$

The input signal x_k sampled at discrete times $t_k = k\Delta t$ where Δt is the constant time interval between samples of data, $k = 0, \dots, N-1$, and N is the total number of samples in the batch of samples being analyzed (we will assume N is an even number to simplify notation). The output (x_n) is the n^{th} term in the Fourier transform of the measured signal x_k . It contains frequency content information related to the frequency f_n where f_n is:

$$f_n = \begin{cases} \frac{n}{\Delta t N} & n = 0, \dots, \frac{N}{2}-1 \\ \frac{N-n-1}{\Delta t N} & n = \frac{N}{2}, \dots, N-1 \end{cases}$$

The frequencies occur at discrete intervals and the step size between frequencies is

$$\Delta f = \frac{1}{\Delta t N}$$

If the data were collected at consistent time intervals of Δt then the corresponding sample rate would be:

$$f_s = \frac{1}{\Delta t}$$

and the smallest change in frequency that could be detected, or the resolution of the DFT, is

$$\Delta f = \frac{f_s}{N}$$

Only the first $\frac{N}{2}$ terms of the DFT (terms 0 through $\frac{N}{2} - 1$) provide new information. The last $\frac{N}{2}$ terms repeat the information from the first $\frac{N}{2}$ terms, but in reverse order, i.e., $x_0 = x_{N-1}$, and $x_1 = x_{N-2}$, and $x_2 = x_{N-3}, \dots, x_{N/2} = x_{N/2+1}$. The maximum frequency that can be detected by the DFT is

$$f_{\frac{N}{2}-1} = \frac{\frac{N}{2}-1}{\Delta t N} = \frac{N-2}{2\Delta t N}$$

which is about half of the sampling frequency

$$f_{\frac{N}{2}-1} \approx \frac{1}{2\Delta t} \approx \frac{1}{2} f_s$$

This frequency is called the Nyquist frequency, or

$$f_{\text{Nyq}} = \frac{1}{2} f_s$$

Thus we can only detect frequencies that are less than, but not equal to the Nyquist frequency.

The DFT x_n contains information about the signal at the frequency f_n . For example, the amplitude $|x_n|$ of the signal x_k at f_n can be calculated from x_n as

$$|x_n| = \frac{2}{N} \sqrt{\left(\sum_{k=0}^{N-1} x_k \cos\left(2\pi \frac{n}{N} k\right) \right)^2 + \left(\sum_{k=0}^{N-1} x_k \sin\left(2\pi \frac{n}{N} k\right) \right)^2} \quad (12.44)$$

or more compactly

$$|x_n| = \frac{2}{N} \sqrt{(\text{Re}(x_n))^2 + (\text{Im}(x_n))^2}$$

where $\text{Re}(x_n) = \sum_{k=0}^{N-1} x_k \cos\left(2\pi \frac{n}{N} k\right)$ is the real part of x_n , and $\text{Im}(x_n) = \sum_{k=0}^{N-1} x_k \sin\left(2\pi \frac{n}{N} k\right)$ is the imaginary part of x_n . The Bode magnitude is

$$\text{Mag} = 20 \log_{10}(|x_n|) \quad (12.45)$$

The phase shift $\angle x_n$ of the signal x_k at f_n can be calculated from x_n by the following equation:

$$\angle x_n = \text{atan2}(\text{Im}(x_n), \text{Re}(x_n)) \quad (12.46)$$

Discrete Fourier Transform

Example 12.3.1. Discrete Fourier Transform

Use the discrete Fourier transform to find frequency data from time-domain data. Given 3000 discrete data points from the signal

$$x(t) = 2 \cos(2\pi 2t) + 4 \cos\left(2\pi 4t + \frac{\pi}{2}\right)$$

collected at a sampling rate of $f_s = 100$ Hz, plot the DFT amplitude and phase shift as a function of frequency.

Solution: Collecting measurements at a sampling rate of $f_s = 100$ Hz means we will take a measurement every $\Delta t = \frac{1}{f_s} = 0.01$ s, or at the discrete times $t_k = 0, 0.01, 0.02, \dots, 29.99$ s. The measured

12.3 System ID in the Frequency Domain

signal at each time interval is

$$x_k = 2 \cos(2\pi 2t_k) + 4 \cos\left(2\pi 4t_k + \frac{\pi}{2}\right) \quad k = 0, 1, 2, 3, \dots, 2999$$

and since $t_k = k\Delta t = 0.01k$

$$\begin{aligned} x_k &= 2 \cos(2\pi 2(0.01k)) + 4 \cos\left(2\pi 4(0.01k) + \frac{\pi}{2}\right) \quad k = 0, 1, 2, 3, \dots, 2999 \\ &= 2 \cos(0.04\pi k) + 4 \cos\left(0.08\pi k + \frac{\pi}{2}\right) \quad k = 0, 1, 2, 3, \dots, 2999 \end{aligned}$$

Notice that this signal has two frequency components. It has a 2 Hz signal with an amplitude of 2 and a phase shift of 0 rad, combined with a 4 Hz signal with an amplitude of 4 and a phase shift of $\frac{\pi}{2}$. A two-second sample of the signal is shown in Figure 12.9. The dots on the graph mark the locations at which the continuous signal $x(t)$ is sampled to get the measured signal x_k .

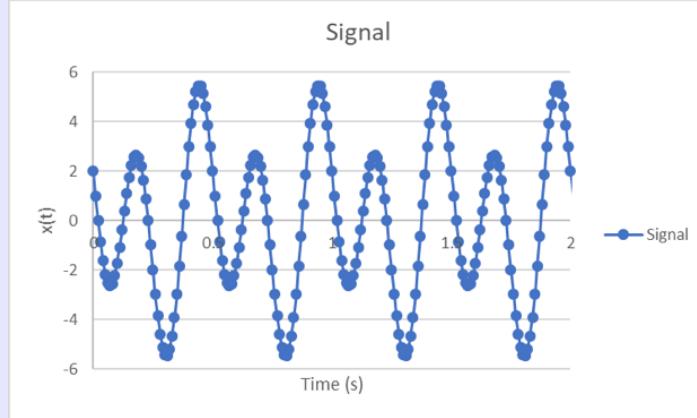


Figure 12.9 Two seconds of the signal $x(t) = 2 \cos(2\pi 2t) + 4 \cos\left(2\pi 4t + \frac{\pi}{2}\right)$ and sampled data points.

Normally we do not know the signal and would need to calculate all the frequencies from 0 to the Nyquist frequency. With $N = 3000$ measurements, a full DFT would calculate the amplitude and phase shift at $n = 0 \dots \frac{N}{2} - 1 = 0 \dots 1499$ different frequencies. In this case, let's just consider f_n at $n = 120$, which is $f_{120} = \frac{n}{\Delta t N} = \frac{120}{(0.01)(3000)} = 4$ Hz, which corresponds to the 4 Hz component of the signal. To find the amplitude we will use Eq. (12.44).

$$\begin{aligned} |x_{120}| &= \frac{2}{N} \sqrt{\left(\sum_{k=0}^{N-1} x_k \cos\left(2\pi \frac{n}{N} k\right) \right)^2 + \left(\sum_{k=0}^{N-1} x_k \sin\left(2\pi \frac{n}{N} k\right) \right)^2} \\ &= \frac{2}{3000} \sqrt{\left(\sum_{k=0}^{2999} x_k \cos\left(2\pi \frac{120}{3000} k\right) \right)^2 + \left(\sum_{k=0}^{2999} x_k \sin\left(2\pi \frac{120}{3000} k\right) \right)^2} \\ &= \frac{2}{3000} \sqrt{\left(\sum_{k=0}^{2999} x_k \cos(0.08\pi k) \right)^2 + \left(\sum_{k=0}^{2999} x_k \sin(0.08\pi k) \right)^2} \end{aligned}$$

and substituting the signal x_k in gives a real and imaginary parts underneath the square root

$$\begin{aligned}\text{Re}(x_{120}) &= \sum_{k=0}^{2999} \left[2 \cos(0.04\pi k) + 4 \cos\left(0.08\pi k + \frac{\pi}{2}\right) \right] \cos(0.08\pi k) \\ \text{Im}(x_{120}) &= \sum_{k=0}^{2999} \left[2 \cos(0.04\pi k) + 4 \cos\left(0.08\pi k + \frac{\pi}{2}\right) \right] \sin(0.08\pi k) \\ |x_{120}| &= \frac{2}{3000} \sqrt{(\text{Re}(x_{120}))^2 + (\text{Im}(x_{120}))^2}\end{aligned}$$

When all the measurements in the signal x_k are multiplied in the real part of the equation by $\cos(0.08\pi k)$ we get the plot shown on the left in Figure 12.10. When the signal x_k is multiplied in the imaginary part of the equation by $\sin(0.08\pi k)$ we get the plot shown on the right in Figure 12.10.

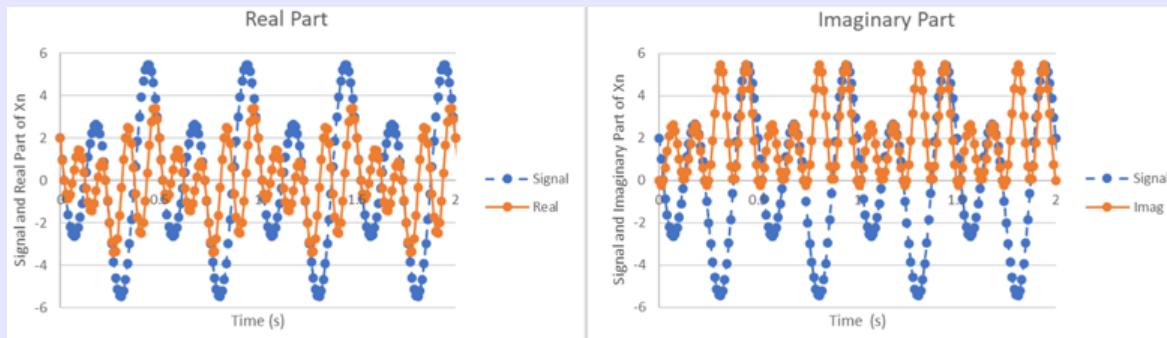


Figure 12.10 Plots of the both the real and imaginary parts of the underneath the square root. The measured signals are the blue dots whereas the multiplication is the orange dots

When we add up all the values of the real part, the sum is very close to zero

$$\text{Re}(x_{120}) = -4.29 \times 10^{-10}$$

This is because the negative values (shown on the left of Figure 12.10) cancel with the positive values.

When we add up all the values of the imaginary part we get a large positive number

$$\text{Im}(x_{120}) = 6000$$

This is because all the values are positive, as shown in the plot in Figure 12.10.

The amplitude of the 4 Hz component of the signal can now be calculated to be

$$\begin{aligned}|x_{120}| &= \frac{2}{3000} \sqrt{(\text{Re}(x_{120}))^2 + (\text{Im}(x_{120}))^2} \\ &= \frac{2}{3000} \sqrt{(-4.29 \times 10^{-10})^2 + 6000^2} \\ &= 4\end{aligned}$$

12.3 System ID in the Frequency Domain

This exactly matches the 4 Hz amplitude of the signal x_k ! We can calculate the phase shift using Eq. (12.46).

$$\angle x_{120} = \text{atan2}(6000, -4.29 \times 10^{-10}) = \frac{\pi}{2}$$

This exactly matches the phase shift of the 4 Hz component of the signal x_k !

What if we apply the Fourier transform corresponding to $n = 60$ or $f_{60} = \frac{n}{\Delta t N} = \frac{60}{(0.01)(3000)} = 2$ Hz?

The amplitude is found to be $|x_{60}| = 2$, which was exactly the amplitude of the 2 Hz component of the signal x_k ! The phase shift is $\angle x_{60} = 0$, which exactly matches the phase shift of the 2 Hz component of the signal x_k !

What happens if we consider any other frequency? For example, consider $n = 90$ which corresponds to a frequency of $f_{90} = \frac{n}{\Delta t N} = \frac{90}{(0.01)(3000)} = 3$ Hz? The real part $\text{Re}(x_n) = -9.93 \times 10^{-13}$, and the imaginary part $\text{Im}(x_n) = 1.20 \times 10^{-14}$. The magnitude is $|x_{90}| = 6.6 \times 10^{-6}$, which is practically zero; the phase shift $\angle x_{90} = \text{atan2}(0, 0)$ is undefined. These results are exactly as expected because the signal x_k does not have a 3 Hz frequency component.

The discrete Fourier transform magnitude and phase of the signal x_k for the entire frequency domain can be seen in Figures 12.11 and 12.12. Only two frequencies have a non-zero amplitude: 2 Hz and 4 Hz. Only the 4 Hz frequency has a phase shift.

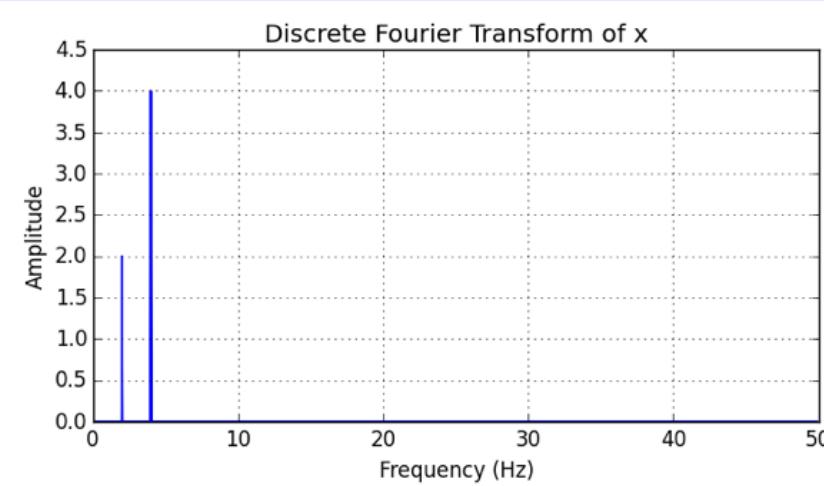


Figure 12.11 DFT amplitude

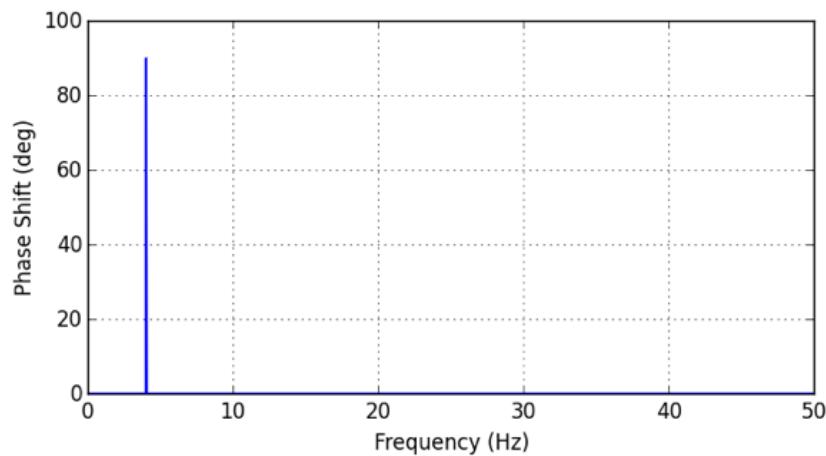


Figure 12.12 DFT phase shift

Note that the plots in Figures 12.11 and 12.12 only go to a maximum frequency of 50Hz. This is, if you recall, the Nyquist frequency because the sampling frequency was $f_s = 100$ Hz so the Nyquist frequency would be $f_{Nyq} = \frac{f_s}{2} = 50$ Hz. Thus, we can only detect frequencies below and not equal to the Nyquist frequency, or half the sampling frequency. To be more exact, the maximum frequency that can be detected is:

$$\begin{aligned}
 f_{\frac{N}{2}-1} &= \frac{\frac{N}{2} - 1}{\Delta t N} \\
 &= \frac{\frac{3000}{2} - 1}{(0.01)(3000)} \\
 &= \frac{1499}{30} \\
 &= 49.6
 \end{aligned}$$

which is less than the Nyquist frequency of $f_{Nyq} = 50$ Hz.

12.3.2 Fast Fourier Transform (FFT)

The Fast Fourier Transform (FFT) is a computationally efficient solution to the discrete Fourier transform. Calculating the discrete Fourier transform can be computationally demanding. For fast sampling rates and limited computational processing speed, calculating the discrete Fourier transform directly may be too slow. A variety of algorithms have been developed to reduce computational requirements. These are called Fast Fourier Transform (FFT) algorithms. The most popular is known as the Cooley-Tukey algorithm. It reduces the computational requirements by dividing the N data points of the measured signal x_k into smaller samples. It calculates the discrete Fourier transform of these smaller samples and combines them to get the transform of the entire signal.

12.3 System ID in the Frequency Domain

12.3.3 Bode Plots from Experimental Data

A useful application of Fourier transforms is the ability to create Bode plots and frequency response plots from experimental data. As discussed in Section 2.8.2, Bode plots can be used to identify parameters of differential equations. The slope of the Bode plot is related to the relative order of the transfer function numerator and denominator. The points on the Bode plot at which two asymptotes intersect can be used to identify cutoff and natural frequencies, time constants, and damping ratios, as discussed in Section 2.8.3.

Fourier transforms can be used to create Bode plots without requiring a mathematical model of the underline governing differential equations. Recall that the transfer function of a system is the ratio of the output $Y(s)$ over the input $U(s)$

$$TF(s) = \frac{Y(s)}{U(s)} \quad (12.47)$$

A Bode plot consists of two graphs: 1) Magnitude $\text{dB} = 20\log_{10}(r_{TF})$, and 2) Phase angle θ_{TF} . The transfer function magnitude r_{TF} and phase angle θ_{TF} were introduced in Section 2.7.5. In that section, the magnitude and phase angle were calculated mathematically. In this chapter, we show that the Fourier transform can find them from experimental data. To find the magnitude r_{TF} of the transfer function at the frequencies f_n , $n = 0, \dots, \frac{N}{2} - 1$, we perform the following element-wise division

$$r_{TF,n} = \frac{|Y_n|}{|U_n|}, \quad n = 0, \dots, \frac{N}{2} - 1 \quad (12.48)$$

where $|Y_n|$ is the amplitude (see Eq. (12.44)) of the Fourier transform of the output $y(t)$ at the n^{th} frequency, and $|U_n|$ is the amplitude (also calculated by Eq. (12.44)) of the Fourier transform of the input $u(t)$ at the n^{th} frequency f_n . The Bode magnitude is calculated from $r_{TF,n}$ by $\text{Mag} = 20\log_{10}(r_{TF})$. Because the input amplitude $|U_n|$ is in the denominator, the frequency content of the input must be **sufficiently rich**, meaning it must contain nonzero information at each analyzed frequency to avoid division by zero. For example, the signal of Example 12.3.1 only has two frequencies with nonzero amplitudes, and would therefore not be sufficiently rich to extract Bode plot data at any other frequencies besides 2 Hz and 4 Hz. Better input signals sweep a wide range of frequencies. Chirp signals, step inputs, and random input signals are often used to provide signal richness.

To get the phase angle $\theta_{TF,n}$ of the transfer function of the n^{th} frequency f_n , we subtract the phase angle $\angle U_n$ (see Eq. (12.46)) of the input from the phase angle $\angle Y_n$ of the output:

$$\theta_{TF,n} = \angle Y_n - \angle U_n, \quad n = 0, \dots, \frac{N}{2} - 1 \quad (12.49)$$

The following example creates a MATLAB function to find the Bode plot magnitude, phase shift, and frequency information from experimental data.

Bode Plot from Experimental Data

Example 12.3.2. Bode plot from experimental data

Create a function in MATLAB that generates a Bode plot from time-domain data.

Solution: The input signal to a dynamic system is the array of data u with a sampling timestep of dt . The output of the dynamic system is the signal y . The following MATLAB function extracts the

frequency domain information from the input and output signals and creates a Bode plot.

```

function [fHz,BodeMag,BodePhaseDeg] = function_BodeFromData(u,y,dt)
% [fHz,BodeMag,BodePhaseDeg] = function_BodeFromData(u,y,dt)
%
%OUTPUTS:
%fHz: (Hz) array of Bode plot frequencies
%BodeMag: (dB) array of Bode plot magnitudes
%BodePhaseDeg: (deg) array of Bode plot phase shift angles
%
%INPUTS:
%u: Time-domain input data
%y: Time-domain output data
%dt: Sampling timestep

N = length(u); %Number of data points
U = fft(u); %Discrete Fourier Transform of the input data
Y = fft(y); %Discrete Fourier Transform of the output data
Z = Y./U; %Frequency domain transfer function
ZMag = sqrt(real(Z).^2+imag(Z).^2); %Transfer function magnitude
ZPhase = atan2d(imag(Z),real(Z)); %(deg) Transfer function phase
Nby2 = floor(N/2); %Get half the length of the data
BodeMag = 20*log10(ZMag(1:Nby2)); %Get the Bode plot magnitude data
BodePhaseDeg = ZPhase(1:Nby2); %Get the Bode plot phase shift angle data
df = 1/(N*dt); %Get the frequency step-size
fHz = 0:df:Nby2-1; %(Hz) get the frequency array

%Create the Bode plot
figure
subplot(211)
semilogx(fHz,BodeMag)
title('Bode Plot')
ylabel('20log_1_0(r_T_F)')
grid on
subplot(212)
semilogx(fHz, BodePhaseDeg)
ylabel('Phase (deg)')
xlabel('Frequency (Hz)')
grid on

end

```

The angle θ is the same as $\theta \pm 2\pi$, or, in degrees, $\theta^\circ = \theta^\circ \pm 360^\circ$. Because of that, it may sometimes be helpful to add or subtract multiples of 360° from some of the phase shift angles to remove jumps in the Bode phase angle graph.

The following example calculates the Bode plot from data for a known transfer function and compares the result with the theoretical Bode plot. A chirp signal is used as the input to ensure signal richness.

12.3 System ID in the Frequency Domain

Experimental and Theoretical Bode Plots

Example 12.3.3. Experimental and theoretical Bode plots

A series resistor capacitor circuit has a resistance of $R = 200 \Omega$ and capacitance of $C = 1 \text{ mF}$. If the output is the voltage drop across the capacitor V_C , the transfer function is

$$\frac{V_C}{V_{in}} = \frac{1}{0.2s + 1}$$

The input voltage V_{in} to the circuit is a chirp waveform:

$$V_{in}(t) = \sin\left(2\pi f_0 \left(\frac{r^{t-T} - 1}{\ln(r)}\right)\right)$$

where the initial frequency is $f_0 = 1 \text{ Hz}$, the exponentially increasing frequency rate is $r = 1.25$, and the sweep time interval is $T = 8$. The time-step is $\Delta t = 0.001 \text{ s}$. Use the Fourier transform to determine the Bode plot of the system. Simulate the circuit for 30 s and record the input and output data. Compare the Bode plot derived from the Fourier transforms of the input and output data with the Bode plot calculated mathematically from the transfer function.

Solution: The input signal is V_{in} , and V_C is the output signal. The input signal is the chirp waveform. To get the input signal experimentally, we would measure the voltage input using a data acquisition device. We would store the result in an array V_{in} . Because the input is the chirp signal, the k^{th} element of the array would be

$$V_{in,k} = \sin\left(2\pi f_0 \left(\frac{r^{k\Delta t - T} - 1}{\ln(r)}\right)\right), \quad k = 0, \dots, N$$

To get the output signal V_C experimentally, we would measure the voltage across the capacitor using a data acquisition device. We would store the result in an array V_C . Because V_C is the voltage drop across the capacitor in a series resistor-capacitor circuit, the k^{th} element of the V_C array would be (assuming ideal components):

$$V_{C,k} = \exp\left(\frac{-\Delta t}{RC}\right) V_{C,k-1} + \left(1 - \exp\left(\frac{-\Delta t}{RC}\right)\right) V_{in,k-1}, \quad k = 1, \dots, N$$

where the initial condition is $V_{C,0} = 0$, if the capacitor was initially discharged. This equation is the discrete-time solution to the ODE governing the behavior of the resistor-capacitor circuit. Experimentally, there would be no need to know this equation. We would only need to measure the voltage drop across the capacitor and store the result in the array V_C . The input and output signals, V_{in} and V_C respectively, are graphed in Figure 12.13.

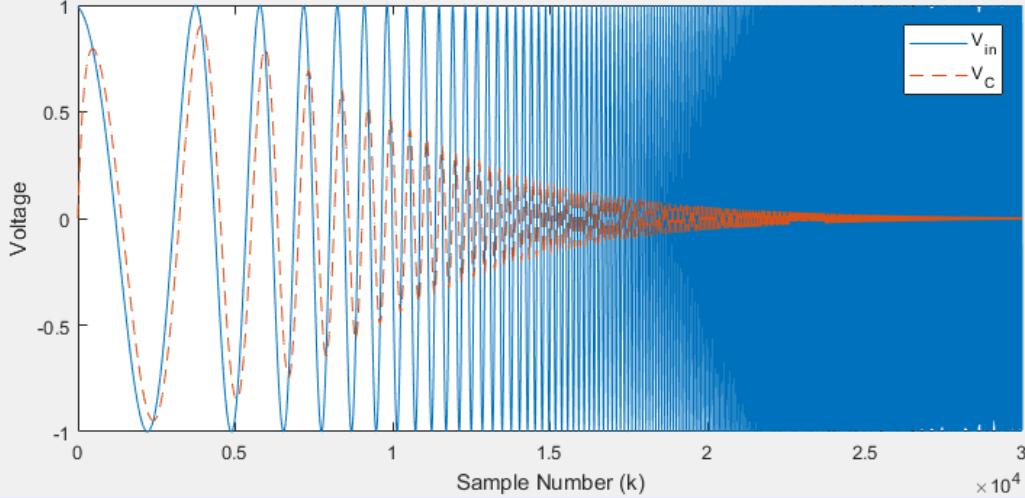
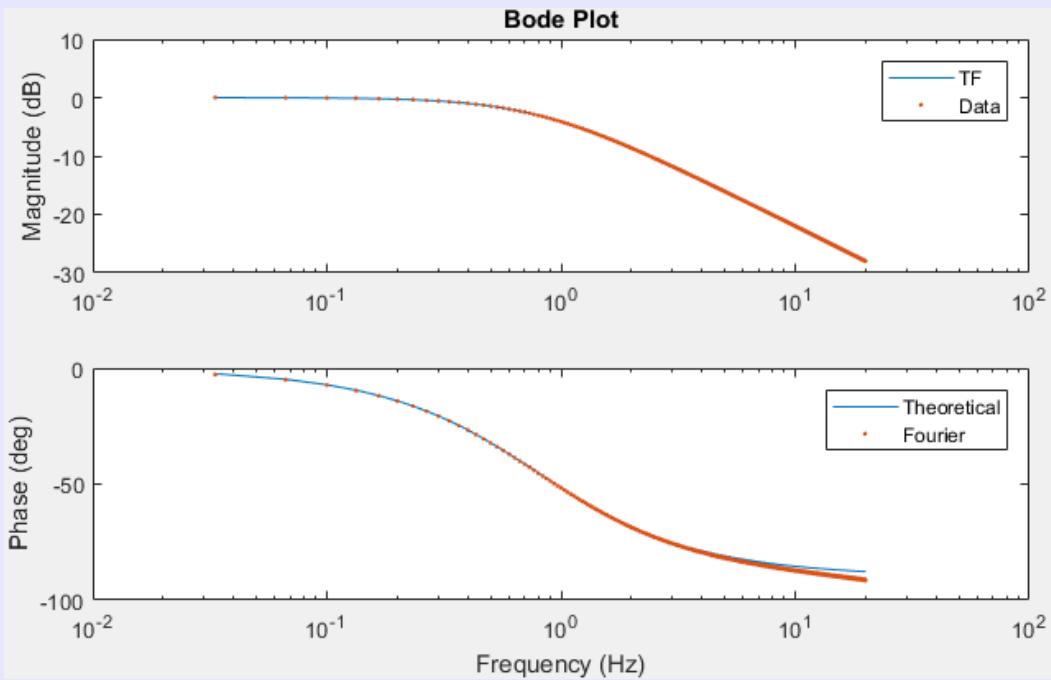


Figure 12.13 Graph of the arrays V_{in} and V_C . Each array has 30000 elements.

We can calculate the magnitude (or amplitude) $|V_{in,n}|$ of the input by replacing the variable x_k in Eq. (12.44) with the input $V_{in,k}$. We can also calculate the magnitude $|V_{C,n}|$ of the output by replacing the variable x_k in Eq. (12.44) with the output $V_{C,k}$.

Similarly, we can calculate the phase angles $\angle V_{in}$ and $\angle V_C$ of V_{in} and V_C by replacing x_k in Eq. (12.46) with V_{in} and V_C respectively. Finally, we can use Eqs. (12.48) and (12.49) to calculate the Bode plot magnitude $r_{TF,N}$ and phase angle $\theta_{TF,n}$ respectively. These results are compared with the theoretical Bode plot in Figure 12.14.



12.3 System ID in the Frequency Domain

Figure 12.14 The Bode plot derived from data is compared with the theoretical Bode plot

The code for this example is provided below:

```
close all
clear
clc
dt = 0.001; %(s) time step
t = 0:dt:(30.001-dt); %(s) time vector
N = length(t); % Number of samples
C = 0.001; %(F) capacitance
R = 200; % (Ohm) resistance
a = exp(-dt/C/R); % Parameter for numerical RC circuit
b = 1-a; % Parameter for numerical RC circuit
f0 = 1; %Initial Chirp Frequency
T = 8; % Chirp time interval
r = 1.25; % Chirp frequency sweep rate
V_in = sin(2*pi*f0*(r.^((t-T)-1)/log(r))); %Input Voltage
V_C = zeros(1,N); % Voltage drop across capacitor
V_C(1) = 0; %initial voltage across capacitor
for ii = 1:N-1
    V_C(ii+1) = a*V_C(ii)+b*V_in(ii); %Simulate the RC circuit
end

%plot of voltage arrays versus sample number
figure
plot(t/dt, V_in, t/dt, V_C,'--')
legend('V_i_n','V_C')
xlabel('Sample Number (k)')
ylabel('Voltage')

% Setup for Fourier transforms
U = fft(V_in); %Discrete Fourier transform of the input
Y = fft(V_C); %Discrete Fourier transform of the output
Z = Y./U; %Frequency domain transfer function from input to output
ZMag = sqrt(real(Z).^2+imag(Z).^2); %Transfer function magnitude
ZPhase = atan2d(imag(Z),real(Z)); %(deg) Transfer function phase
N_freq = 600; %number of frequencies to plot
BodeMag = 20*log10(ZMag(1:N_freq)); %Get the Bode plot magnitude data
BodePhaseDeg = ZPhase(1:N_freq); %Get the Bode plot phase shift angle data
df = 1/(N*dt); %Get the frequency step-size
fHz = 0:df:df*(N_freq-1); %(Hz) get the frequency array

%theoretical Bode magnitude and phase angle
Y_U = 1./sqrt((0.2*2*pi*fHz).^2+1); %Theoretical amplitude
phi_Y_U = atan2d(0,1)-atan2d(0.2*2*pi*fHz,1); %Theoretical phase shift

%plot the theoretical and Fourier Bode plots
figure
subplot(211)
semilogx(fHz, 20*log10(Y_U), fHz, BodeMag, '.')
ylabel('Magnitude (dB)')
title('Bode Plot')
```

```

legend('TF','Data')
subplot(212)
semilogx(fHz, phi_Y_U, fHz, BodePhaseDeg, '.')
legend('Theoretical','Fourier')
ylabel('Phase (deg)')
xlabel('Frequency (Hz)')

```

12.3.4 Transfer Function Models from Bode Plots

Section 2.8.3 and Section 2.8.4 discussed how transfer function poles and zeros affect Bode magnitude and phase plots. These same concepts can be applied to visually identify poles, zeros, and steady-state gain on a Bode plot generated from experimental data. By visually identifying poles, zeros, and steady-state gain, the transfer function model of the dynamic system can be reconstructed.

12.3.5 Laplace Transfer Functions from Experimental Data

One system ID method to determine the Laplace transfer function from experimental data is summarized as follows:

1. Use the methods of Section 12.1 to calculate the discrete state-space matrices A_d , B_d , C_d , and D_d
2. Convert the discrete state-space matrices to continuous state-space A , B , C , and D using one of the methods from Table 3.3
3. Calculate the transfer function from Eq. (2.5): $\frac{Y}{U} = C(sI - A)^{-1}B + D$

12.4 Model Order Reduction and Deduction

System ID algorithms can produce mathematical models that have a higher order than necessary, *i.e.*, they have too many poles and zeros. Section 12.3.5 explained how to derive a Laplace transfer function from experimental data, and Section 2.5 described how to convert a transfer function to its zero-pole-gain form.

The zero-pole-gain form of a transfer function is conducive to model order reduction. It factors the transfer function into individual poles and zeros. The poles and zeros can be mapped onto the complex plane. Poles and zeros that result in sufficiently fast convergence can be eliminated from the transfer function. Slowly converging poles and zeros must be retained. The model order reduction technique of this chapter is based on removing poles and zeros outside of a frequency-range-of-interest, see Figure 12.15.

The portion of the complex plane containing poles and zeros whose convergence is too slow to be neglected is called the **frequency-range-of-interest** (or sometimes “region-of-interest”). Poles and zeros within the frequency range of interest cannot be eliminated without significantly affecting the dynamic response of the transfer function. Quickly converging poles and zeros are outside the frequency-range-of-interest, and can be eliminated from the transfer function. Eliminating poles reduces the order of the

12.4 Model Order Reduction and Deduction

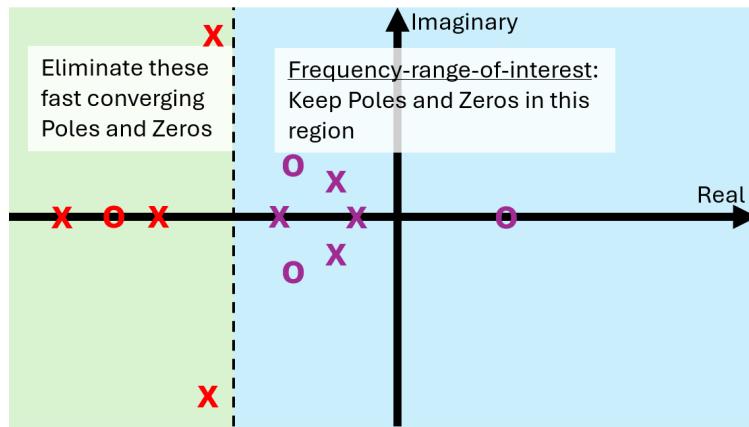


Figure 12.15 Poles and zeros within the frequency-range-of-interest are retained. Poles and zeros outside the frequency-range-of-interest are removed from the transfer function.

transfer function denominator. Eliminating zeros reduces the order of the transfer function numerator. The elimination of poles or zeros is called **model order reduction**.

Example 12.2.2 applied the linear system ID methods of Section 12.1 to analyze flight data from a Boomerang Warbler airplane. A nonlinear output mapping technique improved the model's accuracy. The following example applies model order reduction to simplify the linear model. It shows that a combination of model order reduction, frequency domain system ID, and manually adjusting poles and zeros can produce a lower-order, linear model that improves accuracy for small roll rates.

Model Order Reduction

Example 12.4.1. Model Order Reduction

This example builds on the results of Example 12.2.2 which collected and processed roll data from the Boomerang Warbler flying wing airplane. The aileron command δ_a was the input signal. The output signal was the gyrometer's measurement of the roll rate ω_x in units of rad/s. The following validation graph was the result of applying the linear system ID methods of Section 12.1 (without any input mapping $w = f(u)$ or output mapping $y = g(z)$).

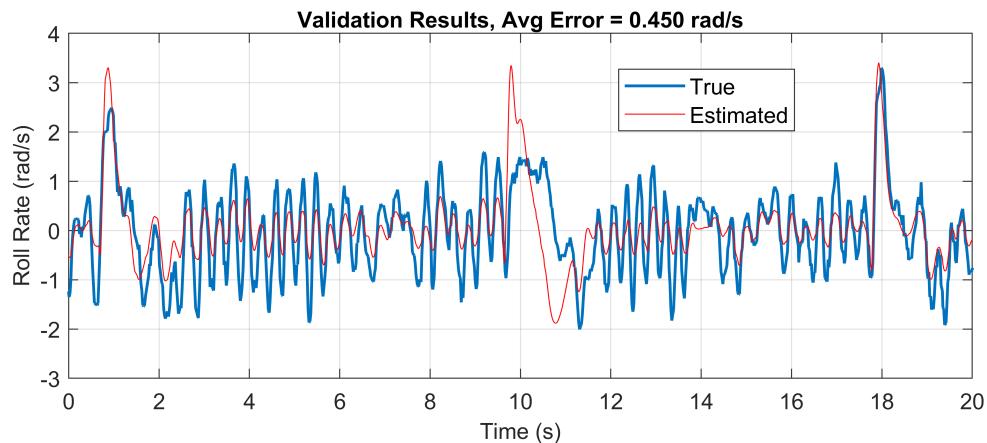


Figure 12.16 Linear system ID results without output mapping

The frequency domain system ID techniques of Section 12.3.3 and Section 12.1 found the following Bode plots for the data of Figure 12.16.

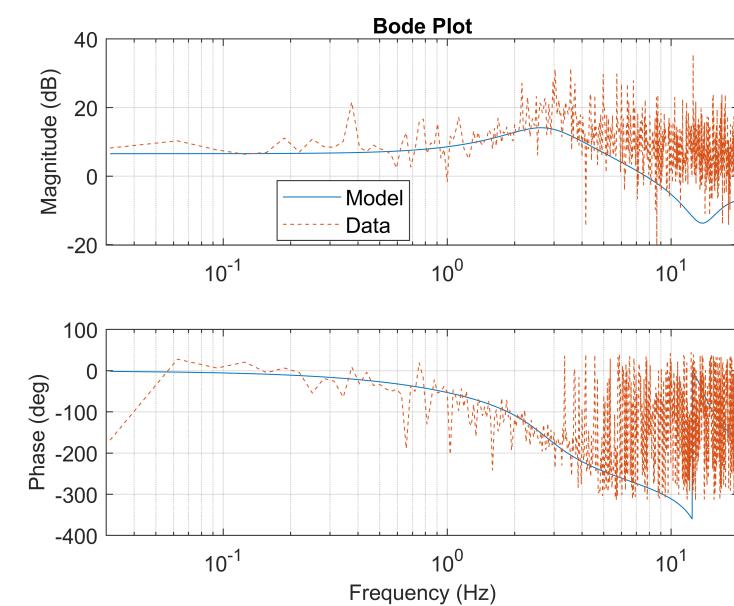


Figure 12.17 Bode plot of the Boomerang Warbler roll rate data

The methods of Section 12.3.4 found the transfer function in zero-pole-gain form to be

$$\frac{\omega_x}{\delta_a} = -0.44 \frac{(s - 46 + j131)(s - 46 - j131)(s - 11)}{(s + 138)(s + 7 + j16)(s + 7 - j16)}$$

which is plotted as the “model” in Figure 12.17. Reduce the transfer function order by eliminating quickly converging poles and zeros. The frequency-range-of-interest is between -10 and +20 on the real part of the complex plane. Manually adjust the remaining poles and zeros to produce a transfer function that causes the model to better match the roll rate ω_x data for small aileron commands δ_a .

Solution: The following pole-zero map shows the poles and zeros along with the frequency-range-of-interest in the complex plane.

12.4 Model Order Reduction and Deduction

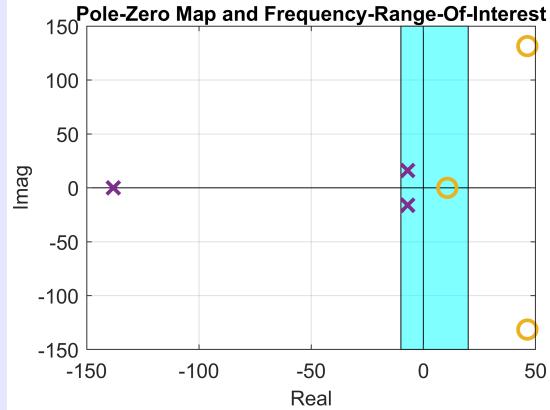


Figure 12.18 Pole-zero map with shading on the frequency-range-of-interest

Only two poles and one zero of the transfer function are within the frequency-range-of-interest. We will remove poles and zeros outside of this region. Eliminating the quickly converging stable pole at $p_3 = -138$ reduces the denominator order from three to two. Eliminating the two faster zeros at $z_{2,3} = 46 \pm j131$ reduces the numerator order to one. By eliminating poles and zeros, we must also adjust the transfer function gain. Also, the Bode magnitude plot of Figure 12.17 shows a large mismatch between the model and the data near the resonance frequency of 3 Hz. To better fit this resonance peak, we manually adjust the remaining poles slightly to be $p_{1,2} = -5.5 \pm j20$. We also adjust the zero to be $z_1 = 9$. We manually adjusted the transfer function gain to be $k = -140$. The adjusted poles and zeros (indicated with larger markers) are shown on the following pole-zero map next to the original ones (indicated with smaller markers).

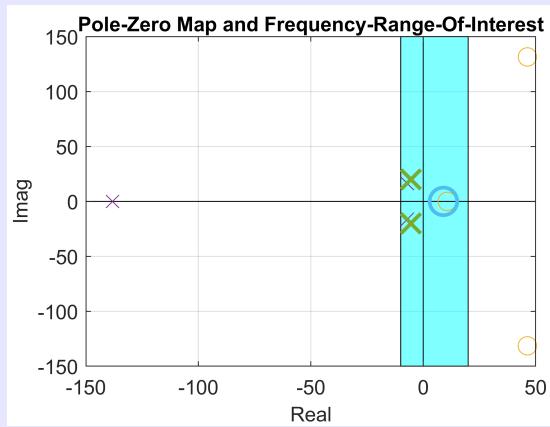


Figure 12.19 Adjusted poles and zeros are indicated with larger markers, original poles and zeros have smaller markers.

The new transfer function is

$$\begin{aligned}\frac{\omega_x}{\delta_a} &= -140 \frac{s - 9}{(s + 5.5 - j20)(s + 5.5 + j20)} \\ &= \frac{-140s + 1260}{s^2 + 11s + 430}\end{aligned}$$

Its Bode plot is labeled “Manual” in the following figure.

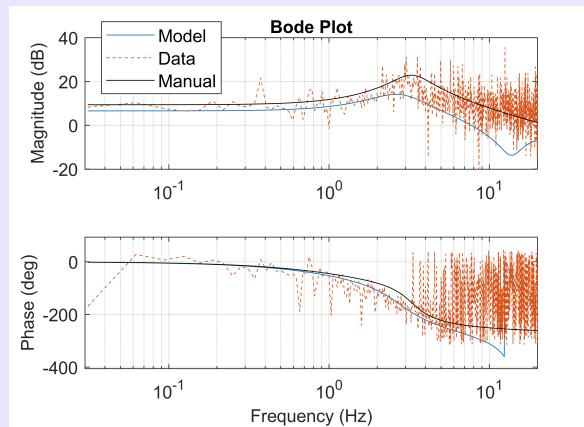


Figure 12.20 Bode plot comparing the reduced-order and manually adjusted transfer function ‘Manual’ with the original “Model” and the flight data “Data”

The validation results of the adjusted model are shown below. For smaller aileron movements, the reduced-order, manually adjusted model fits the data better than the original linear model (see Figure 12.16). However, the original model fit the data better for large aileron commands. This resulted in the original model having a overall smaller average error of 0.45 rad/s compared with the reduced-order model error of 0.496 rad/s.

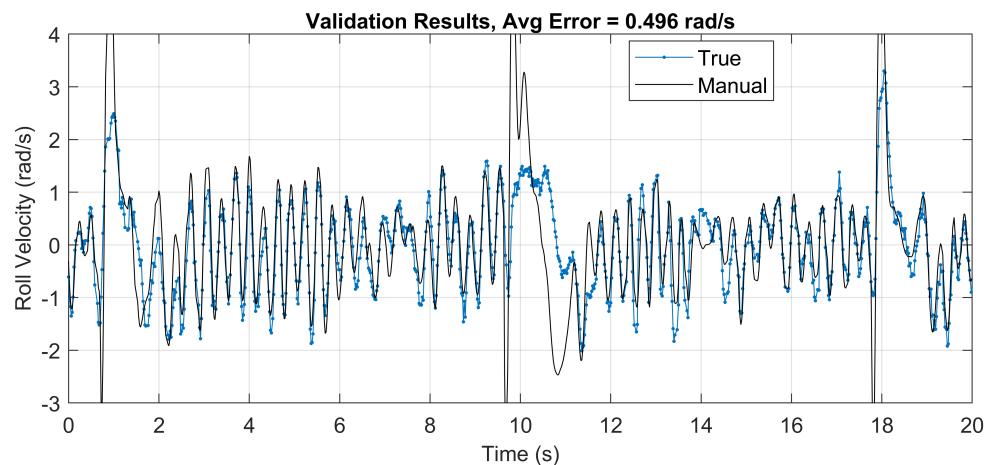


Figure 12.21 Validation results for the reduced-order and adjusted model

12.4 Model Order Reduction and Deduction

There are many approaches to model order reduction besides those described in this book. Modal truncation and balanced truncation are examples of model order reduction techniques that remove poles. Removing pole-zero pairs (zeros can cancel or diminish the effect of poles) is called pole-zero simplification. Some model order reduction algorithms apply singular value decomposition to reduce the model order rather than directly eliminating transfer function poles and zeros.

Chapter 13

Model Based Control Design

Contents

13.1	Full-State Feedback Control using Pole Placement	390
13.1.1	Calculate Feedback Control Gains and Feedforward Gain	391
13.1.2	MATLAB Shortcut Commands	398
13.1.3	Effects of Pole Locations on Closed-Loop Response	399
13.1.4	Determine the Controllability of a System	401
13.1.5	Calculate a Desired Closed Loop Characteristic Equation	402
13.2	Full-State Feedback with Integral Control	403
13.3	Full-State Feedback in Discrete-Time Systems using Pole-Placement	403
13.4	Observer-Based Feedback Control	406
13.4.1	Luenberger Observer-based Feedback Control	407
13.4.2	Observer-Based Feedback with Integral Control	409
13.4.3	Stability of Luenberger-Based Feedback Controllers	410
13.5	Control Design using Root Locus	414
13.6	Control Design using Bode Plots	420
13.6.1	Phase and Gain Margins of Unity Feedback Systems	420
13.6.2	Phase Margins on Bode Plots	421
13.6.3	Gain Margins on Bode Plots	421
13.7	Control Design Requirements	424
13.8	Designing Compensators Using Root Locus and Bode Plots	424
13.8.1	Compensators	425
13.8.2	Adding Poles to Compensators	426
13.8.3	Lead and Lag Compensators	428
13.8.4	Digital Implementation of Compensators	431
13.9	Successive Loop Closure Control Design	434

Chapter 5 introduced PID feedback control and presented block diagrams of dynamic system controllers. Part of the popularity of PID controllers is that they can be used without a mathematical model of the dynamic system. Trial and error can be used to select the PID gains when no model is available. Having a model, however, can allow a PID control designer to select better gains that result in a specific desired closed-loop response. This chapter presents other control design techniques that require a model of the dynamic system, but also allow a designer to more strategically control the response of the system.

13.1 Full-State Feedback Control using Pole Placement

As discussed in Section 1.6, the poles of a system determine the system's stability and response characteristics. Full-state feedback controllers for LTI systems can allow a designer to choose and place the poles of the closed-loop system at any desired location, as long as the actuator does not reach saturation. This section will discuss the method for pole-placement using full-state feedback control.

There are three requirements, or limitations, of the full-state feedback controller of this section: (1) the entire state x must be measured (*e.g.* by sensors) and available for feedback, (2) the state-space matrices A , B , C , and D must be known accurately, and (3) it can only control one signal, *i.e.*, y is scalar. The first limitation can be overcome by using state-estimators such as the Luenberger observer or the Kalman filter. The second limitation can be overcome by applying machine learning and system identification (ID) to learn the state-space model. State-estimators and system ID are topics of Chapter 9 and Chapter 12. The controller can be further refined for uncertain systems by applying an outer control loop with an integrator, which is discussed in Section 13.2. Figure 13.1 shows a block diagram of the full-state feedback controller.

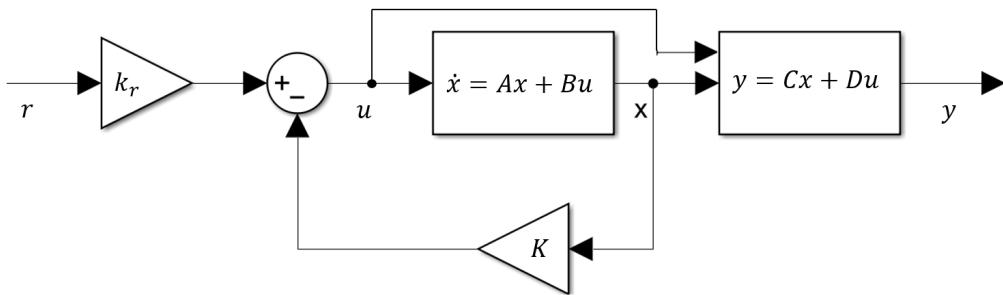


Figure 13.1 Block diagram of the full-state feedback controller

Notice that the state equation $\dot{x} = Ax + Bu$ is in a separate block from the output equation $y = Cx + Du$. This is because the full state x must be available for feedback. To simulate this in Simulink, a state-space block is used with the output matrix as the $n \times n$ identity matrix, $C = I_n$, and the feed-through matrix as the $n \times 1$ zeros matrix $D = 0_{n \times 1}$. The output equation is solved separately as shown in Figure 13.2. The gain blocks must be *Matrix*(K^*u) multiplication instead of *Element-wise*($K.^*u$).

13.1 Full-State Feedback Control using Pole Placement

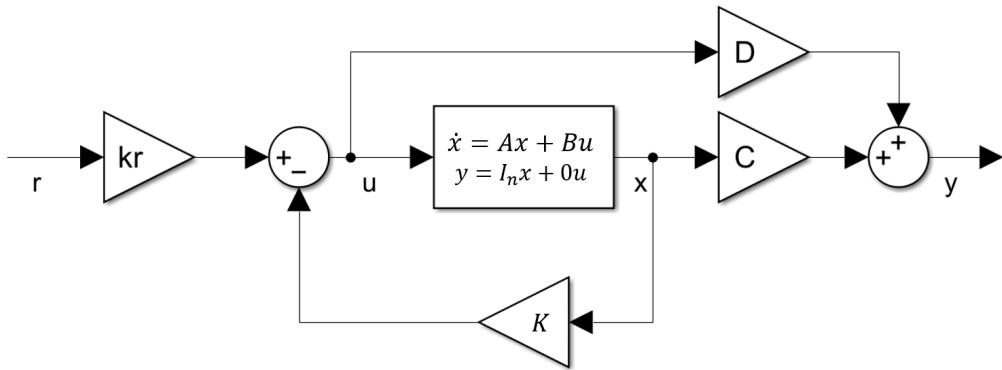


Figure 13.2 Simulation implementation of the full-state feedback controller in Simulink. The gain blocks D , C , K , and k_r must be $\text{Matrix}(K^*u)$ multiplication.

Section 13.1.1 introduces the pole-placement process for calculating the feedback gain K and the feedforward gain k_r for full-state feedback control. Following this process allows the designer to choose the location of the closed loop poles of the controlled system.

13.1.1 Calculate Feedback Control Gains and Feedforward Gain

We can calculate and implement the feedback control gain vector by using the following steps:

Process for Implementing Full-State Feedback Control

1. Calculate the controllability matrix:

$$\mathcal{C} = \begin{bmatrix} B & AB & \dots & A^{n-1}B \end{bmatrix}$$

and check that it is full rank by checking that its determinant is not equal to zero:

$$\det(\mathcal{C}) \neq 0$$

2. Calculate the characteristic equation of the open loop system:

$$\Delta = \det(sI - A)$$

and arrange it in the form:

$$\Delta = s^n + a_{n-1}s^{n-1} + \dots + a_1s + a_0$$

Place the coefficients in the row vector:

$$a = [a_{n-1} \ a_{n-2} \ \dots \ a_1 \ a_0]$$

and use the coefficients to construct the following matrix:

$$\mathcal{A} = \begin{bmatrix} 1 & a_{n-1} & a_{n-2} & \dots & a_2 & a_1 \\ 0 & 1 & a_{n-1} & \dots & a_3 & a_2 \\ 0 & 0 & 1 & \dots & a_4 & a_3 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & \dots & 1 & a_{n-1} \\ 0 & 0 & 0 & \dots & 0 & 1 \end{bmatrix} \quad (13.1)$$

3. Select the desired closed-loop pole locations (p_i , $i = 1, \dots, n$) and determine the closed loop characteristic polynomial Δ_{cl} :

$$\Delta_{cl} = (s - p_1)(s - p_2) \dots (s - p_n)$$

Factor it into the form:

$$\Delta_{cl} = s^n + \alpha_{n-1}s^{n-1} + \dots + \alpha_1s + \alpha_0 \quad (13.2)$$

and place the coefficients in the row vector:

$$\alpha = [\alpha_{n-1} \ \alpha_{n-2} \ \dots \ \alpha_1 \ \alpha_0]$$

4. Calculate the feedback control gain matrix K :

$$K = (\alpha - a)\mathcal{A}^{-1}\mathcal{C}^{-1} \quad (13.3)$$

(Ackermann's formula provides another method to calculate K), and the feedforward gain:

$$k_r = \frac{1}{D - (C - DK)(A - BK)^{-1}B} \quad (13.4)$$

5. Apply the control command:

$$u = k_r r - Kx$$

The following example demonstrates the above process for calculating the control gains.

Full-State Feedback Control

Example 13.1.1. Full-state feedback control

Consider the following state-space system:

$$\begin{aligned} \dot{x} &= \begin{bmatrix} 1 & 0 \\ 2 & 3 \end{bmatrix} x + \begin{bmatrix} 5 \\ 1 \end{bmatrix} u \\ y &= \begin{bmatrix} 0 & 1 \end{bmatrix} x + \begin{bmatrix} 0 \end{bmatrix} u \end{aligned}$$

13.1 Full-State Feedback Control using Pole Placement

Follow the above steps to determine the feedback control gain vector and feedforward gain needed to control this system and place the poles at:

$$p_1 = -2$$

$$p_2 = -3$$

Solution:

- Calculate the controllability matrix and check that it is full rank:

$$\begin{aligned} \mathcal{C} &= \begin{bmatrix} B & AB \end{bmatrix} \\ &= \left[\begin{bmatrix} 5 \\ 1 \end{bmatrix} \quad \begin{bmatrix} 1 & 0 \\ 2 & 3 \end{bmatrix} \begin{bmatrix} 5 \\ 1 \end{bmatrix} \right] \\ &= \begin{bmatrix} 5 & 5 \\ 1 & 13 \end{bmatrix} \end{aligned}$$

$$\det(\mathcal{C}) = 5 \cdot 13 - 1 \cdot 5 = 60 \neq 0$$

$$\text{rank}(\mathcal{C}) = 2$$

- Calculate the characteristic equation of the open loop system:

$$\begin{aligned} \Delta &= \det(sI - A) \\ &= \det \left(\begin{bmatrix} s & 0 \\ 0 & s \end{bmatrix} - \begin{bmatrix} 1 & 0 \\ 2 & 3 \end{bmatrix} \right) \\ &= \det \begin{bmatrix} s-1 & 0 \\ -2 & s-3 \end{bmatrix} \\ &= (s-1)(s-3) - (-2)(0) \\ &= s^2 - 4s + 3 \end{aligned}$$

and place the coefficients in the row vector:

$$a = [-4 \quad 3]$$

and use the coefficients to construct the following matrix:

$$\mathcal{A} = \begin{bmatrix} 1 & -4 \\ 0 & 1 \end{bmatrix}$$

- Using the specified pole locations, determine the closed loop characteristic equation:

$$\begin{aligned} \Delta_{cl} &= (s - (-2))(s - (-3)) \\ &= (s+2)(s+3) \\ &= s^2 + 5s + 6 \end{aligned}$$

and place the coefficients in the row vector:

$$\alpha = [5 \ 6]$$

4. Calculate the feedback control gains:

$$\begin{aligned} K &= (\alpha - a)\mathcal{A}^{-1}\mathcal{C}^{-1} \\ &= \left([5 \ 6] - [-4 \ 3] \right) \begin{bmatrix} 1 & -4 \\ 0 & 1 \end{bmatrix}^{-1} \begin{bmatrix} 5 & 5 \\ 1 & 13 \end{bmatrix}^{-1} \\ &= [1.3 \ 2.5] \end{aligned}$$

and the feedforward gain (with $D = 0$):

$$\begin{aligned} k_r &= -[C(A - BK)^{-1}B]^{-1} \\ &= - \left[[0 \ 1] \left(\begin{bmatrix} 1 & 0 \\ 2 & 3 \end{bmatrix} - \begin{bmatrix} 5 \\ 1 \end{bmatrix} [1.3 \ 2.5] \right)^{-1} \begin{bmatrix} 5 \\ 1 \end{bmatrix} \right]^{-1} \\ &= \frac{6}{9} \end{aligned}$$

5. The control gains are used to calculate the control input given the reference input r and the states x :

$$\begin{aligned} u &= k_r r - Kx \\ &= \frac{6}{9}r - [1.3 \ 2.5]x \end{aligned}$$

The block diagram of Figure 13.3 implements the feedback controller derived in this example. To feedback the full state x , the matrices in the State-Space Simulink block were set to

$$A = \begin{bmatrix} 1 & 0 \\ 2 & 3 \end{bmatrix}, B = \begin{bmatrix} 5 \\ 1 \end{bmatrix}, C = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}, D = \begin{bmatrix} 0 \\ 0 \end{bmatrix}, x_0 = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

13.1 Full-State Feedback Control using Pole Placement

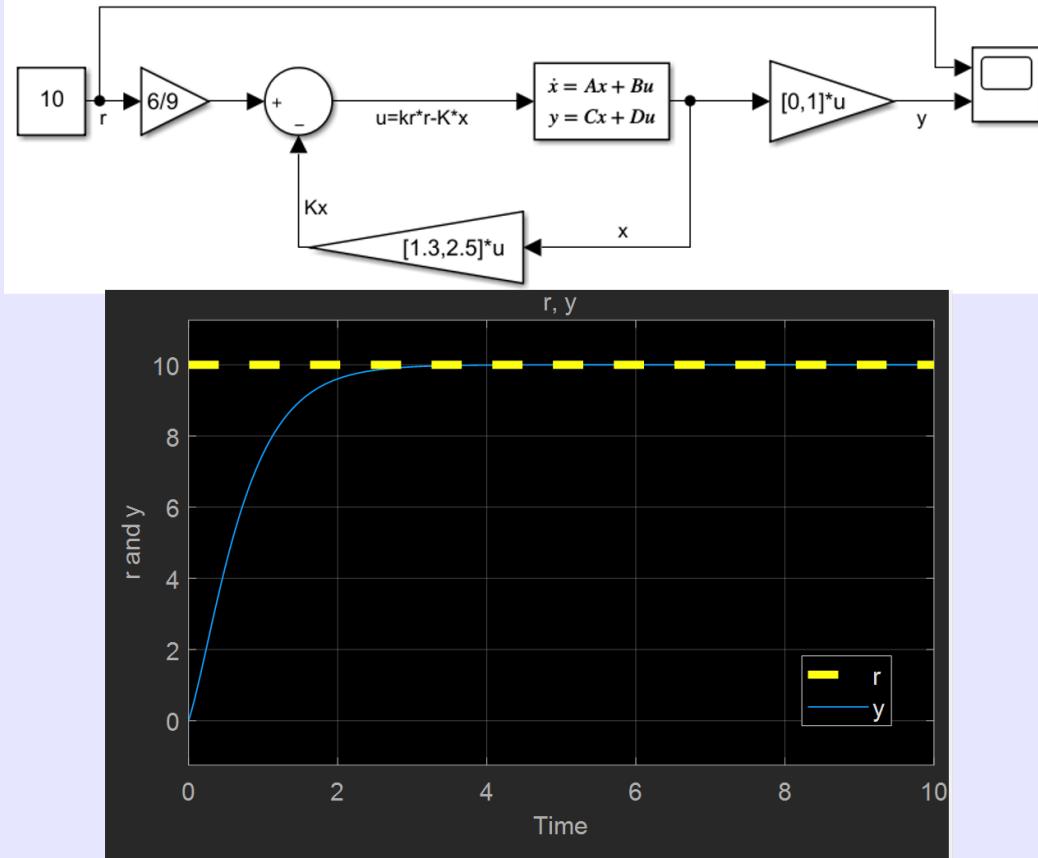


Figure 13.3 Simulink block diagram and scope output of the full-state feedback controller

The following example derives the feedforward gain k_r of Eq. (13.4).

Derivation of the Feedforward Gain k_r

Example 13.1.2. Derivation of the feedforward gain k_r of Eq. (13.4)

From the feedback loop in Figure 13.1, the control input u is calculated to be:

$$u = k_r r - kx$$

Using this in the state equation gives:

$$\begin{aligned}\dot{x} &= Ax + Bu \\ &= Ax + B(k_r r - Kx) \\ &= (A - BK)x + Bk_r r\end{aligned}$$

If the reference input r and the output value y are single values (not vectors) then k_r is a single scalar. We would like to choose a value of k_r such that $y = r$ when the system reaches steady-state.

If the reference input is a constant, and the system is at steady-state, then the derivatives will be zero because the system will be at rest:

$$0 = (A - BK)x_{ss} + Bk_r r$$

This means we can solve for the steady-state values of the states to be:

$$x_{ss} = -(A - BK)^{-1}Bk_r r$$

and the steady-state output will be:

$$\begin{aligned} y_{ss} &= Cx_{ss} + Du \\ &= Cx_{ss} + D(-Kx_{ss} + k_r r) \\ &= (C - DK)x_{ss} + Dk_r r \\ &= (D - (C - DK)(A - BK)^{-1}B)k_r r \end{aligned}$$

If we want $y_{ss} = r$ then we need the terms pre-multiplying r to be equal to one:

$$1 = (D - (C - DK)(A - BK)^{-1}B)k_r$$

and solving for the feedforward gain k_r we get:

$$k_r = \frac{1}{D - (C - DK)(A - BK)^{-1}B}$$

Now, if we know the state matrices A , B , C and D , and the control gain vector K , we can calculate the value of the feedforward gain k_r to make the output equal to the desired reference input when the system reaches steady state. If either $(A - BK)$ or $(D - (C - DK)(A - BK)^{-1}B)$ is not invertible, then k_r does not exist.

The following example proves the pole-placement gain K formula of Eq. (13.3).

Derivation of the Pole-Placement Feedback Gain K

Example 13.1.3. Derivation of the pole-placement feedback gain K of Eq. (13.3)

Derive the full-state feedback control gain matrix K of Eq. (13.3) for an open-loop state-space system:

$$\begin{aligned} \dot{x} &= Ax + Bu \\ y &= Cx + Du \end{aligned}$$

whose controllable canonical realization is given by

$$\begin{aligned} \dot{z} &= A_c z + B_c u \\ y &= C_c z + D_c u \end{aligned}$$

where

13.1 Full-State Feedback Control using Pole Placement

$$A_c = \begin{bmatrix} 0 & 1 & 0 & \dots & 0 \\ 0 & 0 & 1 & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \dots & 1 \\ -a_0 & -a_1 & -a_2 & \dots & -a_{n-1} \end{bmatrix} \quad B_c = \begin{bmatrix} 0 \\ 0 \\ \vdots \\ 0 \\ 1 \end{bmatrix}$$

and a_i , $i = 0, \dots, n-1$ are the coefficients of the characteristic polynomial $\det(sI - A)$. The coordinate transform to the controllable canonical form is $x = Tz$, where the transformation matrix T is given by Eq. (2.21):

$$T = \mathcal{C}(A, B) [\mathcal{C}(A_c, B_c)]^{-1}$$

Solution: The proof of Eq. (2.21) was provided in Example 2.3.1. The controllable canonical form is convenient for finding the full-state feedback gain. Substituting the full-state feedback control law $u = -K_c z$ into the controllable canonical state-equation gives

$$\dot{z} = (A_c - B_c K_c) z \quad (13.5)$$

$$\dot{z} = \begin{bmatrix} 0 & 1 & 0 & \dots & 0 \\ 0 & 0 & 1 & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \dots & 1 \\ -a_0 - K_{c,0} & -a_1 - K_{c,1} & -a_2 - K_{c,2} & \dots & -a_{n-1} - K_{c,n-1} \end{bmatrix} z \quad (13.6)$$

where the full-state feedback gain matrix is

$$K_c = [K_{c,n-1} \ K_{c,n-2} \ \dots \ K_{c,0}]$$

Notice that the elements $K_{c,i}$ in the gain vector K_c appear in the opposite order in Eq. (13.6). The desired closed loop polynomial was found in Eq. (13.2) to be $\Delta_{cl} = s^n + \alpha_{n-1}s^{n-1} + \dots + \alpha_1s + \alpha_0$. Therefore, the desired controllable canonical state-equation is

$$\dot{z} = \begin{bmatrix} 0 & 1 & 0 & \dots & 0 \\ 0 & 0 & 1 & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \dots & 1 \\ -\alpha_0 & -\alpha_1 & -\alpha_2 & \dots & -\alpha_{n-1} \end{bmatrix} z \quad (13.7)$$

By comparing Eq. (13.7) with Eq. (13.6), we find the elements of the gain matrix K_c to be

$$[K_{c,n-1} \ K_{c,n-2} \ \dots \ K_{c,0}] = [(\alpha_{n-1} - a_{n-1}) \ (\alpha_{n-2} - a_{n-2}) \ \dots \ (\alpha_0 - a_0)] \quad (13.8)$$

Therefore, K_c is known. To get K , we need to convert K_c back to the original state-space realization. Since $u = -K_c z$ and $x = Tz$, then $z = T^{-1}x$. Therefore,

$$\begin{aligned} u &= -K_c z \\ &= -K_c T^{-1} x \end{aligned}$$

which indicates that

$$K = K_c T^{-1} \quad (13.9)$$

$$= K_c \left[\mathcal{C}(A, B) [\mathcal{C}(A_c, B_c)]^{-1} \right]^{-1} \quad (13.10)$$

except that K_c has its elements in the opposite order of $(\alpha - a)$. If we reverse the order of the columns of $[\mathcal{C}(A_c, B_c)]^{-1}$ (e.g. by using MATLAB's `fliplr([\mathcal{C}(A_c, B_c)]^{-1})` command, we get \mathcal{A} , i.e., $\text{fliplr}([\mathcal{C}(A_c, B_c)]^{-1}) = \mathcal{A}$, where \mathcal{A} is defined by Eq. (13.1), (for proof, refer to Example 2.3.1). Therefore, Eq. (13.10) becomes

$$\begin{aligned} K &= K_c T^{-1} \\ &= (\alpha - a) \left[\mathcal{C}(A, B) \mathcal{A} \right]^{-1} \\ &= (\alpha - a) \mathcal{A}^{-1} [\mathcal{C}(A, B)]^{-1} \end{aligned}$$

and the proof is complete. Notice, however, that we could have also reversed the order of $(\alpha - a)$ to get the following equivalent solution for the gain matrix K :

$$\begin{aligned} K &= \text{fliplr}(\alpha - a) T^{-1} \\ &= \text{fliplr}(\alpha - a) \mathcal{C}(A_c, B_c) [\mathcal{C}(A, B)]^{-1} \end{aligned}$$

13.1.2 MATLAB Shortcut Commands

MATLAB has shortcut commands to determine controllability and calculate the feedback control gains:

- The “ctrb(A,B)” command can be used to calculate the controllability matrix.
- The “rank(C)” command can be used to determine the rank of the controllability matrix.
- The “place(A,B,p)” or “acker(A,B,p)” command can be used to calculate the feedback control gain K needed to place the poles at the locations listed in the vector p . The “place” command cannot place two or more poles at the same location, but the “acker” command can. The “acker” command, however, is not as numerically well-conditioned as the “place” command.

13.1 Full-State Feedback Control using Pole Placement

13.1.3 Effects of Pole Locations on Closed-Loop Response

As discussed in Section 2.4, the location of the poles governs the response of the system to the input. If the real component of the poles is positive, then the poles are unstable, and the response will grow exponentially. This can be a severe problem when trying to control the states of a system. For example, if we try to control the output to go to the reference input using a feedforward controller only, but the poles are unstable, the output will not go to the desired reference input. Instead, it will grow exponentially.

With full-state feedback, if we have an inherently unstable system with poles on the right-hand plane, we can create a feedback controller to move the poles to the left-hand plane. Consider the state-space model:

$$\begin{aligned}\dot{x} &= Ax + Bu \\ y &= Cx + Du\end{aligned}$$

The poles of the system are calculated as the eigenvalues of the A matrix:

$$\det(\lambda I - A) = 0$$

However, if we place the system in a full-state feedback control loop shown in Figure 13.1, the input u is:

$$u = k_r r - Kx$$

and using this in the state equation gives:

$$\begin{aligned}\dot{x} &= Ax + B(k_r r - Kx) \\ \dot{x} &= (A - BK)x + Bk_r r\end{aligned}$$

Thus, there is now a new state equation, or a closed loop state equation. Defining the new closed loop A_{cl} and B_{cl} matrices to be:

$$\begin{aligned}A_{cl} &= A - BK \\ B_{cl} &= Bk_r\end{aligned}$$

then the closed loop state equations are:

$$\begin{aligned}\dot{x} &= A_{cl}x + B_{cl}r \\ y &= Cx\end{aligned}$$

Now we can calculate the poles, or eigenvalues, of the closed loop system from:

$$\begin{aligned}\det(\lambda I - A_{cl}) &= 0 \\ \det(\lambda I - (A - BK)) &= 0\end{aligned}$$

and we can see that the values we choose for K will affect the location of the poles. Thus, if we have an unstable system, we could possibly choose a value of K to stabilize the system by moving the location of the poles to the left-hand plane. We could also place the poles in a location that gives a desired time-constant or oscillation frequency. The following example demonstrates how the choice of the control gain K can cause an open-loop unstable system to become a closed-loop stable system.

Closed Loop Poles of a Full-State Feedback System

Example 13.1.4. Closed loop poles of a full-state feedback system

Determine the open loop poles of the following state-space system:

$$\begin{bmatrix} \dot{x} \\ \dot{v} \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 2 & -1 \end{bmatrix} \begin{bmatrix} x \\ v \end{bmatrix} + \begin{bmatrix} 0 \\ 1 \end{bmatrix} u(t)$$

$$y = \begin{bmatrix} 1 & 0 \end{bmatrix} \begin{bmatrix} x \\ v \end{bmatrix}$$

and the closed loop poles using the following feedback control gain vector:

$$K = \begin{bmatrix} 4 & 1 \end{bmatrix}$$

Solution: The eigenvalues of the open loop system (without being placed in a feedback controller yet) are calculated by first calculating the characteristic equation:

$$\det\left(\lambda \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} - \begin{bmatrix} 0 & 1 \\ 2 & -1 \end{bmatrix}\right) = 0$$

$$\det\left(\begin{bmatrix} \lambda & -1 \\ -2 & \lambda + 1 \end{bmatrix}\right) = 0$$

$$\lambda(\lambda + 1) - (-2)(-1) = 0$$

$$\lambda^2 + \lambda - 2 = 0$$

then solving for the poles:

$$\lambda_{1,2} = \frac{-1 \pm \sqrt{1^2 - 4(1)(-2)}}{2(1)}$$

$$\lambda_{1,2} = \frac{-1 \pm \sqrt{9}}{2}$$

$$= -0.5 \pm 1.5$$

$$= -2, 1$$

So it appears the open loop system has one stable pole at -2 and one unstable pole at +1.

Now, using the feedback control gain vector of:

$$K = \begin{bmatrix} 4 & 1 \end{bmatrix}$$

13.1 Full-State Feedback Control using Pole Placement

the closed loop A_{cl} matrix is:

$$\begin{aligned} A_{cl} &= A - BK \\ &= \begin{bmatrix} 0 & 1 \\ 2 & -1 \end{bmatrix} - \begin{bmatrix} 0 \\ 1 \end{bmatrix} \begin{bmatrix} 4 & 1 \end{bmatrix} \\ &= \begin{bmatrix} 0 & 1 \\ 2 & -1 \end{bmatrix} - \begin{bmatrix} 0 & 0 \\ 4 & 1 \end{bmatrix} \\ &= \begin{bmatrix} 0 & 1 \\ -2 & -2 \end{bmatrix} \end{aligned}$$

and the characteristic equation is found to be:

$$\begin{aligned} \det\left(\lambda \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} - \begin{bmatrix} 0 & 1 \\ -2 & -2 \end{bmatrix}\right) &= 0 \\ \det\left(\begin{bmatrix} \lambda & -1 \\ 2 & \lambda + 2 \end{bmatrix}\right) &= 0 \\ \lambda(\lambda + 2) - (2)(-1) &= 0 \\ \lambda^2 + 2\lambda + 2 &= 0 \end{aligned}$$

resulting in the closed loop eigenvalues:

$$\begin{aligned} \lambda_{1,2} &= \frac{-2 \pm \sqrt{2^2 - 4(1)(2)}}{2(1)} \\ \lambda_{1,2} &= \frac{-2 \pm \sqrt{-4}}{2} \\ &= -1 \pm 1j \end{aligned}$$

Thus our feedback control gain vector was able to move the locations of the poles to both having a real value of -1 (which is stable) and imaginary values of ± 1 .

13.1.4 Determine the Controllability of a System

In some cases, we may not be able to place the poles at a desired location. In order to determine whether or not we can move all of the poles, we need to determine the controllability of the system. To check the controllability of a system we calculate the following controllability matrix:

$$\mathcal{C} = \begin{bmatrix} B & AB & \dots & A^{n-1}B \end{bmatrix}$$

where n is the order of size of the state matrix.

Next, we check the rank of the controllability matrix and make sure that it is full rank:

$$\text{rank}(\mathcal{C}) = n$$

which basically means all of the rows and columns of the controllability matrix are unique. So if you row reduce the matrix, you will not get a row of zeros.

One way to simply check that the controllability matrix is full rank is to make sure that its determinant is not equal to zero:

$$\det(\mathcal{C}) \neq 0$$

If the controllability matrix is full rank, then all of the closed loop poles can be placed at a location we choose using the feedback control gain vector K .

13.1.5 Calculate a Desired Closed Loop Characteristic Equation

Before we can calculate the feedback control gain vector K , we need to be able to determine the desired pole locations of the closed loop system. The method for choosing the pole locations depends on the desired response of the system. The effect of pole locations on the system response was discussed previously. A more thorough description of how to select desired pole locations is covered in a System Dynamics course.

Note that the desired pole locations should all have negative real components, and any complex poles will need to have a complex conjugate.

Once we have chosen the desired pole locations:

$$p_1, p_2, \dots, p_n$$

and knowing that the poles are the roots of the characteristic equation, the characteristic equation is simply calculated by expanding the following closed loop characteristic equation:

$$\Delta_{cl} = (s - p_1)(s - p_2) \dots (s - p_n)$$

into the form:

$$\Delta_{cl} = s^n + \alpha_{n-1}s^{n-1} + \dots + \alpha_1s + \alpha_0$$

Full-State Feedback Closed Loop Characteristic Polynomial

Example 13.1.5. Closed Loop Characteristic Polynomial of a Full-State Feedback System

Suppose we would like to control a third-order system ($n = 3$) and place the poles at:

$$\begin{aligned} p_1 &= -3 \\ p_2 &= -2 + 1j \\ p_3 &= -2 - 1j \end{aligned}$$

Calculate the closed loop characteristic equation.

Solution: The closed loop characteristic equation is calculated from the closed loop poles as:

$$\begin{aligned} \Delta_{cl} &= (s - (-3))(s - (-2 + j))(s - (-2 - j)) \\ &= (s + 3)(s + 2 - j)(s + 2 + j) \\ &= (s + 3)(s^2 + 4s + 5) \\ &= s^3 + 7s^2 + 17s + 15 \end{aligned}$$

13.2 Full-State Feedback with Integral Control

It can be rare to perfectly know the state-space matrices A , B , C , and D for any real system. Model uncertainty can degrade the performance of the full-state feedback controllers of the previous section, especially the convergence of the output y to the reference input r . An outer feedback loop with an integrator can be used to compensate for this uncertainty. The block diagram for full-state feedback with integral control is shown in Figure 13.4.

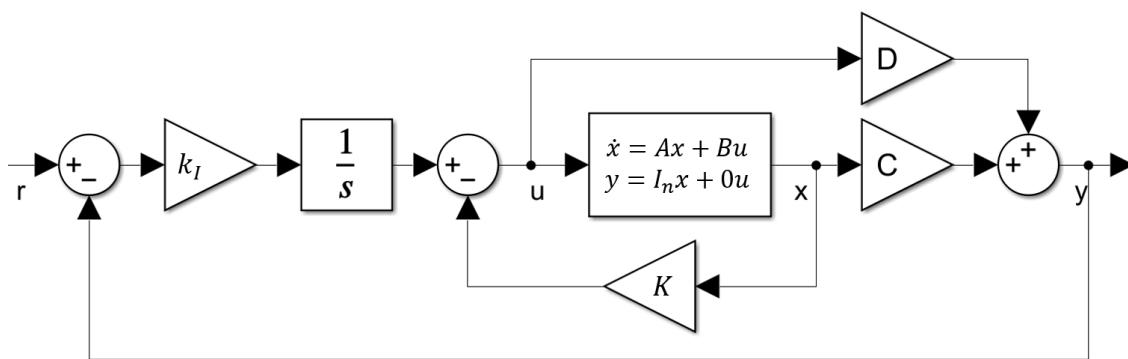


Figure 13.4 Block diagram of full-state feedback with integral control

The integral control gain k_I is a tuning parameter. Successive loop closure controller design, discussed in Section 13.9, suggests that the integral control gain k_I should be

$$k_I \leq \frac{k_r}{5\tau_{cl}} \quad (13.11)$$

where k_r is the feedforward gain calculated in Eq. (13.4), and τ_{cl} is the time-constant of the closed-loop full-state feedback controller, *i.e.*,

$$\tau_{cl} = \frac{-1}{\max(\text{Real}(p_i))}, \quad i = 1, \dots, n \quad (13.12)$$

where p_i are the user-selected poles of the full-state feedback system (see Section 13.1.1).

13.3 Full-State Feedback in Discrete-Time Systems using Pole-Placement

The pole-placement procedure for full-state feedback control of discrete-time systems

$$y_k = Cx_k + Du_k$$

$$x_{k+1} = A_d x_k + B_d u_k$$

is similar to continuous-time systems. There are only a few important differences. It is important to remember that stability in discrete-time systems requires poles to be in the unit circle of the complex plane (see Section 3.1.7). The mapping from continuous-time poles to discrete-time poles can be found by $z = e^{s\Delta t}$, where s is the continuous-time pole and z is the corresponding discrete-time pole.

The feedback control u_k for discrete-time is similar to continuous time:

$$u_k = k_{r,d} r_k - K_d x_k \quad (13.13)$$

The reference signal is r_k , the feedback gain is K_d , and the feedforward gain is $k_{r,d}$. The full-state feedback gain K_d can be calculated using the “place” or “acker” command:

$$K_d = \text{place}(A_d, B_d, p_d) \quad (13.14)$$

where p_d is an array of the desired pole locations of the closed loop system.

The formula for calculating the feedforward gain $k_{r,d}$ is slightly different from continuous-time:

$$k_{r,d} = \frac{1}{D + (C - DK_d)(I - A_d + B_d K_d)^{-1} B_d} \quad (13.15)$$

The example below shows how to create identical full-state feedback controllers in both continuous-time and discrete-time systems.

Discrete-Time Full-State Feedback

Example 13.3.1. Full-State Feedback in Discrete-Time Systems

Create MATLAB code to implement identical full-state feedback controllers using both continuous-time and discrete-time pole-placement techniques. Use random matrices for the state-space equations $\dot{x} = Ax + Bu$ and $y = Cx + Du$.

Solution: The solution is shown below in the following MATLAB code. The graph produced by the code in Figure 13.5 shows that both continuous and discrete-time implementations produce identical results.

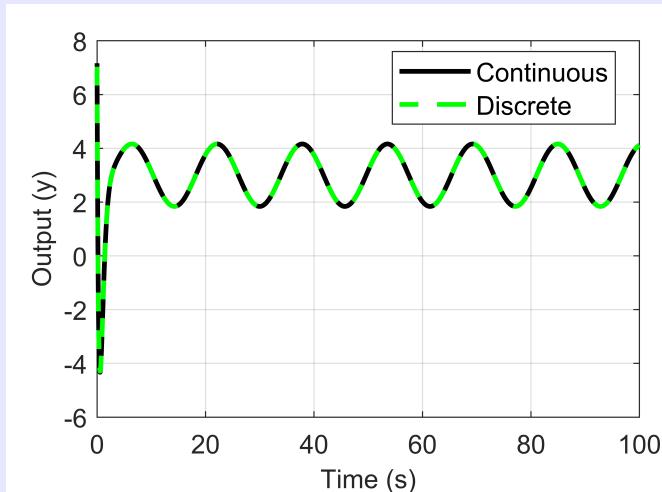


Figure 13.5 Results of discrete and continuous-time pole placement control

```
%Simulation time parameters
dt = 0.01; %(s) simulation time step
t = 0:dt:100; %(s) time vector
N = length(t); % length of the time vector
```

13.3 Full-State Feedback in Discrete-Time Systems using Pole-Placement

```
%reference input
r = 3+1*sin(0.4*t);

%continuous time parameters
A = randn(2,2); %State matrix in dx=Ax+Bu
B = randn(2,1); %Input matrix in dx=Ax+Bu
C = randn(1,2); %Output matrix in y=Cx+Du
D = randn; %Feedthrough matrix in y=Cx+Du

%discrete-time parameters in x_{k+1}=Ad*x_k+Bd*u_k
Fd = expm([A*dt, B*dt; zeros(1,3)]);
Ad = Fd(1:2,1:2);
Bd = Fd(1:2,3);

%desired continuous-time pole locations
p = [-2+1i, -2-1i] %Continuous-time closed loop pole locations
K = place(A,B,p); %Continuous-time full-state feedback gain
kr = 1/(D-(C-D*K)*(A-B*K)^(-1)*B); %Feedforward gain
pa = eig(A-B*K) %Actual closed loop pole locations (check)

%equivalent desired discrete-time pole locations
pd = exp(p*dt) %Discrete-time closed loop pole locations
Kd = place(Ad,Bd,pd); %Discrete-time full-state feedback gain
krd = 1/(D+(C-D*Kd)*(eye(2)-Ad+Bd*Kd)^(-1)*Bd); %Feedforward gain
pda = eig(Ad-Bd*Kd) %Actual closed loop pole locations (check)

%% simulation
y = zeros(1,N); %Allocate memory for the continuous-time output
yd = zeros(1,N); %Allocate memory for the discrete-time output
x = [0;0]; %Initial condition of the continuous-time state
xd = [0;0]; %Initial condition of the discrete-time state
for ii = 1:N
    u = kr*r(ii)-K*x; %Feedback controller for continuous-time
    ud = krd * r(ii) - Kd*xd; %Feedback controller for discrete-time
    y(ii) = C*x+D*u; %Continuous-time output
    yd(ii) = C*xd+D*ud; %Discrete-time output
    x = Ad*x+Bd*u; %Solution of the continuous-time simulation
    xd = Ad*xd+Bd*ud; %Solution of the discrete-time simulation
end
%Plot and compare the results
figure
plot(t, y,'k', t, yd,'g--','LineWidth',2)
xlabel('Time (s)')
ylabel('Output (y)')
legend('Continuous','Discrete','Location','Best')
```

13.4 Observer-Based Feedback Control

Full-state feedback control is powerful because it places the closed loop poles at user-specified locations. The A_d , B_d , C , and D matrices must be known, and they could be determined experimentally using the system ID techniques of Chapter 12. However, a main limitation of full-state feedback control is that the entire state x must be measured, e.g., with sensors. This chapter eliminates that limitation by using observers to estimate the state. Chapter 9 discussed two different types of state estimators: Kalman filters and Luenberger observers. The Kalman filter is optimal but more computationally complex. The Luenberger observer is less optimal, but it places the observer poles at user-selected locations within the complex plane. This allows us to study the stability of the Luenberger observer-based feedback controller.

An observer-based feedback controller is shown in Figure 13.6. The block diagram shows two different outputs from the state-space system: y and y_m . The output y_m must be measured. The output y does not need to be measured unless integral control is added as discussed in Section 13.2. Rather, the purpose of the controller is to cause y to eventually match the reference input r . The reason for two outputs is because the desired output y that tracks the reference r might not be the measured output y_m .

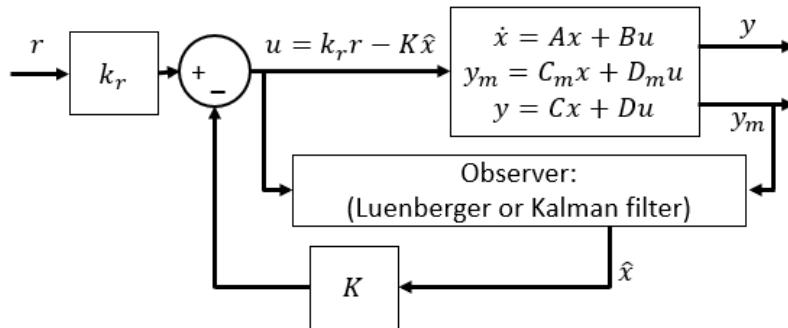


Figure 13.6 Observer-based feedback control

The observer estimates the state x . The state estimate is given the symbol \hat{x} . The Luenberger observer calculates the state-estimate by solving the following dynamic equation for \hat{x} :

$$\dot{\hat{x}} = A\hat{x} + Bu + L(y_m - (C_m\hat{x} + D_m u)) \quad (13.16)$$

Details about the Luenberger observer were described in Section 9.3. The Kalman filter estimates the state using model prediction and sensor correction steps:

Model Prediction	Sensor Correction
$x_p = A_d\hat{x}_k + B_d u_k, \quad (1)$	$S = CP_p C^T + R, \quad (4)$
$P_p = A_d \hat{P}_k A_d^T + Q, \quad (2)$	$\mathcal{K} = P_p C^T S^{-1}, \quad (5)$
$y_p = C x_p + D u_k, \quad (3)$	$\hat{x}_{k+1} = x_p + \mathcal{K}(y_{m,k} - y_p), \quad (6)$
	$\hat{P}_{k+1} = (I_n - \mathcal{K}C) P_p, \quad (7)$

where $y_{m,k}$ is the value of the measured output at time t_k . The details of the Kalman filter were discussed

13.4 Observer-Based Feedback Control

in Section 9.2. For observer-based feedback control, either the Luenberger observer or the Kalman filter can be used for the Observer block in Figure 13.6.

If estimation is the goal instead of control, either the Luenberger observer or the Kalman filter can be used without a feedback controller to estimate the state x . Feedback control is not required for state-estimation. The block diagram for state-estimation without feedback control is shown in Figure 13.7.

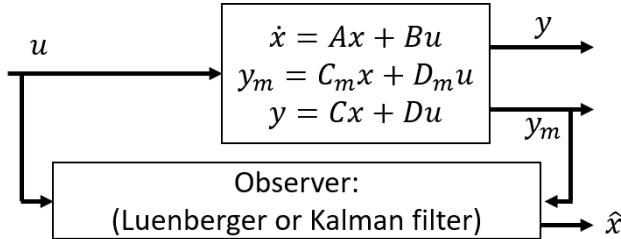


Figure 13.7 State-estimation without feedback control

13.4.1 Luenberger Observer-based Feedback Control

This section provides the steps for implementing a Luenberger observer in a feedback controller. The block diagram is shown in Figure 13.8.

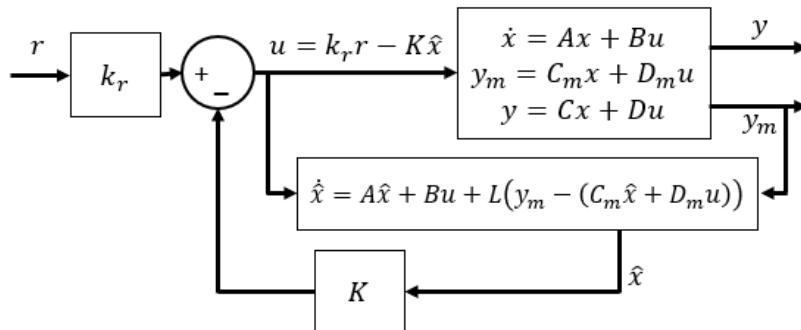


Figure 13.8 Luenberger observer-based feedback control

Process for Luenberger Observer-based Feedback Control

1. Calculate the observability matrix:

$$\mathcal{O} = \begin{bmatrix} C_m \\ C_m A \\ \vdots \\ C_m A^{n-1} \end{bmatrix}$$

and check that it is full rank by checking that its determinant is not equal to zero:

$$\det(\mathcal{O}) \neq 0$$

2. Calculate the characteristic equation of the open loop system:

$$\Delta = \det(sI - A)$$

and arrange it in the form:

$$\Delta = s^n + a_{n-1}s^{n-1} + \cdots + a_1s + a_0$$

Place the coefficients in the row vector:

$$a = [a_{n-1} \ a_{n-2} \ \cdots \ a_1 \ a_0]$$

and use the coefficients to construct the following matrix:

$$\mathcal{A} = \begin{bmatrix} 1 & a_{n-1} & a_{n-2} & \cdots & a_1 \\ 0 & 1 & a_{n-1} & \cdots & a_2 \\ \vdots & \vdots & \ddots & \ddots & \vdots \\ 0 & 0 & \cdots & 1 & a_{n-1} \\ 0 & 0 & 0 & \cdots & 1 \end{bmatrix}$$

3. Using the desired observer pole locations o_1, o_2, \dots, o_n , determine the observer characteristic equation:

$$\Delta_O = (s - o_1)(s - o_2) \cdots (s - o_n)$$

Factor it into the form:

$$\Delta_O = s^n + \beta_{n-1}s^{n-1} + \cdots + \beta_1s + \beta_0$$

and place the coefficients in the row vector:

$$\beta = [\beta_{n-1} \ \beta_{n-2} \ \cdots \ \beta_1 \ \beta_0]$$

4. Calculate the observer gains:

$$L = \mathcal{O}^{-1} (\mathcal{A}^T)^{-1} (\beta - a)^T$$

5. To use the observer, you may need to convert it to discrete-time. To calculate the discrete-time observer gain L_d , use a process similar to finding the input transition matrix B_d :

$$\begin{bmatrix} A_d & L_d \\ \mathbf{0} & \mathbf{I} \end{bmatrix} = \text{expm}\left(\begin{bmatrix} A\Delta t & L\Delta t \\ \mathbf{0} & \mathbf{0} \end{bmatrix}\right)$$

where $\text{expm}()$ is the matrix exponential operator.

13.4 Observer-Based Feedback Control

6. Calculate the controllability matrix:

$$\mathcal{C} = \begin{bmatrix} B & AB & \dots & A^{n-1}B \end{bmatrix}$$

and check that it is full rank by checking that its determinant is not equal to zero:

$$\det(\mathcal{C}) \neq 0$$

7. Select the desired pole locations and determine the closed loop characteristic equation:

$$\Delta_{cl} = (s - p_1)(s - p_2) \dots (s - p_n)$$

Factor it into the form:

$$\Delta_{cl} = s^n + \alpha_{n-1}s^{n-1} + \dots + \alpha_1s + \alpha_0$$

and place the coefficients in the row vector:

$$\alpha = \begin{bmatrix} \alpha_{n-1} & \alpha_{n-2} & \dots & \alpha_1 & \alpha_0 \end{bmatrix}$$

8. Calculate the feedback control gains:

$$K = (\alpha - a)\mathcal{A}^{-1}\mathcal{C}^{-1}$$

and the feedforward gain:

$$k_r = \frac{1}{D - (C - DK)(A - BK)^{-1}B}$$

9. At each time iteration t_k , apply the control command:

$$u_k = k_r r_k - K\hat{x}_k$$

10. Implement the Luenberger observer and calculate the state-estimate \hat{x} . The discrete-time solution is

$$\hat{x}_{k+1} = A_d\hat{x}_k + B_d u_k + L_d(y_{m,k} - (C_m\hat{x}_k + D_m u_k))$$

13.4.2 Observer-Based Feedback with Integral Control

As in Section 13.2, an integral feedback loop can be added as an outer loop on observer-based feedback controllers. Adding integral control can reduce the effect of modeling uncertainties and nonlinearities. Figure 13.9 shows an observer-based feedback controller with integral control. Integral feedback control requires both outputs y_m and y to be measured and available for feedback. Guidance for selecting the integral gain k_I is the same as in Eq. (13.11).

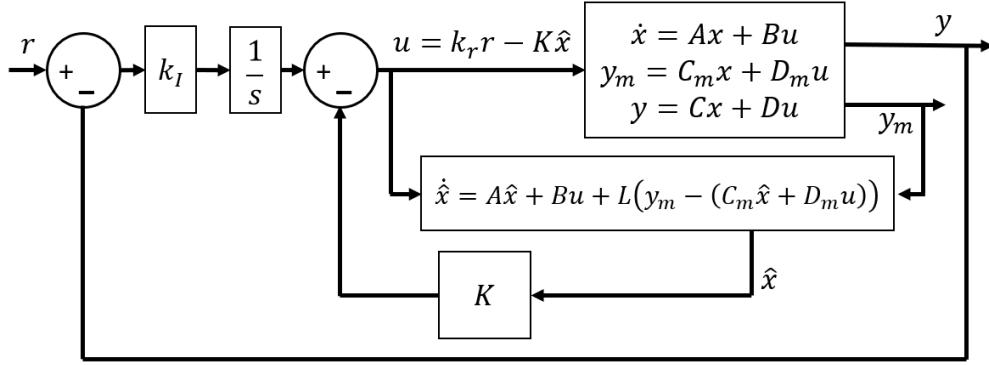


Figure 13.9 Block diagram of observer-based full-state feedback with integral control

13.4.3 Stability of Luenberger-Based Feedback Controllers

This section studies the stability of a system controlled by observer-based feedback. The states are estimated using a Luenberger observer, and the control $u = -K\hat{x} + k_r r$ is determined using the full-state feedback control algorithms derived in Section 13.1.

Before adding an observer to our feedback loop, the closed-loop system looked like the following:

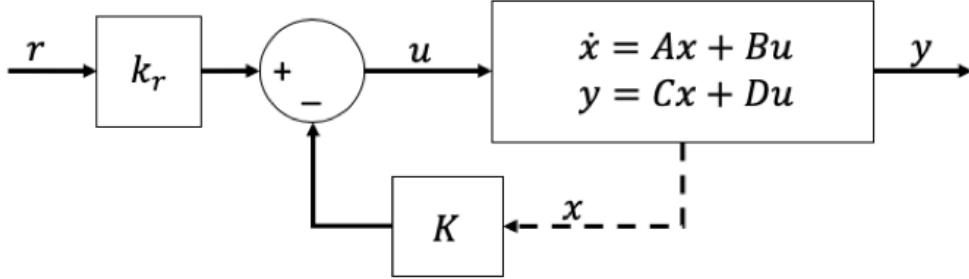


Figure 13.10 Full-state feedback control

From the above feedback loop, the control input u is calculated to be:

$$u = k_r r - Kx$$

where we assume we know all the state values in x . Using this in the state equation gives:

$$\begin{aligned}\dot{x} &= Ax + B(k_r r - Kx) \\ &= (A - BK)x + Bk_r r\end{aligned}$$

and we can see how the feedback control gains K affect the placement of the closed-loop poles by evaluating:

$$\lambda = \text{eig}(A - BK)$$

13.4 Observer-Based Feedback Control

i.e., by solving for the roots of:

$$\det(\lambda I - (A - BK)) = 0$$

When using an observer to estimate the unknown or unmeasurable states in x then the feedback loop looks like the following figure:

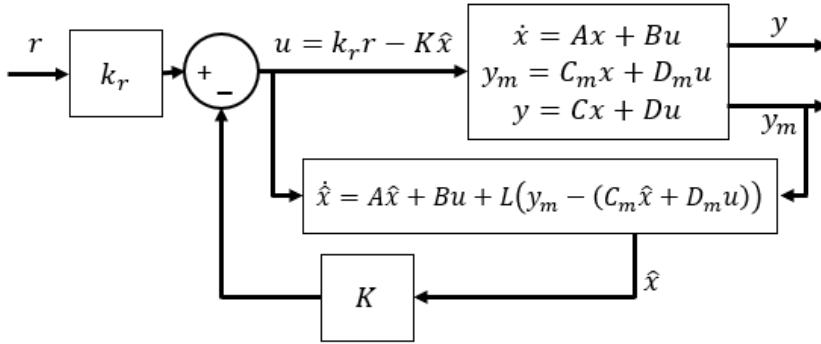


Figure 13.11 Full-state feedback control using a Luenberger observer.

From the above feedback loop, the control input u is calculated to be:

$$u = k_r r - K\hat{x}$$

where we use the estimated states from the observer (\hat{x}). Using this in the state equation gives:

$$\begin{aligned}\dot{x} &= Ax + B(k_r r - K\hat{x}) \\ &= Ax - BK\hat{x} + Bk_r r\end{aligned}$$

Recognizing that $y_m = C_m x + D_m u$, the the $D_m u$ terms in the Luenberger observer

$$\dot{\hat{x}} = A\hat{x} + Bu + L(C_m x + D_m u - (C_m \hat{x} + D_m u))$$

cancel each other. As a result, the Luenberger observer can theoretically be written as follows:

$$\dot{\hat{x}} = A\hat{x} + Bu + L(C_m x - C_m \hat{x})$$

Replacing the input with $u = k_r r - K\hat{x}$ gives:

$$\begin{aligned}\dot{\hat{x}} &= A\hat{x} + B(k_r r - K\hat{x}) + L(C_m x - C_m \hat{x}) \\ &= LC_m x + (A - BK - LC_m) \hat{x} + Bk_r r\end{aligned}$$

If we augment the state matrices to include both x and \hat{x} together such that:

$$X = \begin{bmatrix} x \\ \hat{x} \end{bmatrix}$$

then the total state equation is a combination of the above state equations to get:

$$\dot{X} = \begin{bmatrix} A & -BK \\ LC_m & (A - BK - LC_m) \end{bmatrix} X + \begin{bmatrix} Bk_r \\ Bk_r \end{bmatrix} r$$

If we want to calculate the poles or eigenvalues of the combined system then we do so by solving the following:

$$\det\left(\lambda_X I - \begin{bmatrix} A & -BK \\ LC_m & (A - BK - LC_m) \end{bmatrix}\right) = 0$$

where λ_X is a vector of $2n$ number of eigenvalues, and I is a $2n$ by $2n$ identity matrix. Evaluating the determinant of above equation gives us:

$$(\lambda_X I - A)(\lambda_X I - A + BK + LC_m) - (LC_m)(-BK) = 0$$

Expanding the above equation is quite messy, but results in a second order polynomial for the vector of eigenvalues λ_X :

$$\lambda_X^2 I - 2\lambda_X A - \lambda_X LC_m + A^2 - ALC_m + \lambda_X BK - ABK + BKLC_m = 0$$

Interestingly, the above expansion can also be collected into the form:

$$(\lambda_X I - (A - BK))(\lambda_X I - (A - LC_m)) = 0$$

which shows that the eigenvalues λ_X are a solution to both parenthetical terms:

$$\begin{aligned} \lambda_X I - (A - BK) &= 0 \\ \lambda_X I - (A - LC_m) &= 0 \end{aligned}$$

meaning the total eigenvalues are a combination of the eigenvalues of the controller λ_C and the eigenvalues of the observer λ_O :

$$\lambda_X = \begin{bmatrix} \lambda_C \\ \lambda_O \end{bmatrix}$$

The above eigenvalue solution is used to show what is known as the **separation principle**, which means that the poles of the controller can be determined and placed independently of the poles of the observer. Thus adding an observer should not affect the pole locations of the controller. As long as the observer is correctly estimating the values of the states in the controller, the controller can act like it knows all of the states as if they are all being measured. The separation principle is illustrated in the following example.

Separation Principle

Example 13.4.1. Separation Principle

13.4 Observer-Based Feedback Control

Consider the state-space system

$$\begin{aligned}\dot{x} &= \begin{bmatrix} 1 & 2 \\ 0 & 3 \end{bmatrix} x + \begin{bmatrix} 5 \\ 1 \end{bmatrix} u \\ y &= \begin{bmatrix} 1 & 0 \end{bmatrix} x\end{aligned}$$

If we want to place the closed-loop poles at:

$$p_1 = -2, \quad p_2 = -3$$

then we can calculate that we will need to use the following feedback control and feedforward gains:

$$\begin{aligned}K &= \begin{bmatrix} -1.5 & 16.5 \end{bmatrix} \\ k_r &= -6\end{aligned}$$

If we want to place the observer poles at:

$$o_1 = -20, \quad o_2 = -30$$

then we can calculate that we will need to use the following observer gains:

$$L = \begin{bmatrix} 54 \\ 379.5 \end{bmatrix}$$

Using these values, we can calculate the combined state equations from:

$$\dot{X} = \begin{bmatrix} A & -BK \\ LC & (A - BK - LC) \end{bmatrix} X + \begin{bmatrix} Bk_r \\ Bk_r \end{bmatrix} r$$

by evaluating each term individually:

$$\begin{aligned}-BK &= \begin{bmatrix} 5 \\ 1 \end{bmatrix} \begin{bmatrix} -1.5 & 16.5 \end{bmatrix} = \begin{bmatrix} 7.5 & -82.5 \\ 1.5 & -16.5 \end{bmatrix} \\ A - BK - LC &= \begin{bmatrix} 1 & 2 \\ 0 & 3 \end{bmatrix} - \begin{bmatrix} 5 \\ 1 \end{bmatrix} \begin{bmatrix} -1.5 & 16.5 \end{bmatrix} - \begin{bmatrix} 54 \\ 379.5 \end{bmatrix} \begin{bmatrix} 1 & 0 \end{bmatrix} = \begin{bmatrix} -45.5 & -80.5 \\ -378 & -13.5 \end{bmatrix} \\ LC &= \begin{bmatrix} 54 \\ 379.5 \end{bmatrix} \begin{bmatrix} 1 & 0 \end{bmatrix} = \begin{bmatrix} 54 & 0 \\ 379.5 & 0 \end{bmatrix} \\ Bk_r &= \begin{bmatrix} 5 \\ 1 \end{bmatrix} (-6) = \begin{bmatrix} -30 \\ -6 \end{bmatrix}\end{aligned}$$

and then combine them into the combined state equation:

$$\dot{X} = \begin{bmatrix} 1 & 2 \\ 0 & 3 \\ 54 & 0 \\ 379.5 & 0 \end{bmatrix} \begin{bmatrix} 7.5 & -82.5 \\ 1.5 & -16.5 \\ -45.5 & -80.5 \\ -378 & -13.5 \end{bmatrix} X + \begin{bmatrix} -30 \\ -6 \\ -30 \\ -6 \end{bmatrix} r$$

and simplified to get

$$\dot{X} = \begin{bmatrix} 1 & 2 & 7.5 & -82.5 \\ 0 & 3 & 1.5 & -16.5 \\ 54 & 0 & -45.5 & -80.5 \\ 379.5 & 0 & -378 & -13.5 \end{bmatrix} X + \begin{bmatrix} -30 \\ -6 \\ -30 \\ -6 \end{bmatrix} r$$

The eigenvalues of the combined system are calculated from:

$$\det \left(\lambda_X I - \begin{bmatrix} 1 & 2 & 7.5 & -82.5 \\ 0 & 3 & 1.5 & -16.5 \\ 54 & 0 & -45.5 & -80.5 \\ 379.5 & 0 & -378 & -13.5 \end{bmatrix} \right) = 0$$

and are calculated to be:

$$\lambda_X = [-2 \quad -3 \quad -20 \quad -30]^T$$

Thus, the presence of the observer did not change the location of the poles of the feedback controller.

13.5 Control Design using Root Locus

A **root locus plot** is a graph of how the closed-loop poles change as a result of varying a feedback control gain. Root locus tools help determine stability and system response by showing how controller gains affect closed-loop poles. Writing the characteristic equation (*i.e.*, by setting the denominator equal to zero) of a closed-loop transfer function in the following form is helpful for root locus design:

$$1 + kG_{rl}(s) = 0 \quad (13.17)$$

where G_{rl} is the root locus transfer function. As k increases from 0 to ∞ , the branches of the root locus plot begin at the poles of G_{rl} . A branch of the root locus ends at each of the zeros of G_{rl} as $k \rightarrow \infty$. If G_{rl} has more poles than zeros, some of the branches diverge along various paths in the complex plane. Consider the block diagram of Figure 13.12.

It has a controller gain k and a transfer function $G_1(s)$ in the forward path and a transfer function $G_2(s)$ in the feedback path. Therefore, its closed-loop transfer function from R to Y is

$$\frac{Y}{R} = \frac{kG_1(s)}{1 + kG_1(s)G_2(s)} \quad (13.18)$$

13.5 Control Design using Root Locus

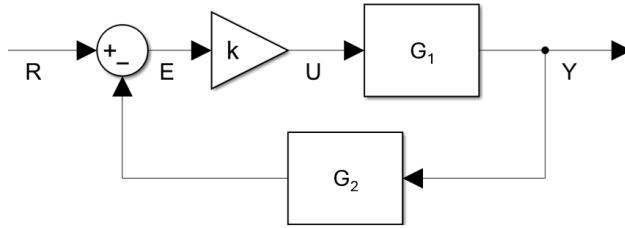


Figure 13.12 Block diagram of a feedback controller

The denominator $1 + kG_1(s)G_2(s)$ is the characteristic polynomial whose roots are the closed-loop poles of the system. The root locus transfer function is $G_{rl} = G_1(s)G_2(s)$. A root-locus plot is a graph of the closed-loop pole locations as the gain k is varied. The following example demonstrates one way to create a root locus plot using MATLAB to help select gains for a proportional-integral (PI) controller.

Selecting PI Gains using Root Locus Design

Example 13.5.1. Selecting PI gains using root locus design methods

Consider a closed-loop system modeled by the following block diagram:

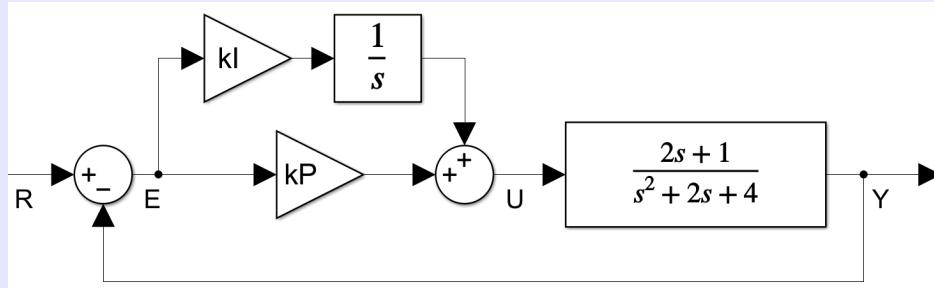


Figure 13.13 PI feedback controller

Demonstrate how root-locus plots can be used to help select the proportional k_P and integral k_I feedback gains.

Solution: The PI controller of Figure 13.13 has two gains, k_P and k_I . Root locus techniques can evaluate only one gain at a time. A common practice in tuning PID controllers is to begin by setting the integral k_I and derivative k_D gains to zero. Only the proportional gain k_P is adjusted at first. Once an acceptable proportional gain k_P is set, the next step is to adjust the integral gain k_I . The derivative gain k_D is found last. This example will follow those steps for the PI controller of this problem. With $k_I = 0$, the closed-loop transfer function is

$$\begin{aligned}\frac{Y}{R} &= \frac{k_P \frac{2s+1}{s^2+2s+4}}{1 + k_P \frac{2s+1}{s^2+2s+4}} \\ &= \frac{k_P (2s+1)}{s^2 + (2+2k_P)s + (4+k_P)}\end{aligned}$$

The closed-loop characteristic equation is therefore $s^2 + (2 + 2k_P)s + (4 + k_P) = 0$, and its roots are the poles of the closed-loop system. The scalar proportional gain k_P can be positive or negative, but this example will consider only positive values for k_P . The following MATLAB script can calculate and plot the poles as k_P is varied.

```

close all
clear
clc

kP = 0:0.001:100;
N = length(kP);
rootlocus = zeros(4,N); %2nd order polynomials have two roots for each kP
for ii = 1:N
    %Use the quadratic formula to find the roots of
    % s^2+(2+2*kP)s+(4+kP)=0
    s1 = (- (2+2*kP(ii))+sqrt((2+2*kP(ii))^2-4*(4+kP(ii))))/(2);
    s2 = (- (2+2*kP(ii))-sqrt((2+2*kP(ii))^2-4*(4+kP(ii))))/(2);
    %Get the real and imaginary parts
    r1 = real(s1);
    i1 = imag(s1);
    r2 = real(s2);
    i2 = imag(s2);
    %store the results
    rootlocus(1,ii) = r1;
    rootlocus(2,ii) = i1;
    rootlocus(3,ii) = r2;
    rootlocus(4,ii) = i2;
end

%Create a root locus plot
figure
plot(rootlocus(1,:),rootlocus(2,:),'b',...
    rootlocus(3,:),rootlocus(4,:),'b')
title('Root Locus Plot')
xlabel('Real')
ylabel('Imaginary')
grid on
xlim([-4,1])
ylim([-2,2])
%Include open-loop poles and zeros on the graph
hold on
plot(-1,sqrt(12)/2,'rx',...open-loop pole
    -1,-sqrt(12)/2,'rx',...open-loop pole
    -0.5,0,'ro'...open-loop zero
)

```

The result is shown in Figure 13.14. The poles of the open-loop transfer function

$$\frac{Y}{U} = \frac{2s+1}{s^2+2s+4}$$

are each marked with an x on the graph. The zero $s = -\frac{1}{2}$ is marked with an o. The root locus plot

13.5 Control Design using Root Locus

starts at the open-loop poles. The two branches of the plot converge when $k_P = -\frac{1}{2} + \frac{\sqrt{12}}{2} \approx 1.3$ at $s = -\frac{1}{2} - \frac{\sqrt{12}}{2} \approx -2.3$. As $k_P \rightarrow \infty$, one branch of the root locus converges to the open loop zero at $s = -0.5$, the other branch diverges towards $-\infty$. The complex-valued portion of the root locus appears to lie on a portion of a circle whose center is the open-loop zero.

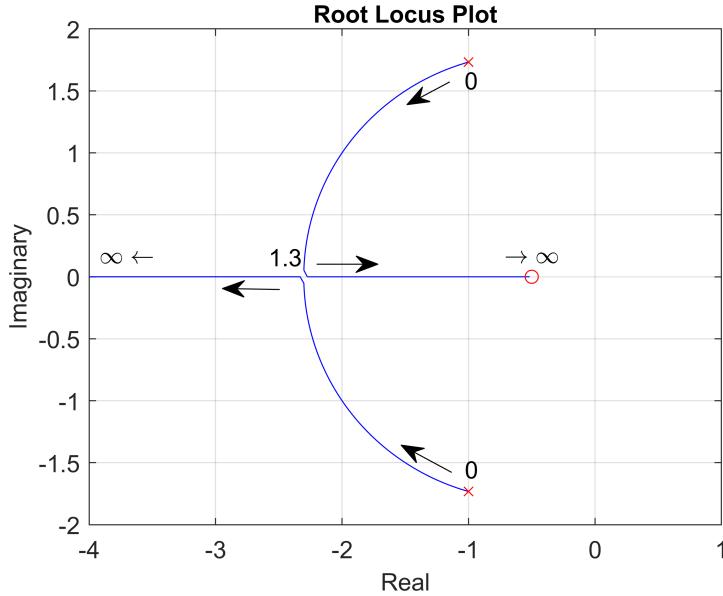


Figure 13.14 Root-locus plot for k_P with $k_I = 0$

The root locus plot shows that a value of $k_P \approx 1.3$ results in two stable, real-valued, closed-loop poles at about -2.3 . Because there is no imaginary part, the system response to a step input would not experience overshoot or oscillations. Smaller values, *i.e.*, $k_P < 1.3$, would cause the closed loop system to have overshoot and oscillations. Larger values of k_P would cause the system response to be slower because one closed-loop pole would get closer to the imaginary axis as it converges to the open-loop zero at $s = -0.5$. If k_P was the only controller gain, it may be optimal to select $k_P = 1.3$ to obtain the fastest response with no overshoot. However, we must also consider the integral gain. Integral control often induces oscillations, so we may want to select a larger value of k_P to decrease the risk of oscillations when the integral control is added. To avoid overshoot and oscillations, we will select a value of $k_P = 2$.

The next step is to determine the effect of the integral gain k_I on the closed-loop poles. Adding the integral gain, the closed-loop transfer function for the block diagram of Figure 13.13 is

$$\begin{aligned}\frac{Y}{R} &= \frac{\left(k_P + \frac{k_I}{s}\right)\left(\frac{2s+1}{s^2+2s+4}\right)}{1 + \left(k_P + \frac{k_I}{s}\right)\left(\frac{2s+1}{s^2+2s+4}\right)} \\ &= \frac{2k_P s^2 + (2k_I + k_P)s + k_I}{s^3 + (2 + 2k_P)s^2 + (4 + k_P + 2k_I)s + k_I}\end{aligned}$$

The characteristic equation for the closed-loop system is therefore $s^3 + (2 + 2k_P)s^2 + (4 + k_P + 2k_I)s + k_I = 0$. The root locus form $1 + kG_{rl}(s) = 0$ of this characteristic equation can be found by factoring out k_I :

$$s^3 + (2 + 2k_P)s^2 + (4 + k_P)s + k_I(2s + 1) = 0$$

$$1 + k_I \frac{2s + 1}{s^3 + (2 + 2k_P)s^2 + (4 + k_P)s} = 0$$

With $k_P = 2$, G_{rl} has one zero at $s = -0.5$. It has poles at $s = 0$, $s \approx -4.73$, and $s \approx -1.27$. The root locus plot will start at each G_{rl} pole, and one branch will end at its zero. At $k_I = 1.8065$, the other two branches converge on the real axis at $s = -2.892$ then diverge towards $\pm\infty$ along opposite paths in the complex plane (see Figure 13.15). With $k_P = 2$, this suggests that the fastest response without overshoot occurs when $k_I \approx 1.8065$. The corresponding closed-loop poles are $s_1 = s_2 = -2.892$, and $s_3 = -0.216$. The following MATLAB code creates the root locus plot as k_I is varied.

```

close all
clear
clc

kP = 2;
kI = 0:0.001:10;
N = length(kI);
rootlocus = zeros(6,N); %3rd order polynomials have two roots for each kI
for ii = 1:N
    %Find the roots of
    % s^3+(2+2*kP)s^2+(4+kP+2*kI)s+kI=0
    z = roots([1,2+2*kP,4+kP+2*kI(ii),kI(ii)]);
    %Get the real and imaginary parts
    rootlocus(1,ii) = real(z(1));
    rootlocus(2,ii) = imag(z(1));
    rootlocus(3,ii) = real(z(2));
    rootlocus(4,ii) = imag(z(2));
    rootlocus(5,ii) = real(z(3));
    rootlocus(6,ii) = imag(z(3));
end

%Create a root locus plot
figure
plot(rootlocus(1,:),rootlocus(2,:),'b.',...
      rootlocus(3,:),rootlocus(4,:),'b.',...
      rootlocus(5,:),rootlocus(6,:),'b.')
title('Root Locus Plot')
xlabel('Real')
ylabel('Imaginary')
grid on

%Include open-loop poles and zeros on the graph
hold on
plot(0,0,'rx',...
      -1.27,0,'rx',...
      -4.73,0,'rx',...
      -0.5,0,'rx',...
      -0.216,0,'rx')

```

13.5 Control Design using Root Locus

```

-4.73,0,'rx',...
-0.5,0,'ro',...
)

```

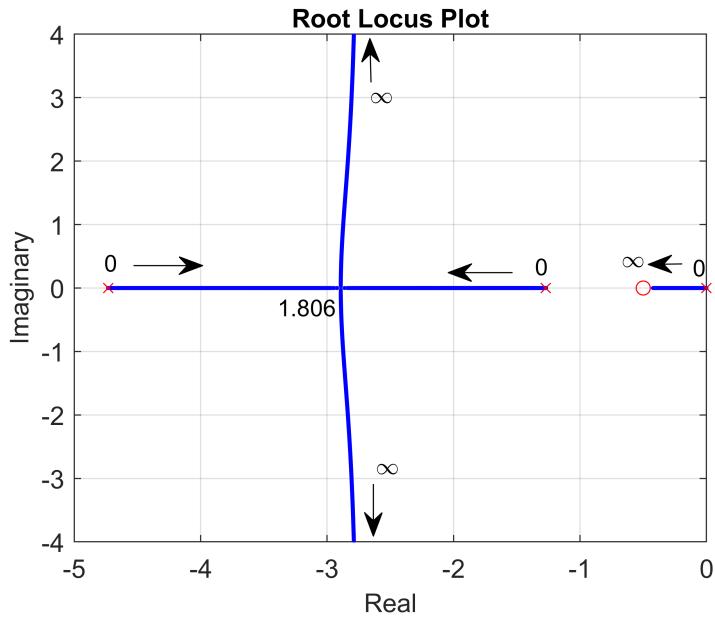


Figure 13.15 Root-locus plot for k_I with $k_P = 2$

Figure 13.16 shows the response of the system of Figure 13.13 to a unit step input. Although there is no overshoot, because of the pole at $s_3 = -0.216$, the transient rise time is $4/0.216 \approx 19$ s.

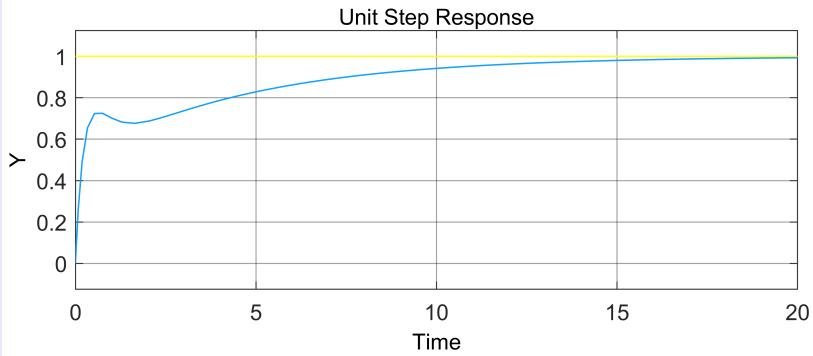


Figure 13.16 Unit step response if $k_I = 1.8$ and $k_P = 2$

There are a variety of software tools to assist with creating root locus plots, Bode plots, step response plots and other control system graphs. MATLAB's Control System Designer facilitates selecting control gains by showing how they simultaneously affect the root locus plot, Bode plot (with phase and gain

margins), and unit step response. Documentation on the Control System Designer can be located on the Mathworks website <https://www.mathworks.com/help/control/ug/getting-started-with-the-control-system-designer.html> (accessed April 2024).

13.6 Control Design using Bode Plots

Bode plots were introduced in Section 2.8.2. They are a graphical depiction of a transfer function in the frequency domain. Observing how the Bode magnitude and phase graphs change as controller gains vary can help in control system design. In addition, understanding how gain and phase margins of open-loop systems are related to the stability and response of closed loop systems can help a control designer select feedback gains.

13.6.1 Phase and Gain Margins of Unity Feedback Systems

The discussion of phase and gain margins in this section applies to unity feedback systems. The difference between unity and non-unity feedback is shown in Figure 13.17. With **unity feedback**, there is no transfer function in the feedback path. A sound understanding of the theory of this section should facilitate extending the concepts to determine gain and phase margins of non-unity feedback systems.

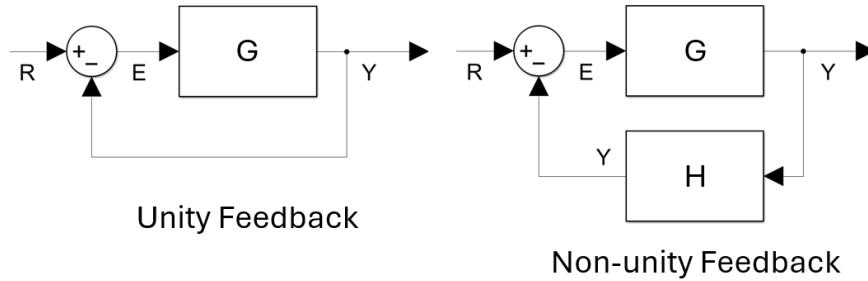


Figure 13.17 Unity versus non-unity feedback systems

The closed-loop transfer function of the unity feedback system of Figure 13.17 is

$$\frac{Y}{R} = \frac{G}{1+G}$$

Its Bode magnitude is

$$20\log_{10}(|G(j\omega)|) - 20\log_{10}(|1+G(j\omega)|) \quad (13.19)$$

Notice that the term $20\log_{10}(|G(j\omega)|)$ is the Bode magnitude of the open loop system. The second term is $-20\log_{10}(|1+G(j\omega)|)$. The closed loop system becomes unstable when

$$-20\log_{10}(|1+G(j\omega)|) \rightarrow \infty$$

which happens when

$$1+G(j\omega) \rightarrow 0$$

13.6 Control Design using Bode Plots

The term $1 + G(j\omega) \rightarrow 0$ when the magnitude $|G(j\omega)| = 1$ and the system output is 180° out of phase with the input; *i.e.*, the system reaches instability when the Bode magnitude is 0 dB and the Bode phase is $-180^\circ \pm n360^\circ$. Phase and gain margins measure how far the system is from the unstable point of 0 dB and $-180^\circ \pm n360^\circ$. Whether a system is stable or not is determined by closed-loop pole locations; for systems with open-loop unstable poles, gain and phase margins do not tell us whether a system is stable or not. However, gain and phase margins are an indication of the stability robustness or fragility. Smaller margins indicate a less stable system, while larger margins indicate a more stable system. If the open-loop system has no unstable poles, its closed-loop system is stable if and only if the phase and gain margins are both positive as determined by Eqs. (13.20) and (13.22).

13.6.2 Phase Margins on Bode Plots

If the Bode magnitude plot crosses 0 dB, the Bode plot will have a phase margin. If the Bode magnitude plot crosses 0 dB at multiple locations, there will be multiple corresponding phase margins. If the Bode magnitude never crosses 0 dB, the phase margin is infinite. The frequency ω_{gc} at which the Bode magnitude crosses 0 dB is called the **gain crossover frequency**. The **phase margin** Φ_M is the gap (in degrees or radians) that the Bode phase $\angle G(j\omega_{gc})$ at the gain crossover frequency ω_{gc} is above -180° :

$$\Phi_M = \angle G(j\omega_{gc}) + 180^\circ \quad (13.20)$$

or sometimes

$$\Phi_M = \min_n |\angle G(j\omega_{gc}) + (180^\circ \pm n360^\circ)|, \text{ where } n = 0, 1, 2, \dots \quad (13.21)$$

where $\angle G(j\omega_{gc})$ is the bode phase angle at the gain crossover frequency. Therefore, phase margins can be identified by visually inspecting a Bode plot of the *open-loop* transfer function G . First, identify the gain crossover frequency. Then measure the smallest gap between the phase plot and $-180^\circ \pm n360^\circ$. The gap is the phase margin. If the phase angle at the gain crossover frequency is above $-180^\circ \pm n360^\circ$, then the phase margin is positive. If it is below $-180^\circ \pm n360^\circ$, the phase margin is negative. Example 13.6.1 demonstrates how to identify phase margins on a Bode plot.

13.6.3 Gain Margins on Bode Plots

If the Bode phase plot crosses $-180^\circ \pm n360^\circ$, the Bode plot will have a gain margin. If it crosses at multiple locations, there will be multiple corresponding gain margins. If the Bode phase never crosses $-180^\circ \pm n360^\circ$, the gain margin is infinite. The frequency ω_{pc} at which the Bode phase crosses $-180^\circ \pm n360^\circ$ is called the **phase crossover frequency**. The **gain margin** G_M is the gap (in dB) between 0 dB and the Bode gain $20 \log_{10}(|G(j\omega_{pc})|)$ at the phase crossover frequency ω_{pc} .

$$G_M = 0 - 20 \log_{10}(|G(j\omega_{pc})|) \quad (13.22)$$

Therefore, gain margins can be identified by visually inspecting a Bode plot of the *open-loop* transfer function G . First, identify the phase crossover frequency. Then measure the gap between 0 dB and the Bode magnitude at that frequency. The gap is the gain margin. Example 13.6.1 demonstrates how to identify phase and gain margins on a Bode plot.

Phase and Gain Margins

Example 13.6.1. Identify gain and phase margins on a Bode plot

A Bode plot of the open-loop transfer function

$$\frac{Y}{U} = \frac{1.5(s - 4)}{(s^3 + 6s^2 + 3s + 5)(s + 2)} \quad (13.23)$$

is shown in the following figure:

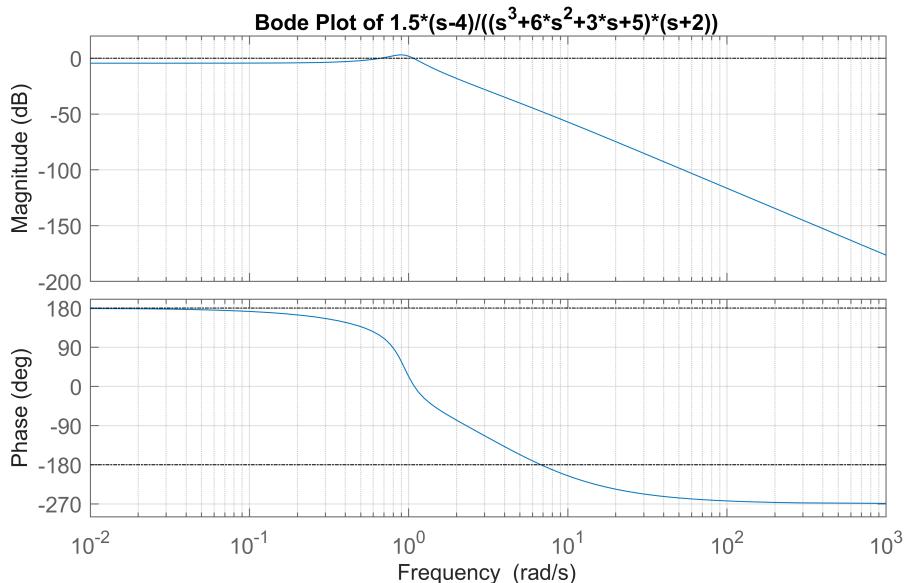


Figure 13.18 Determine the phase and gain margins

Identify the phase and gain margins for the closed-loop system. Assume unity feedback. Also determine whether the closed-loop system is stable or not.

Solution: The Bode magnitude of the open-loop system crosses 0 dB at two frequencies: 0.667 rad/s and 1.97 rad/s. These are the gain crossover frequencies. At the first gain crossover frequency, the Bode phase is 115.9°. The gap between 115.9° and 180° is 64.1°. The gap between 115.9° and -180° is 295.9°. Since 64.1° is less than 295.9°, the phase margin at the 0.667 rad/s crossover frequency is 64.1°. At the second gain crossover frequency, 1.97 rad/s, the Bode phase is 3°. The gap between 180° and 3° is 177°. Therefore, the phase margin at the 1.97 rad/s crossover frequency is 177°.

The Bode phase plot of the open-loop system crosses -180° at a frequency of 6.77 rad/s. Therefore, the phase crossover frequency is 6.77 rad/s. The Bode magnitude at the phase crossover frequency is -47.5 dB. The gap between 0 dB and the Bode magnitude is 47.5 dB. Therefore, the gain margin is 47.5 dB. The gain and phase margins are shown in Figure 13.19.

13.6 Control Design using Bode Plots

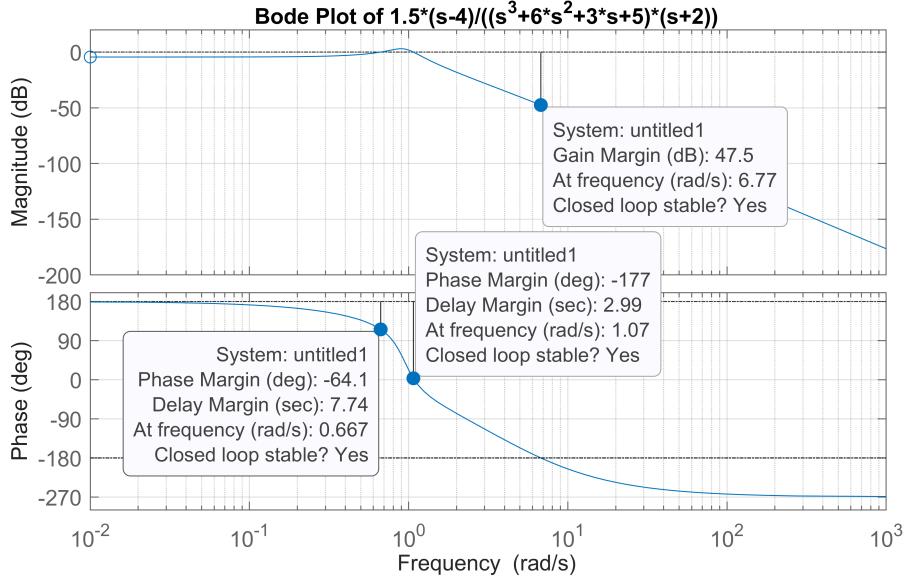


Figure 13.19 Phase and gain margins solution

Gain and phase margins do not necessarily determine closed-loop stability, unless it is known that the open loop system is stable. Assuming unity feedback, the closed-loop transfer function is

$$\frac{G}{1+G}$$

where G is the transfer function of the open loop system. The poles of the closed loop system are the roots of the characteristic polynomial $1 + G$, which is the denominator of the closed-loop transfer function. From Eq. (13.23), the characteristic polynomial is

$$\begin{aligned} 0 &= 1 + \frac{1.5(s-4)}{(s^3+6s^2+3s+5)(s+2)} \\ &= (s^3+6s^2+3s+5)(s+2) + 1.5(s-4) \\ &= s^4 + 8s^3 + 15s^2 + 12.5s + 4 \end{aligned}$$

which has roots at $-5.75, -0.75 \pm 0.602j$, and -0.752 . These roots are the poles of the closed-loop system. Since all poles are on the left half of the complex plane, they are all stable poles. The closed-loop system is stable.

The Bode plots in this example were generated using MATLAB's Control System Toolbox. The Control System Toolbox includes tools for creating Bode plots and displaying gain and phase margins. More information is provided at the following website: <https://www.mathworks.com/help/control/ug/assessing-gain-and-phase-margins.html> (accessed April 2024).

Bode plots are valuable to designing feedback controllers. This section introduced phase and gain

margins. There are many other uses of Bode plots in control system design, some of which will be introduced in the discussion of compensators later in this chapter. Loop shaping is a controller design technique using Bode plots that directly manipulates the frequency response characteristics to achieve design requirements. Design requirements include Bode phase and gain margins; they may also include response time and other time-domain requirements such as those presented in Section 13.7. The design of compensators for loop shaping is discussed later in this chapter.

13.7 Control Design Requirements

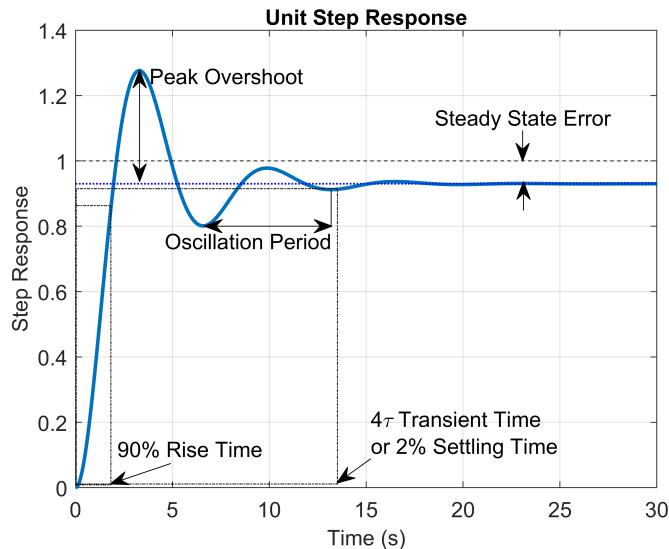


Figure 13.20 Common metrics for controller design shown on a unit step response

Specifications in the time and frequency domains can help communicate requirements for designing autonomous controllers. Requirements may be specified for response times, stability robustness, or performance metrics. Figure 13.20 shows common time domain metrics for a system's response to a unit step input. Some of these metrics include the 90% rise time (or 10-90% rise time), peak overshoot, oscillation period, 2% settling time (or 4τ transient time, see Section 1.6.1), and steady-state error. Stability robustness metrics can be specified in the frequency domain by open-loop phase and gain margins on a Bode plot, see Section 13.6.1. The oscillation period and peak overshoot on a unit step response can also be interpreted as stability robustness metrics. Less overshoot with fewer oscillations can be an indication of better stability robustness.

13.8 Designing Compensators Using Root Locus and Bode Plots

A compensator is a subsystem of a feedback controller that modifies the dynamics of the open-loop system. For example, the PID subsystem is the compensator in a PID feedback controller, see Figure 13.21. A feedback controller is the entire system, which includes the compensator, plant, feedback loop, sensors,

13.8 Designing Compensators Using Root Locus and Bode Plots

and actuators. The PID compensator has the ability to change the dynamic response of the open-loop system, which includes the compensator and the plant.

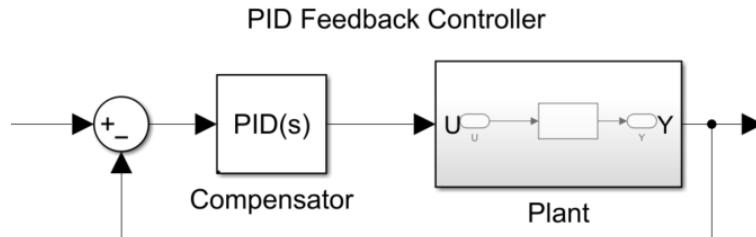


Figure 13.21 A PID compensator in a PID feedback control architecture

This section discusses the aspects and functions of compensators. It illustrates the effects of adding poles and zeros to a compensator and visualizes their effects on Bode and root locus plots. It introduces lead, lag, and lead-lag compensation and shows how to implement compensators with digital microcontrollers and analog circuits. Finally, it reviews PID compensation from the perspective of how it affects Bode and root locus plots.

13.8.1 Compensators

¹In the context of feedback control systems, a compensator is a component or subsystem designed to adjust the response characteristics of the system. Its primary purpose is to improve the overall performance of the control system by shaping the system's frequency response, transient response, stability, or robustness.

Here are some key aspects and functions of a compensator:

1. **Frequency Response Adjustment:** Compensators can alter the gain and phase characteristics of the system across different frequencies. This helps in achieving desired bandwidth, stability margins, and frequency-domain specifications.
2. **Transient Response Improvement:** By introducing appropriate poles and zeros, compensators can enhance the system's transient response, making it faster or damping oscillations.
3. **Stability Enhancement:** Compensators can improve the stability of the closed-loop system by adjusting the location of poles and zeros in the open-loop transfer function.
4. **Robustness to Parameter Variations:** Some compensators are designed to provide robust performance in the face of uncertainties and variations in system parameters, ensuring reliable operation over a range of conditions.

Types of compensators commonly used include proportional-integral-derivative (PID) controllers, lead-lag compensators, phase-lead compensators, phase-lag compensators, and others. Each type has specific characteristics and is chosen based on the control objectives and the dynamics of the system being controlled.

Overall, compensators play a crucial role in ensuring that the feedback control system meets its design requirements, such as response speed, stability, accuracy, and robustness to disturbances.

¹This subsection was written by ChatGPT 3.5 in response to the request: Define compensator in feedback controllers

13.8.2 Adding Poles to Compensators

A real-valued pole, $s = -p$ (rad/s), can be added to a compensator. Adding a pole is like adding a low-pass filter to the open-loop dynamics. It dampens higher frequencies and adds a phase lag, or negative phase-shift, at frequencies near or higher than p , see Figure 13.22.a. The transfer function $C(s)$ for a compensator with a gain k and a pole at $s = -p$ has the following form:

$$C(s) = k \frac{p}{s + p} \quad (13.24)$$

Figure 13.22.b shows a root locus plot of the pole assignment compensator. As the gain k increases from $k = 0$ to $k \rightarrow \infty$, the root locus begins at the pole at $s = -p$ and diverges to $s \rightarrow -\infty$ along the real axis. Mainly, however, the compensator adds a real pole at $s = -p$ to the open-loop dynamics of the system.

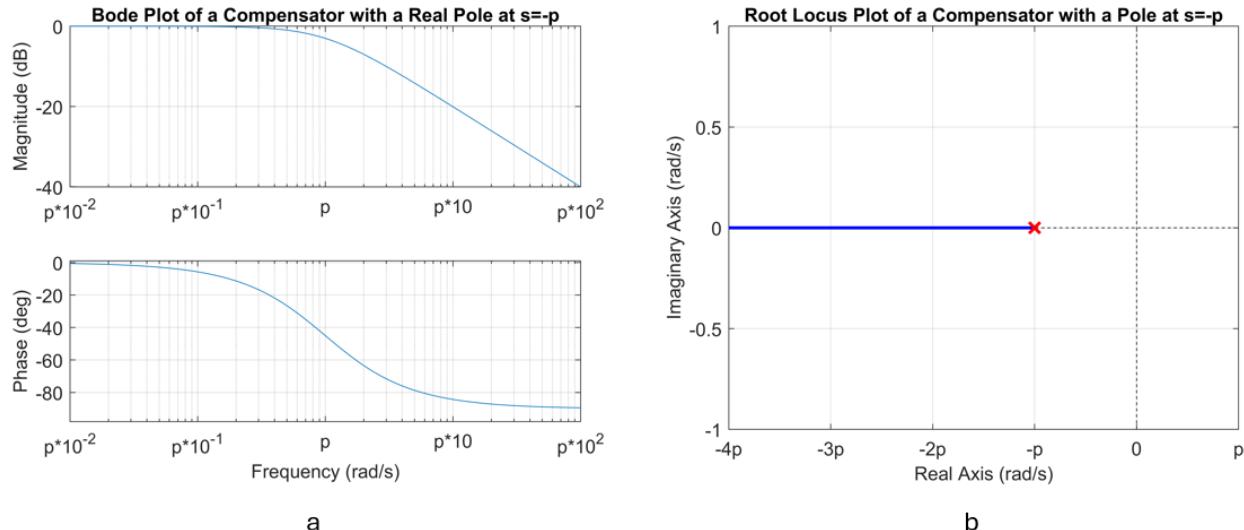


Figure 13.22 Bode and root locus plots of a compensator $C(s)$ with a real pole at $s = -p$

Adding a pole $s = -p$ at or near zero can decrease the steady-state error in the closed-loop response of a system. Another important application for adding a pole to a compensator is to improve the robustness and time response of a system with a resonance peak. This is shown in the following example:

Pole Assignment Compensation

Example 13.8.1. Compensator Design for a System with a Resonance Peak

Design a compensator to improve the gain margin, phase margin, and closed loop step response of a system with the following open-loop transfer function:

$$\frac{Y}{U} = \frac{-5s + 100}{s^3 + 2s^2 + 100s}$$

Use the feedback control architecture shown in Figure 13.21.

Solution: This example used MATLAB's Control System Designer to generate the Bode plot, root locus plot, and closed-loop step response plot by typing the following command in MATLAB's

13.8 Designing Compensators Using Root Locus and Bode Plots

command window:

```
controlSystemDesigner(tf([-5,100],[1,2,100,0]))
```

The Control System Designer shows that although the closed-loop step response is stable, and the phase margin is 85.9 deg, the gain margin is only 5.19 dB and the step response has many oscillations because of the resonance peak near 10 rad/s on the Bode Magnitude plot, see Figure 13.23. The goal is to improve the gain margin and dampen out oscillations in the closed-loop step response by adding a compensator with a pole and adjusting the compensator gain. Since adding a pole decreases the slope of the Bode magnitude, we will add a compensator with a pole at $s = -7$ rad/s, which is slightly before the resonance peak at $s = -10$ rad/s. We will also decrease the compensator gain from $k = 1$ to $k = 0.7$. To do so, we open the Compensator Editor in the Control System Designer, add a real pole at the location -7 , and change the compensator gain to 0.7. The resulting compensator transfer function is shown as

$$C = 0.7 \frac{1}{1 + 0.14s}$$

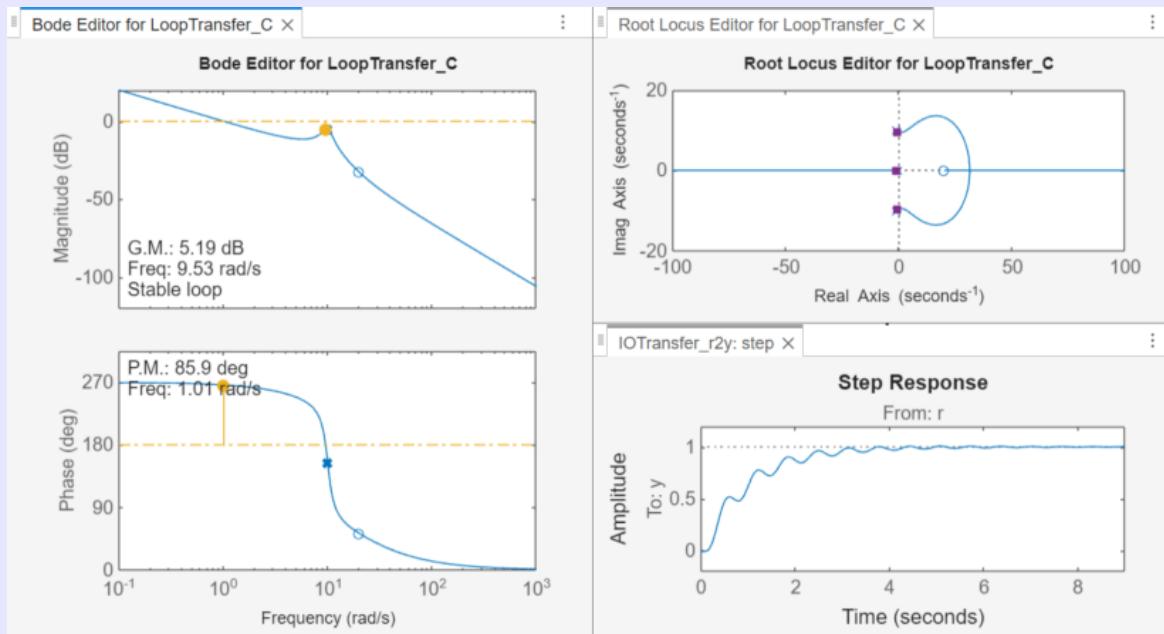


Figure 13.23 MATLAB's Control System Designer: `controlSystemDesigner(tf([-5,100],[1,2,100,0]))`

Figure 13.24 shows the improved response using the compensator. The gain margin was improved, which improves the stability robustness. It increased from 5.19 dB in the uncompensated system to 16.4 dB with compensation. The oscillations in the closed-loop step response were damped without negatively affecting other performance metrics such as rise-time, overshoot, steady-state response, or settling time. Only the phase margin was negatively affected. It decreased slightly from 85.9 to 81.5 deg. Adding a pole to the compensator adds phase lag. This phase lag caused the slight decrease in the phase margin.

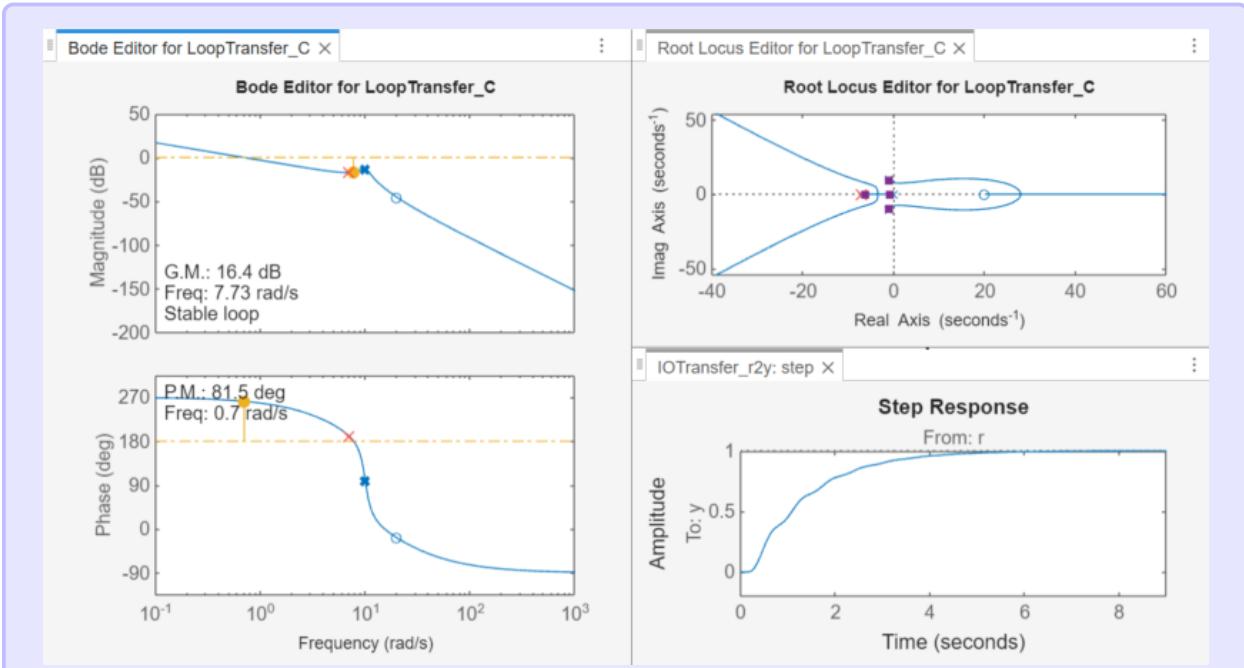


Figure 13.24 The response is improved by adding the compensator with a pole at $s = -7$ and gain $k = 0.7$.

Poles can also be added to a compensator in complex conjugate pairs. The effect is similar to adding a real pole. The transfer function of the compensator is

$$C(s) = k \frac{\omega_n^2}{s^2 + 2\zeta\omega_n s + \omega_n^2} \quad (13.25)$$

where the damping ratio ζ is less than one: $0 < \zeta < 1$.

13.8.3 Lead and Lag Compensators

Lead and lag compensators are commonly used alternatives to PID compensation. Lead compensators can perform some of the functions of PD compensation. Lag compensators can perform functions similar to PI compensators. Lead and lag compensators can be combined to behave like a PID compensator. Lead and lag compensators both have the following transfer function:

$$u = k \frac{s+z}{s+p} e \quad (13.26)$$

where $s = -z$ is the zero and $s = -p$ is the pole of the transfer function. If $z < p$, the transfer function represents a lead (or phase-lead) compensator. If $z > p$, the transfer function is for a lag (or phase-lag) compensator. To understand how a lag compensator can perform like a PI compensator, consider the limit as $p \rightarrow 0$. In the limit as $p \rightarrow 0$, Eq. (13.26) is equivalent to a PI compensator with the proportional gain $k_p = k$ and the integral gain $k_I = kz$. To understand how a lead compensator can behave like a PD controller, consider the limiting case as $p \rightarrow \infty$. The lead compensator can be written as

13.8 Designing Compensators Using Root Locus and Bode Plots

$$u = \frac{\frac{k}{p}s + \frac{k}{p}z}{\frac{1}{p}s + 1} e \quad (13.27)$$

Therefore, as $p \rightarrow \infty$, $u = \left(\frac{k}{p}s + \frac{k}{p}z\right)e$, which is a PD controller with proportional gain $k_P = \frac{k}{p}z$ and derivative gain $k_D = \frac{k}{p}$. The fact that lead and lag compensators can behave like PI, PD, PID and other compensators makes them a more general form of compensation.

Figure 13.25 shows a Bode plot of lead versus lag compensators with poles or zeros at -0.1 and -10 . The phase angle plot shows that a lead compensator creates a temporary increase in the phase angle, *i.e.*, a phase lead. This increased phase can improve stability robustness of a system that has a poor phase margin. A lag compensator has a decrease in the phase angle, *i.e.*, a phase lag. The magnitude plot shows that the lag compensator increases the steady-state gain. This increase in steady-state gain can decrease closed-loop steady state error for some systems. The lead compensator decreases steady-state gain.

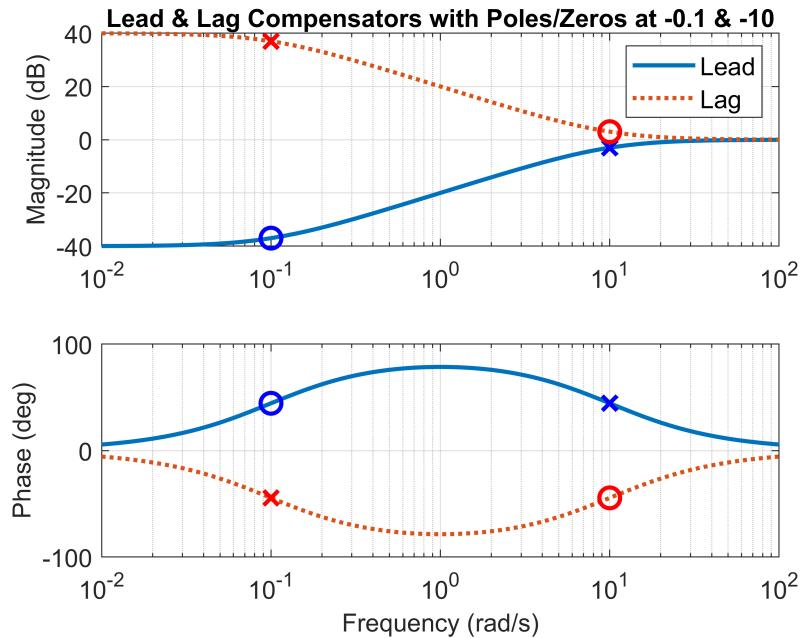


Figure 13.25 Bode plot of lead and lag compensators

Loop Shaping Controller Design

Example 13.8.2. Use Loop Shaping to Design a Compensator

*a*Use MATLAB's Control System Designer with loop shaping techniques to cause the following transfer function

$$G(s) = \frac{24s + 1}{s^3 + 4s^2 + 16s + 8}$$

to meet or exceed the following requirements:

- 10-90% rise time less than 5 s
- Transient time less than 8 s
- Peak overshoot less than 5%
- Zero steady-state error
- Gain margin greater than 12 dB
- Phase margin greater than 70 deg

Solution: The following command is used to open MATLAB's Control System Designer:

```
controlSystemDesigner(tf([24,1],[1,4,16,8]))
```

To satisfy the requirement of zero steady-state error, an integrator is added to the compensator. To add the integrator, we double-click "C" under the "Controllers and Fixed Blocks". When the Compensator Editor appears, we right-click the table under Dynamics and add an integrator. The Step Response graph shows that the output y eventually reaches the reference input r indicating that the steady-state error converges to zero. We right-click in the Step Response graph and click Characteristics and Rise Time. This places a dot on the graph at the 10-90% Rise Time. Clicking the dot shows that the Rise Time is 0.73 s. We also right-click the Step Response graph, click Characteristics, and add the Transient Time and Peak Response. The transient time of 78.5 s does not meet the requirement. Unfortunately, we cannot improve the transient time sufficiently by increasing the compensator gain. Increasing the compensator gain by dragging the Bode Magnitude upwards causes instability before the specification is met because of insufficient gain and phase margins. We can add a lag compensator to improve the stability margins. In the Compensator Editor, we right-click the table under Dynamics and add a Lag compensator. We click the Location box of the Lag compensator and change the Real Zero to -0.5 and the Real Pole to -0.03. Doing so removes some of the oscillations in the Step Response graph, increases the gain margin to 27.2, and increases the phase margin to 85.3. It also negatively affected the Rise Time, changing it to 13.7 s. The Transient Time is 50.5 s. Because of the increased gain and phase margins, we can drag the Bode Magnitude plot upward while watching the Rise Time, Transient Time, Gain Margin, and Phase Margin. When the compensator gain reaches 0.306, the Rise Time is 3.98 s, the Transient Time is 7.2 s, the Phase Margin is 85.4 deg, the Gain Margin is 16.5 dB, and there are no oscillations or overshoot. Therefore, all design requirements are met, and the resulting compensator is

$$C(s) = 0.306 \frac{s + 0.5}{s^2 + 0.03s}$$

Figure 13.26 shows the results of these adjustments in the Control System Designer.

13.8 Designing Compensators Using Root Locus and Bode Plots

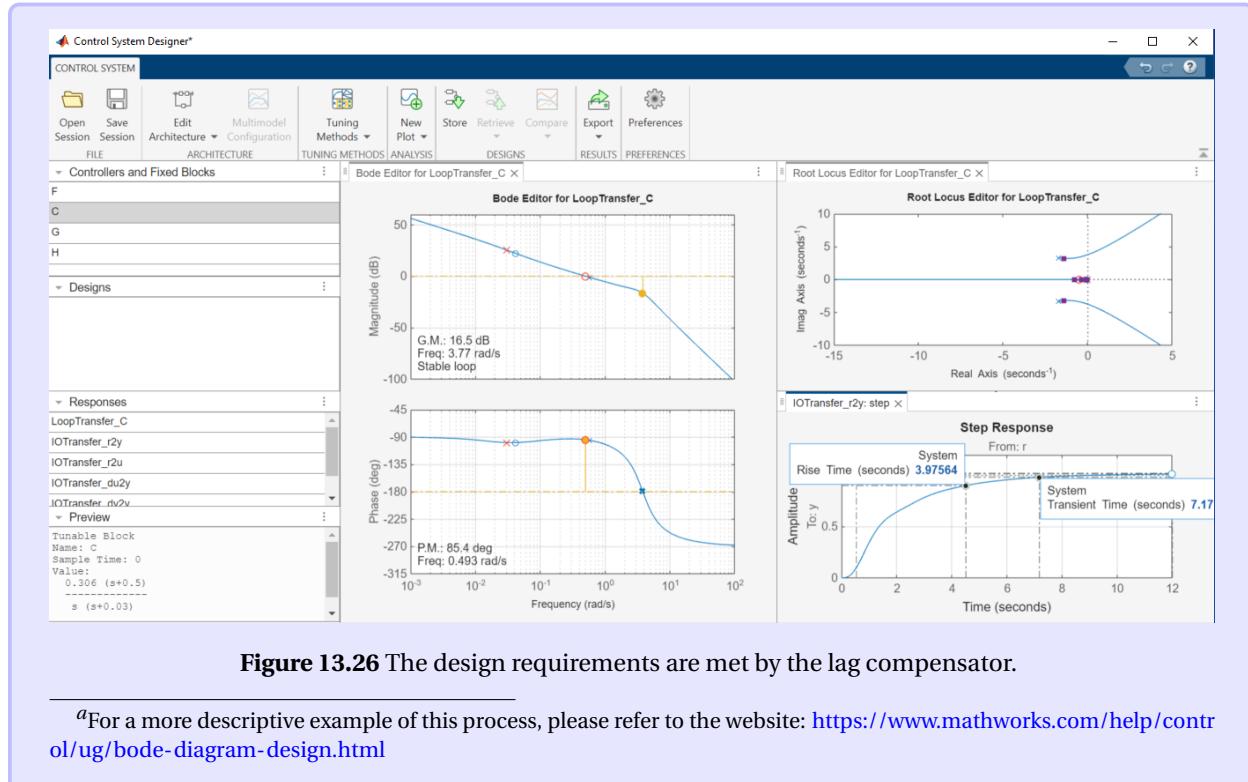


Figure 13.26 The design requirements are met by the lag compensator.

^aFor a more descriptive example of this process, please refer to the website: <https://www.mathworks.com/help/control/ug/bode-diagram-design.html>

13.8.4 Digital Implementation of Compensators

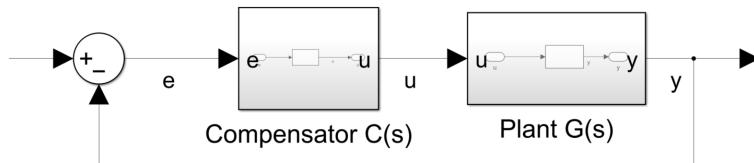


Figure 13.27 A possible control architecture with a compensator and a plant

The digital implementation, e.g., in a microcontroller, of a compensator depends on the structure of the controller. This section uses the control architecture shown in the block diagram of Figure 13.27. From this architecture, the transfer function from the error signal e to the control signal u is the compensation $C(s)$:

$$\frac{u}{e} = C(s) \quad (13.28)$$

Digital implementation requires converting Eq. (13.28) to discrete-time using a numerical method, such as the methods of Chapter 3. The following example demonstrate the digital implementation of a PID compensator.

Digital Implementation of PID Compensation

Example 13.8.3. Write code to numerically implement PID compensation

Write MATLAB code for the digital implementation of a PID controller using the backward Euler method of Table 3.1.

Solution: A PID compensator with the control architecture of 13.27 has the following transfer function:

$$u = k_P e + \frac{k_I}{s} e + k_D s e \quad (13.29)$$

where k_P is the proportional gain, k_I is the integral gain, and k_D is the derivative gain. $k_D s$ is not a proper transfer function because the order of the denominator (0) is less than the order of the numerator. To fix this problem we note that

$$se = \lim_{d \rightarrow 0} \frac{s}{ds + 1} e \quad (13.30)$$

$$= \lim_{d \rightarrow 0} \frac{1}{d} \frac{s}{s + 1/d} e \quad (13.31)$$

$$= \frac{1}{d} \left(\frac{s + 1/d - 1/d}{s + 1/d} \right) e \quad (13.32)$$

$$= \frac{1}{d} \left(e - \frac{1/d}{s + 1/d} e \right) \quad (13.33)$$

$$= \frac{1}{d} (e - e_d), \text{ where } e_d = \frac{1/d}{s + 1/d} e \quad (13.34)$$

To implement the derivative control compensation, therefore, we will replace s with $\frac{1}{d} \frac{s}{s+1/d}$ where d is a small number. Then we can convert Eq. (13.34) to discrete-time using the backward Euler method of Table 3.1:

$$e_{d,k+1} = \frac{1}{1 + \Delta t/d} e_{d,k} + \frac{\Delta t/d}{1 + \Delta t/d} e_k \quad (13.35)$$

If e_I is the integral of e , i.e., $e_I = \frac{1}{s} e$, then its backward Euler discrete-time equivalent is

$$e_{I,k+1} = e_{I,k} + \Delta t e_k \quad (13.36)$$

Therefore, the discrete-time equations for the digital implementation of the PID compensator are

$$u_k = k_P e_k + k_I e_{I,k} + \frac{k_D}{d} (e_k - e_{d,k}) \quad (13.37)$$

$$e_{d,k+1} = \frac{1}{1 + \Delta t/d} e_{d,k} + \frac{\Delta t/d}{1 + \Delta t/d} e_k \quad (13.38)$$

$$e_{I,k+1} = e_{I,k} + \Delta t e_k \quad (13.39)$$

The MATLAB implementation is provided below:

13.8 Designing Compensators Using Root Locus and Bode Plots

```

...
%Set initial conditions for ed and eI
ed = 0;
eI = 0;
%Set parameter values for the gains kP, kI, kD, and d
kP = ... %Proportional control gain
kI = ... %Integral control gain
kD = ... %Derivative control gain
d = ... %Small number for the derivative calculation
while (loop)
    ...
    %Get the time step
    dt = t_now - t_last;
    %Get the error signal, e, by subtracting the measured output y from the reference r
    e = y-r;
    %Calculate the control law u
    u = kP * e + kI * eI + kD/d * (e-ed);
    %Update ed
    ed = 1/(1+dt/d)*ed+(dt/d)/(1+dt/d)*e;
    %Update eI
    eI = eI + dt * e;
    %Perform the control action
    ...
end

```

The next example derives the discrete-time equations for the digital implementation of a lead or lag compensator.

Discrete Equations for a Lead or Lag Compensator

Example 13.8.4. Derive the discrete equations for a lead or lag compensator

Lead and lag compensators both have the following transfer function:

$$u = k \frac{s+z}{s+p} e \quad (13.40)$$

where $z < p$ indicates a lead filter, and $z > p$ indicates a lag filter. Use the backward Euler method of Table 3.1 to derive the discrete-time equations for a digital implementation of the filter.

Solution: A different way to write Eq. (13.40) is as follows

$$u = k \left(\frac{s}{s+p} + \frac{z}{p} \frac{p}{s+p} \right) e \quad (13.41)$$

$$= k \left(\frac{s+p-p}{s+p} + \frac{z}{p} \frac{p}{s+p} \right) e \quad (13.42)$$

$$= k \left(1 - \frac{p}{s+p} + \frac{z}{p} \frac{p}{s+p} \right) e \quad (13.43)$$

$$= k \left(e + \left(\frac{z}{p} - 1 \right) e_p \right), \text{ where } e_p = \frac{p}{s+p} e \quad (13.44)$$

The backward Euler solution to $e_p = \frac{p}{s+p} e$ is

$$e_{p,k+1} = \frac{1}{1 + \Delta tp} e_{p,k} + \frac{\Delta tp}{1 + \Delta tp} e_k \quad (13.45)$$

Therefore, the discrete-time equations for the digital implementation of the lead or lag compensator is

$$u_k = k e_k + k \left(\frac{z}{p} - 1 \right) e_{p,k} \quad (13.46)$$

$$e_{p,k+1} = \frac{1}{1 + \Delta tp} e_{p,k} + \frac{\Delta tp}{1 + \Delta tp} e_k \quad (13.47)$$

13.9 Successive Loop Closure Control Design

Successive loop closure is a control design process that applies to systems with nested feedback loops. An inverted pendulum is an example. A fast, inner feedback loop could be used to balance the robot in the vertical position. A second, middle loop might control the speed of the robot, and a third, slower outer loop might be used to control its position. Another example is a car. An inner loop might control engine or motor, and an outer loop could control the vehicle speed. This section will use airplane yaw control to teach the principles of successive loop closure design. An inner loop controls the yaw-rates, a middle loop controls the roll angle, and an outer loop controls the resulting yaw angle of the airplane.

A simplified model of roll rate dynamics is given by the following transfer function

$$\frac{\omega_x}{\delta_a} = \frac{b}{s+a} \quad (13.48)$$

where ω_x (rad/s) is the airplane's roll rate, δ_a (-1 to 1) is the aileron command, and a and b are constant parameters for the roll rate transfer function determined by system ID (see Chapter 12). If a gyrometer measures the roll rate ω_x , it can be used for feedback as shown in Figure 13.28. The signal $\omega_{x,d}$ is the desired or set-point roll rate, k_ω is the feedback gain, and δ_a is the aileron command.

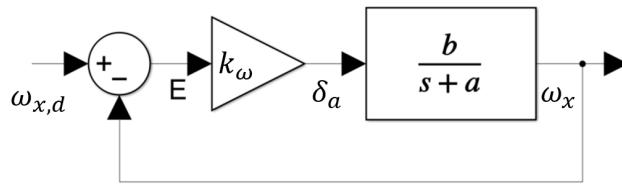


Figure 13.28 The roll-rate feedback loop

The roll angle ϕ_x (rad) is the time-integral of the roll rate ω_x . It can be estimated from other sensor measurements using state-estimation algorithms. Therefore it may also be available for feedback, and an outer loop can be added as shown in Figure 13.29. The signal $\phi_{x,d}$ is the desired or set-point roll angle, ϕ_x is the actual roll angle, and k_ϕ is the feedback control gain.

13.9 Successive Loop Closure Control Design

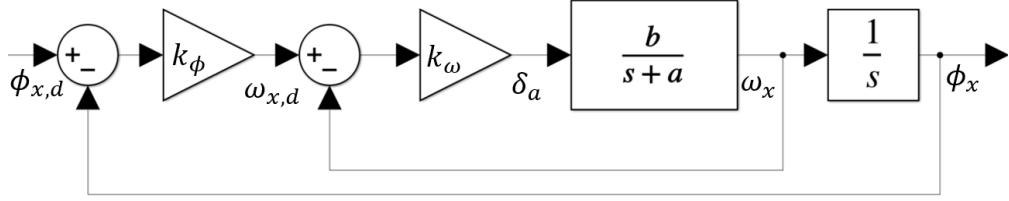


Figure 13.29 The roll angle ϕ_x control diagram

Finally, the yaw rate can be approximated as the scaled integral $\frac{c}{s}$ of the roll angle, where c is a constant scaling parameter. Like the roll angle, the yaw angle ψ_z can be estimated using sensor fusion techniques and made available for feedback. Therefore, an outer feedback loop can be added as shown in Figure 13.30. The signal $\psi_{z,d}$ is the desired or set-point yaw angle, ψ_z is the actual yaw angle, and k_ψ is the feedback gain for the outer loop.

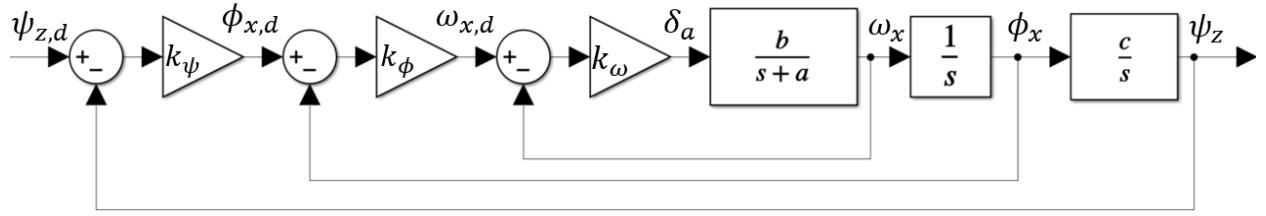


Figure 13.30 The yaw angle ψ_z control diagram

Successive loop closure is a control design technique that helps select the feedback gains k_ω , k_ϕ , and k_ψ . It begins by choosing controller gains for the innermost loop, *i.e.* k_ω . The selected gain results in a time-constant τ_1 for the innermost loop. If the time constant is at least 5 to 10 times faster than the successive output loops, *i.e.*, $\tau_2 \geq 5\tau_1$, then the transient response is fast enough that the inner loop can be approximated by a constant static gain g_1 . The transfer function for the innermost feedback loop of Figure 13.28 is

$$\frac{\omega_x}{\omega_{x,d}} = \frac{k_\omega \frac{b}{s+a}}{1 + k_\omega \frac{b}{s+a}} \quad (13.49)$$

$$= \frac{k_\omega b}{s + k_\omega b + a} \quad (13.50)$$

Since the root of the denominator is $s = -(k_\omega b + a)$, the time-constant τ_1 for the innermost loop is (see Eq. (1.44))

$$\tau_1 = \frac{1}{k_\omega b + a} \quad (13.51)$$

The **steady-state gain** of a transfer function is found by setting $s = 0$. The steady state gain g_1 of the inner loop transfer function Eq. (13.50) is

$$g_1 = \frac{k_\omega b}{0 + k_\omega b + a} \quad (13.52)$$

$$= \frac{k_\omega b}{k_\omega b + a} \quad (13.53)$$

We will select τ_1 to satisfy $\tau_2 \geq 5\tau_1$, therefore, the innermost loop can be replaced by the steady state gain g_1 , and the yaw control diagram is simplified as shown in 13.31.

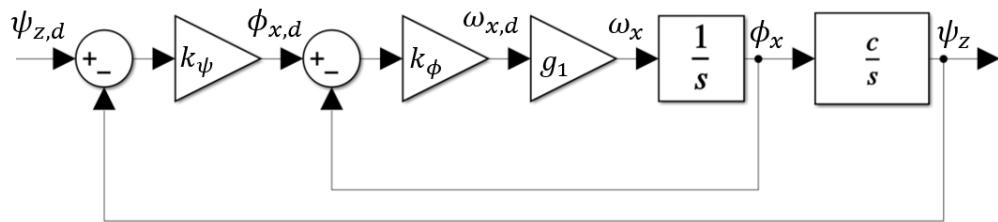


Figure 13.31 The yaw angle ψ_z control diagram with a simplified inner loop

Now, the inner loop in Figure 13.31 is the roll angle feedback control loop. Its closed loop transfer function is

$$\frac{\phi_x}{\phi_{x,d}} = \frac{\frac{k_\phi g_1}{s}}{1 + \frac{k_\phi g_1}{s}} \quad (13.54)$$

$$= \frac{k_\phi g_1}{s + k_\phi g_1} \quad (13.55)$$

which has a steady state gain of $g_2 = 1$ and a time constant of $\tau_2 = \frac{1}{k_\phi g_1} = \frac{k_\omega b + a}{k_\phi k_\omega b}$. If $\tau_3 \geq 5\tau_2$, the roll angle feedback loop can be replaced by its steady state gain $g_2 = 1$, and the yaw control diagram can be simplified even further as shown in Figure 13.32.

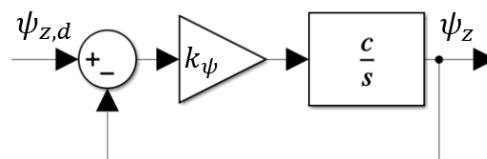


Figure 13.32 The yaw angle ψ_z control diagram simplified by successive loop closure

The transfer function for the simplified yaw control diagram of Figure 13.32 is

$$\frac{\psi_z}{\psi_{z,d}} = \frac{k_\psi c}{s + k_\psi c} \quad (13.56)$$

13.9 Successive Loop Closure Control Design

which has a time constant of $\tau_3 = \frac{1}{k_\psi c}$ and a steady state gain of $g_3 = 1$, which means that if the input $\psi_{z,d}$ is a step function, the output will eventually equal the input, *i.e.*, $\psi_z = \psi_{z,d}$ after about $4\tau_3$.

The successive loop closure analysis resulted in three different time constants, each corresponding to one of the feedback loops:

$$\begin{aligned}\tau_1 &= \frac{1}{k_\omega b + a} \\ \tau_2 &= \frac{k_\omega b + a}{k_\phi k_\omega b} \\ \tau_3 &= \frac{1}{k_\psi c}\end{aligned}$$

Successive loop closure requires that $\tau_3 \geq 5\tau_2 \geq 5\tau_1$. These time constants help determine the user-selected gains k_ω , k_ϕ , and k_ψ . For example, if we want the time constant of the roll angle to be $\tau_2 = 0.3$ s, then $\tau_3 = 5\tau_2 = 5\tau_1$ requires that $\tau_3 = 1.5$ s and $\tau_1 = 0.06$ s. Plugging these values back into the time-constants, we find that

$$\begin{aligned}k_\omega &= \frac{\frac{1}{0.06} - a}{b} \\ k_\psi &= \frac{1}{1.5c} \\ k_\phi &= \frac{k_\omega b + a}{0.3k_\omega b}\end{aligned}$$

To demonstrate the performance of the successive loop closure technique, this section will use the following values for the transfer function parameters a , b , and c :

$$a = 12$$

$$b = 34$$

$$c = 0.8$$

and the resulting gains are $k_\omega = 0.1373$, $k_\psi = 0.8333$, and $k_\phi = 11.9$. The Simulink model of Figure 13.33 is used to simulate the yaw dynamics with the successive loop closure feedback strategy. The resulting graph comparing the complete model of Figure 13.30 with the simplified model of Figure 13.32 is shown in Figure 13.34. Although the complete model has higher-order dynamics, its transient-time, step response, and steady-state final value using the successive loop closure gains are similar to those of the simplified model.

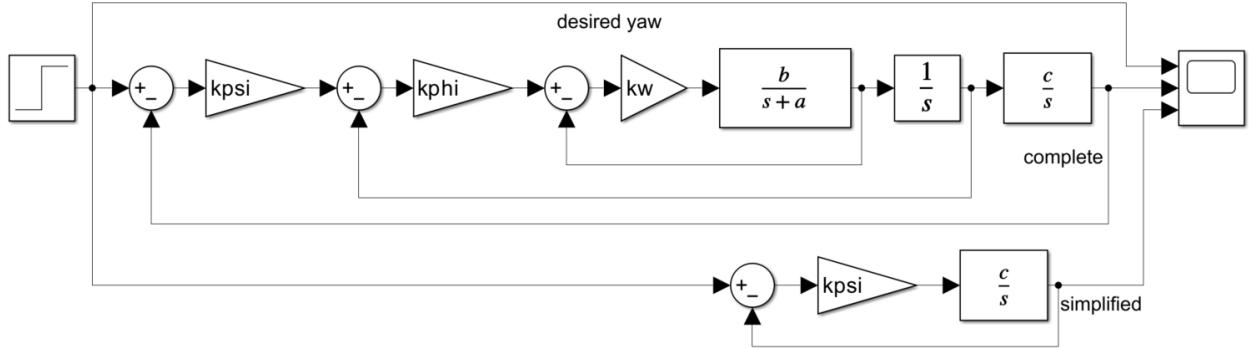


Figure 13.33 Simulink model to simulate the yaw feedback controllers

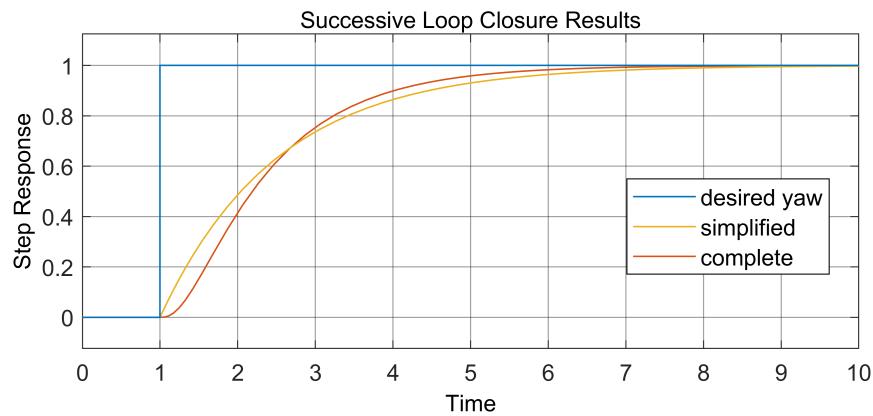


Figure 13.34 Results of the successive loop closure control design. The simplified model has a similar transient time and step response as the complete model by using successive loop closure to select the gains.

Chapter 14

Basic Autonomous Flight

Contents

14.1	Combining GPS with the Quaternion Estimation Algorithm	439
14.2	Roll, Pitch, and Yaw Euler Angles	441
14.2.1	Rotations Using Euler Angles	442
14.3	Waypoint Tracking and Path Planning	443
14.4	Autonomous Control of Roll, Pitch, and Yaw	445
14.4.1	Autonomous Flight Control	445
14.5	Constraints as Guides for Selecting Feedback Control Gains	448
14.6	Modifications for Flying Wing Airplanes	449

This chapter presents algorithms for basic autonomous flight. This basic strategy does not include takeoff or landing controls. It provides control for flying planes to navigate to target waypoints. The basic flight strategy combines a GPS displacement algorithm, a quaternion orientation algorithm, and proportional-derivative control. It uses the quaternion orientation estimator of Chapter 11 and a modified version of the GPS displacement algorithm of Section 8.7.

14.1 Combining GPS with the Quaternion Estimation Algorithm

Combining the GPS displacement algorithm of Section 8.7 with the quaternion estimation algorithm of Chapter 11 provides a way to estimate both the airplane's inertial displacement from a waypoint and its orientation. Knowing inertial displacement and orientation is critical to autonomous control. The computationally simplest way to determine inertial displacement and orientation is to solve for each separately, which is the topic of this section.

The following algorithm is used to calculate the inertial distances, $\Delta x_{I,k}$, $\Delta y_{I,k}$, and $\Delta z_{I,k}$ (m), between the airplane and a target waypoint.

GPS Waypoint Algorithm

The GPS waypoint algorithm calculates the inertial x, y, and z distances $\Delta x_{I,k}$, $\Delta y_{I,k}$, and $\Delta z_{I,k}$ (m) respectively between the airplane and a target waypoint. The latitude of the target waypoint is ϕ_T

(rad), the target longitude is λ_T (rad), and the target altitude it h_T (m). The GPS sensor measures the airplane's latitude ϕ_k (rad), its longitude λ_k (rad), and its altitude h_k (m) at each timestep t_k (s).

GPS latitude and longitude are usually provided in units of degrees (see Section 8.6). To use the following algorithm, latitude and longitude must be converted to have units of radians.

```

 $\Delta x_{I,k} = r(\phi_T - \phi_k)$ 
if ( $|\lambda_T - \lambda_k| \leq \pi$ )
     $\Delta\lambda = \lambda_T - \lambda_k$ 
else if ( $|\lambda_T - \lambda_k| > \pi$ )
    if ( $\lambda_T > \lambda_k$ )
         $\Delta\lambda = \lambda_T - (\lambda_k + 2\pi)$ 
    else
         $\Delta\lambda = (2\pi + \lambda_T) - \lambda_k$ 
    end
end
 $\Delta y_{I,k} = \text{sign}(\Delta\lambda) r \cos^{-1}((\cos(\Delta\lambda) - 1) \cos^2 \phi_T + 1)$ 
 $\Delta z_{I,k} = -(h_T - h_k)$ 

```

(14.1)

The earth's radius r (m) is approximately 6,371,000 m, but a more accurate local value can be used if necessary. The subscript k relates to the present iteration of the microcontroller.

The quaternion estimation algorithm was described in detail in Chapter 11. Figure 14.1 shows a block diagram of the combined GPS waypoint tracking algorithm and the quaternion estimation algorithm.

14.2 Roll, Pitch, and Yaw Euler Angles

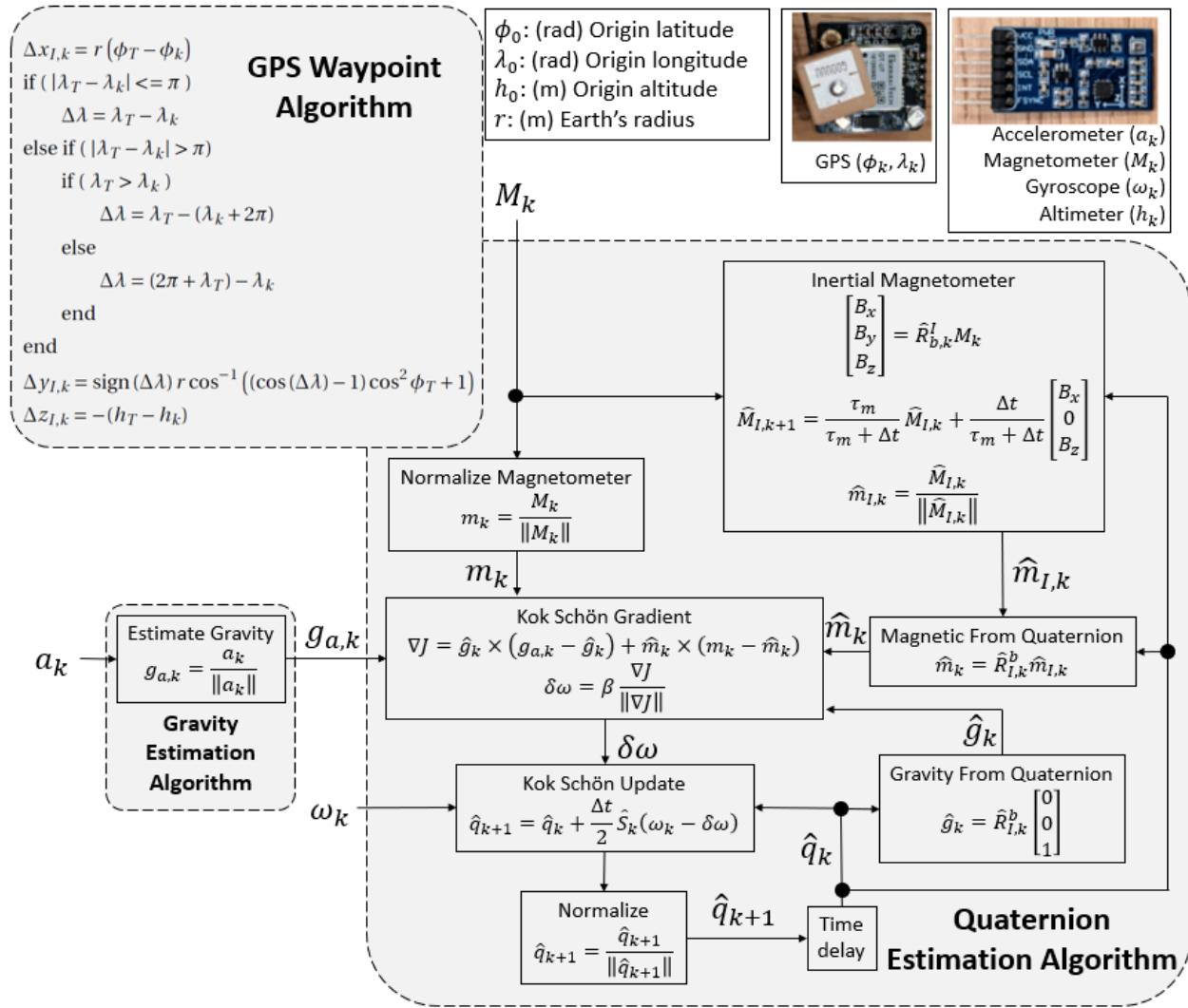


Figure 14.1 GPS waypoint tracking algorithm and the quaternion estimation algorithm.

14.2 Roll, Pitch, and Yaw Euler Angles

Roll (ϕ_x), pitch (θ_y), and yaw (ψ_z) angles can describe the orientation of the aircraft, see Figure 14.2. One way to describe roll, pitch, and yaw is in terms of Euler angles. Using Euler angles decouples roll, pitch, and yaw, making control simpler.

Euler angles depend on the order in which they are applied. The most intuitive sequence for aerospace applications is yaw – pitch – roll, see Figure 14.3. Yawing first means to rotate to the appropriate x-y heading angle, then pitch to the angle above or below the horizontal plane, then roll around the airplane's body-fixed x-axis. The derivations in this section use this sequence.

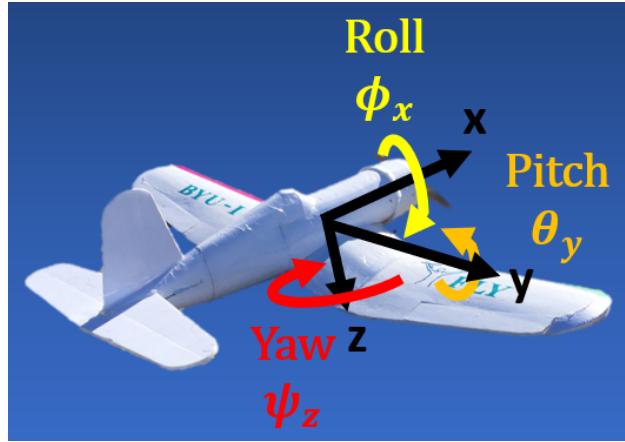


Figure 14.2 Roll, pitch, and yaw angles.



Figure 14.3 The yaw-pitch-roll Euler sequence. The airplane in the image demonstrates yaw -90° , then pitch 30° , and finally roll 45° .

14.2.1 Rotations Using Euler Angles

A yaw-rotation of ψ_z from one orientation in the inertial frame to a new orientation can be described mathematically using a rotation matrix. Conversion from the inertial frame (x_I, y_I, z_I) to the body frame (x, y, z) is calculated by the following equation:

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} \cos \psi_z & -\sin \psi_z & 0 \\ \sin \psi_z & \cos \psi_z & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_I \\ y_I \\ z_I \end{bmatrix} \quad (14.2)$$

If the plane undergoes a pitch rotation θ_y in addition to yaw ψ_z , the conversion from the inertial frame (x_I, y_I, z_I) to the body frame (x, y, z) is

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} \cos \theta_y & 0 & \sin \theta_y \\ 0 & 1 & 0 \\ -\sin \theta_y & 0 & \cos \theta_y \end{bmatrix} \begin{bmatrix} \cos \psi_z & -\sin \psi_z & 0 \\ \sin \psi_z & \cos \psi_z & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_I \\ y_I \\ z_I \end{bmatrix} \quad (14.3)$$

The order of the matrices from left to right is opposite of the sequence in which the rotations occur.

Finally, if a roll rotation of ϕ_x is applied after pitch and yaw, the conversion is

14.3 Waypoint Tracking and Path Planning

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\phi_x & -\sin\phi_x \\ 0 & \sin\phi_x & \cos\phi_x \end{bmatrix} \begin{bmatrix} \cos\theta_y & 0 & \sin\theta_y \\ 0 & 1 & 0 \\ -\sin\theta_y & 0 & \cos\theta_y \end{bmatrix} \begin{bmatrix} \cos\psi_z & -\sin\psi_z & 0 \\ \sin\psi_z & \cos\psi_z & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_I \\ y_I \\ z_I \end{bmatrix} \quad (14.4)$$

Combining the rotation matrices, Eq. (14.4) becomes

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} c\theta_y c\psi_z & -c\theta_y s\psi_z & s\theta_y \\ c\phi_x s\psi_z + s\phi_x s\theta_y c\psi_z & c\phi_x c\psi_z - s\phi_x s\theta_y s\psi_z & -s\phi_x c\theta_y \\ s\phi_x s\psi_z - c\phi_x s\theta_y c\psi_z & s\phi_x c\psi_z + c\phi_x s\theta_y s\psi_z & c\phi_x c\theta_y \end{bmatrix} \begin{bmatrix} x_I \\ y_I \\ z_I \end{bmatrix} \quad (14.5)$$

where c indicates the cosine and s indicates the sine. The equivalent quaternion rotation is

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} e_0^2 + e_1^2 - e_2^2 - e_3^2 & 2(e_0 e_3 + e_1 e_2) & 2(e_1 e_3 - e_0 e_2) \\ 2(e_1 e_2 - e_0 e_3) & e_0^2 - e_1^2 + e_2^2 - e_3^2 & 2(e_0 e_1 + e_2 e_3) \\ 2(e_0 e_2 + e_1 e_3) & 2(e_2 e_3 - e_0 e_1) & e_0^2 - e_1^2 - e_2^2 + e_3^2 \end{bmatrix} \begin{bmatrix} x_I \\ y_I \\ z_I \end{bmatrix} \quad (14.6)$$

Comparing Eqs. (14.5) and (14.6), we can derive conversions from quaternions to Euler angles and vice-versa. The equations that convert from quaternion components to Euler angles are

$$\phi_x = \text{atan2}(2(e_0 e_1 + e_2 e_3), e_0^2 - e_1^2 - e_2^2 + e_3^2) \quad (14.7)$$

$$\theta_y = \sin^{-1}(2(e_0 e_2 - e_1 e_3)) \quad (14.8)$$

$$\psi_z = \text{atan2}(2(e_0 e_3 + e_1 e_2), e_0^2 + e_1^2 - e_2^2 - e_3^2) \quad (14.9)$$

The relationship for ϕ_x is derived by dividing the second element by the third element in the third column of each matrix. Similarly, the relationship for ψ_z is derived by dividing the second element by the first element in the first row of each matrix. The relationship for θ_y is derived by comparing the third element in the first row of each matrix. Converting back, we get

$$e_0 = \cos\phi_x \cos\theta_y \cos\psi_z + \sin\phi_x \sin\theta_y \sin\psi_z \quad (14.10)$$

$$e_1 = \sin\phi_x \cos\theta_y \cos\psi_z - \cos\phi_x \sin\theta_y \sin\psi_z \quad (14.11)$$

$$e_2 = \cos\phi_x \sin\theta_y \cos\psi_z + \sin\phi_x \cos\theta_y \sin\psi_z \quad (14.12)$$

$$e_3 = \cos\phi_x \cos\theta_y \sin\psi_z - \sin\phi_x \sin\theta_y \cos\psi_z \quad (14.13)$$

14.3 Waypoint Tracking and Path Planning

For waypoint tracking, a list of 3-dimensional coordinates are provided. The goal is to have the airplane visit each of the 3D coordinates (waypoints) in the listed sequence at a desired speed or throttle position. Waypoint tracking algorithms use the 3D coordinates to calculate desired roll, pitch, and yaw angles. They determine the distance from the plane to the present waypoint. The algorithms determine when the plane has visited one waypoint and can move on to the next.

The GPS displacement algorithm of Eq. (14.1) calculates the distances Δx_I , Δy_I , and Δz_I (m) between the airplane and the target waypoint. The absolute distance between the airplane and the waypoint is

$$\sqrt{\Delta x_I^2 + \Delta y_I^2 + \Delta z_I^2} \quad (14.14)$$

The waypoint tracking algorithm can compare the absolute distance to a threshold to decide when one waypoint has been visited and to move onto the next. In general, as the airplane approaches a waypoint, the desired roll angle is zero:

$$\phi_{x,d} = 0 \quad (14.15)$$

The desired pitch angle is calculated from the absolute distance $\sqrt{\Delta x_I^2 + \Delta y_I^2 + \Delta z_I^2}$ and the change in altitude Δz_I as follows (see Figure 14.4):

$$\theta_{y,d} = \sin^{-1} \left(\frac{-\Delta z_I}{\sqrt{\Delta x_I^2 + \Delta y_I^2 + \Delta z_I^2}} \right) \quad (14.16)$$

Eq. (14.16) assumes a positive downward z-direction.

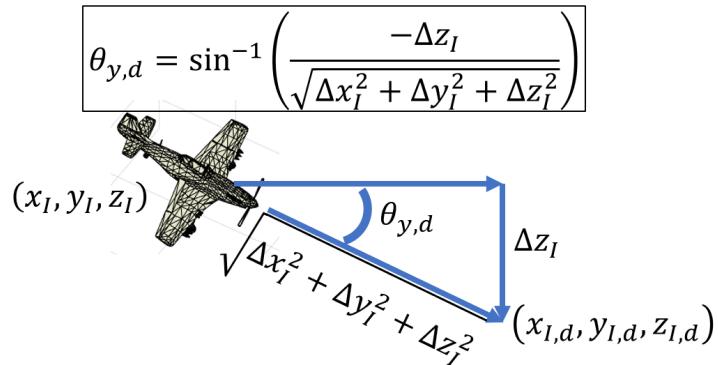


Figure 14.4 The desired pitch angle.

The desired yaw angle $\psi_{z,d}$ is calculated as

$$\psi_{z,d} = \text{atan2}(\Delta y_I, \Delta x_I) \quad (14.17)$$

where atan2 is the four-quadrant inverse tangent.

$$\phi_{x,d} = 0 \quad (14.18)$$

$$\theta_{y,d} = \sin^{-1} \left(\frac{-\Delta z_I}{\sqrt{\Delta x_I^2 + \Delta y_I^2 + \Delta z_I^2}} \right) \quad (14.19)$$

$$\psi_{z,d} = \text{atan2}(\Delta y_I, \Delta x_I) \quad (14.20)$$

14.4 Autonomous Control of Roll, Pitch, and Yaw

14.4 Autonomous Control of Roll, Pitch, and Yaw

The previous sections provided a way to calculate the roll ϕ_x , pitch θ_y , and yaw ψ_z and desired roll $\phi_{x,d}$, pitch $\theta_{y,d}$, and yaw $\psi_{z,d}$ Euler angles in units of radians. These signals can be used for feedback control. The feedback controllers also use the gyrometer measurements of roll rate ω_x , pitch rate ω_y , and yaw rate ω_z (rad/s) as feedback signals. This section will design autonomous strategies to control the roll, pitch, and yaw angles of the airplane.

Calculating Differences in Angles

Calculating the difference or error between two angles α_1 and α_2 is not always as simple as subtracting one from the other: $\alpha_2 - \alpha_1$. For example, on the unit circle, the difference between the angles $\alpha_1 = 179^\circ$ and $\alpha_2 = -179^\circ$ is $\Delta\alpha = 2^\circ$, but subtracting them results in a different value: $\alpha_2 - \alpha_1 = -179^\circ - 179^\circ = -358^\circ$, see Figure 14.5.

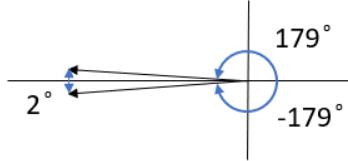


Figure 14.5 The difference is $\Delta\alpha = 2^\circ$ but subtracting them produces $\alpha_2 - \alpha_1 = -179^\circ - 179^\circ = -358^\circ$

The absolute shortest angle $|\delta\alpha|$ between α_1 and α_2 can be calculated by the cosine formula:

$$|\delta\alpha| = \cos^{-1}(\cos \alpha_1 \cos \alpha_2 + \sin \alpha_1 \sin \alpha_2) \quad (14.21)$$

The sign of the angle difference is also important. To get the signed difference $\Delta\alpha$ between α_1 and α_2 , we use the following logic:

```

 $|\delta\alpha| = \cos^{-1}(\cos \alpha_1 \cos \alpha_2 + \sin \alpha_1 \sin \alpha_2)$ 
if  $|\alpha_2 - \alpha_1| - |\delta\alpha| > 0.1$ 
     $\Delta\alpha = -\text{sign}(\alpha_2 - \alpha_1) |\delta\alpha|$ 
else
     $\Delta\alpha = \alpha_2 - \alpha_1$ 
end

```

(14.22)

14.4.1 Autonomous Flight Control

The autopilot strategy for autonomous flight depends on whether the plane has a rudder or not. A rudder facilitates yaw control, but rudderless yaw control works whether the plane has a rudder or not, assuming it has ailerons and an elevator (or elevons, as with a flying wing).

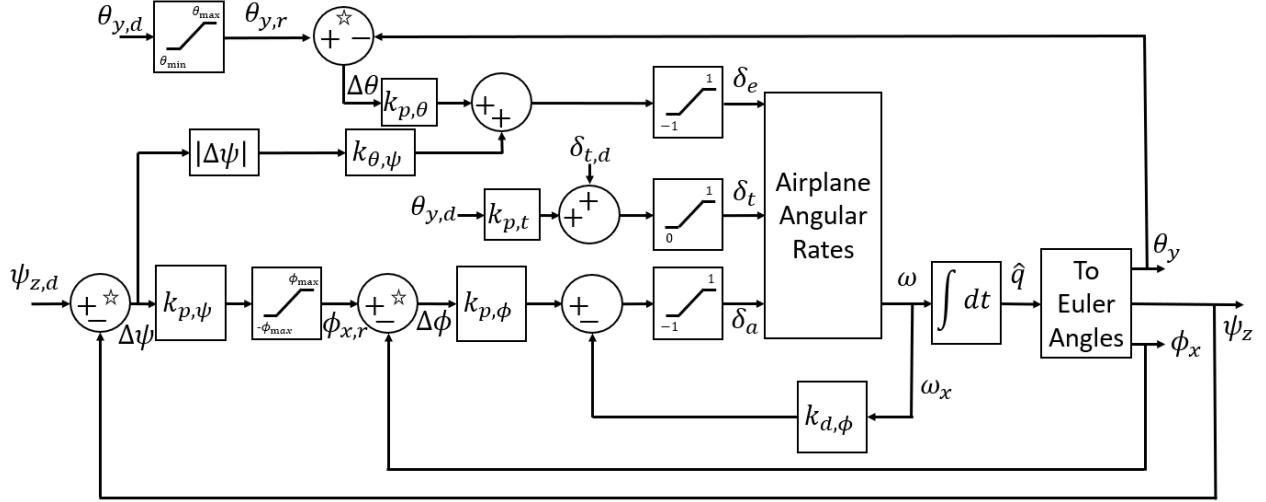


Figure 14.6 Block diagram of the rudderless control strategy.

Rudderless Control

Figure 14.6 shows a block diagram of a rudderless autonomous control strategy. The subtraction blocks containing stars indicate that the difference is calculated using the smallest angle formula of Eq. (14.22). The quaternion orientation \hat{q} is calculated using the strategies of Section 14.1. The angular roll velocity ω_x is measured using a gyrometer. The desired angles $\psi_{z,d}$, $\theta_{y,d}$, and $\phi_{x,d}$ are found using the waypoint tracking algorithms of Section 14.3. The proportional and derivative gains $k_{p,\psi}$, $k_{p,\phi}$, $k_{p,\theta}$, $k_{\theta,\psi}$, $k_{p,t}$ and $k_{d,\phi}$ are user-selected. They can be chosen by trial and error; however, there are better methods such as root-locus design and successive loop closure.

The block diagram of Figure 14.6 shows that the reference roll angle $\phi_{x,r}$

$$\phi_{x,r} = k_{p,\psi} \Delta\psi \quad (14.23)$$

is saturated to stay within the limits $\phi_{x,r} \in [-\phi_{\max}, \phi_{\max}]$. Saturation limits prevent the plane from rolling too much when trying reach a desired yaw angle $\psi_{z,d}$. For example, if the plane must make a 180° turn, setting the saturation limits to $\phi_{x,r} \in [-\pi/4, \pi/4]$ prevents the airplane from banking beyond $\pm 45^\circ$ while making the turn. All the angle differences $\Delta\psi$, $\Delta\phi$, and $\Delta\theta$ are calculated using the smallest angle difference formula, Eq. (14.22). The roll error $\Delta\phi$

$$\Delta\phi = \phi_{x,r} - \phi_x \quad (14.24)$$

calculated by Eq. (14.22) is the smallest angle difference between the reference roll angle $\phi_{x,r}$ and the actual roll angle ϕ_x of the plane. The yaw error $\Delta\psi$

$$\Delta\psi = \psi_{z,d} - \psi_z \quad (14.25)$$

calculated by Eq. (14.22) is the smallest angle difference between the desired yaw angle $\psi_{z,d}$ and the actual yaw angle ψ_z .

The aileron command δ_a is a proportional-derivative (PD) controller that depends on the roll error $\Delta\phi$ and the roll-rate ω_x .

$$\delta_a = k_{p,\phi}\Delta\phi - k_{d,\phi}\omega_x \quad (14.26)$$

14.4 Autonomous Control of Roll, Pitch, and Yaw

It is constrained to within $\delta_a \in [-1, 1]$. The proportional and derivative gains, $k_{p,\phi}$ and $k_{d,\phi}$ respectively, are user-selected tuning parameters.

Like roll angles, the reference pitch angle $\theta_{y,r}$ is constrained to be within limits $\theta_{y,r} \in [\theta_{y,\min}, \theta_{y,\max}]$:

$$\theta_{y,r} = \max(\theta_{y,\min}, \min(\theta_{y,d}, \theta_{y,\max})) \quad (14.27)$$

For example, the reference pitch angle $\theta_{y,r}$ may be saturated to be within the limits $\pm\frac{\pi}{4}$. The pitch error $\Delta\theta$

$$\Delta\theta = \theta_{y,r} - \theta_y \quad (14.28)$$

is the smallest angle difference (see Eq. (14.22)) between the reference pitch angle $\theta_{y,r}$ and the actual pitch angle θ_y of the airplane. The elevator command δ_e is proportional to the pitch error if there is no yaw error $\Delta\psi$. The combination of roll plus pitch can result in a change in yaw. Therefore, the elevator command is also proportional to the absolute value of the yaw error $|\Delta\psi|$.

$$\boxed{\delta_e = k_{p,\theta}\Delta\theta + k_{\theta,\psi}|\Delta\psi|} \quad (14.29)$$

Elevator commands are constrained to $\delta_e \in [-1, 1]$. Recall that an elevator command of $\delta_e = -1$ causes the airplane to pitch up. Therefore, $k_{p,\theta}$ and $k_{\theta,\psi}$ should be negative. The throttle command δ_t and gravity, combined with elevator commands, are used to decrease the airplane's altitude.

Having a negative angle of attack is one way to cause an airplane to decrease its altitude. Another way is to maintain a zero or slightly positive angle of attack but decrease the airspeed. Decreased airspeed results in less lift, and gravity causes the airplane to lose altitude. Conversely, climbing to higher altitudes is benefited by higher airspeeds. Eq. (14.19) shows that if the waypoint altitude is below the plane's altitude, that $\theta_{y,d}$ is negative. For these reasons, the throttle command δ_t is calculated as a function of the pitch error $\theta_{y,d}$ and the desired or cruise-speed throttle command $\delta_{t,d}$.

$$\boxed{\delta_t = \delta_{t,d} + k_{p,t}\theta_{y,d}} \quad (14.30)$$

The throttle command is constrained be between off (0) and full throttle (1), $\delta_t \in [0, 1]$.

In this control strategy, the desired throttle command $\delta_{t,d}$ is set by the waypoint algorithm, see Section 14.3. It is the cruise-speed throttle command to maintain airspeed when the airplane has reached a desired altitude. The rudderless control sets the rudder command to $\delta_r = 0$.

Control with a Rudder

If the airplane has a rudder, the yaw control can be improved. The rudderless control algorithm of Figure 14.6 (see Eqs. (14.23) - (14.29)) is still applicable, and can be augmented with a proportional rudder command δ_r

$$\boxed{\delta_r = k_\psi\Delta\psi} \quad (14.31)$$

where $\Delta\psi$ is calculated using the smallest angle formula Eq. (14.22). The proportional gain k_ψ is a user-selected tuning parameter. A block diagram of the control strategy is shown in Figure 14.7.

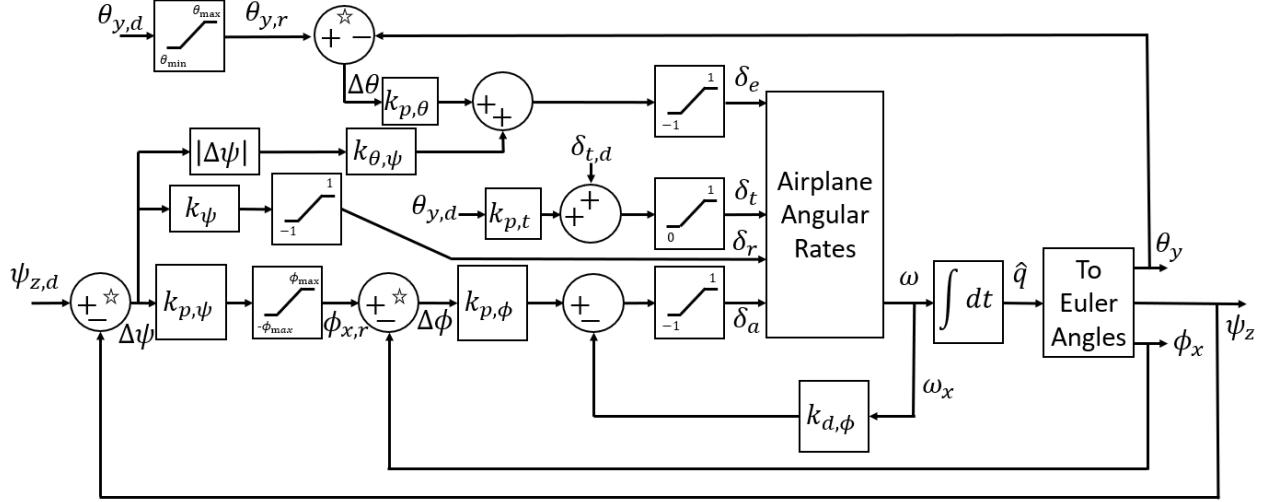


Figure 14.7 Block diagram of the autonomous flight control strategy with rudder control.

14.5 Constraints as Guides for Selecting Feedback Control Gains

The commands δ_t , δ_e , δ_a , and δ_r are physically constrained to be within saturation limits. For example, full throttle cannot physically be exceeded, and so the throttle command is constrained within the limits $\delta_t \in [0, 1]$. Consequently, the feedback gains $k_{p,t}$, $k_{p,\phi}$, $k_{d,\phi}$, etc., should also have constraints to avoid reaching saturation limits. This section uses these constraints to help select feedback control gains. Even when other controller design techniques are used, such as root-locus or successive loop closure, this section can provide approximate limits that can aid in selecting feedback control gains.

First consider the aileron feedback loop. Figures 14.6 and 14.7 show that the aileron command δ_a is constrained to be within the limits $\delta_a \in [-1, 1]$. They also show that the controller strives to keep the airplane's roll angle ϕ_x within the limits $\phi_x \in [-\phi_{\max}, \phi_{\max}]$. Consider a situation in which the airplane is banked with a roll angle of $-\phi_{\max}$, but the desired roll angle is $+\phi_{\max}$. The error is $\Delta\phi = 2\phi_{\max}$. This represents the largest roll error that should be expected during controlled flight. In proportional roll control, the aileron command is proportionally related to the roll error: $\delta_a = k_{p,\phi}\Delta\phi$. The maximum aileron command is $\delta_a = 1$ and the maximum roll error is $\Delta\phi = 2\phi_{\max}$. Therefore, the proportional control law provides a constraint for the control gain $k_{p,\phi}$.

$$0 \leq k_{p,\phi} \leq \frac{1}{2\phi_{\max}} \quad (14.32)$$

The constraints on the elevator command are $\delta_e \in [-1, 1]$. The feedback control strategies of Figures 14.6 and 14.7 strive to restrain the airplane's pitch to within the limits $\theta_y \in [\theta_{\min}, \theta_{\max}]$. The proportional control law is $\delta_e = k_{p,\theta}\Delta\theta$. The largest error in the airplane's pitch is when the airplane is at a pitch angle of θ_{\max} but the desired pitch angle is θ_{\min} . In this extreme case, the pitch error is $\Delta\theta = \theta_{\max} - \theta_{\min}$. Therefore, the proportional control law provides a constraint for the control gain $k_{p,\theta}$.

$$\frac{-1}{\theta_{\max} - \theta_{\min}} \leq k_{p,\theta} \leq 0 \quad (14.33)$$

14.6 Modifications for Flying Wing Airplanes

The proportional feedback gain $k_{p,\theta}$ should be a negative number. Negative elevator commands result in positive pitching motion.

The largest yaw error is 180° or $\Delta\psi = \pi$ rad. The rudder command is constrained to the saturation limits $\delta_r \in [-1, 1]$. Therefore, the proportional feedback gain constraints are

$$0 \leq k_\psi \leq \frac{1}{\pi} \quad (14.34)$$

and

$$0 \leq k_{p,\psi} \leq \frac{2\phi_{\max}}{\pi} \quad (14.35)$$

The throttle command is limited to $\delta_t \in [0, 1]$. Differences between the actual and desired altitudes result in a maximum desired pitch angle of $\theta_{y,d} = \pi$. The constraints on the proportional feedback gain $k_{p,t}$ are therefore

$$0 \leq k_{p,t} \leq \frac{1}{\pi} \quad (14.36)$$

These are not hard constraints. Exceeding them may or may not lead to poor performance. If feedback control gains are chosen outside the limits of these constraints, the airplane commands are more likely to reach saturation limits during controlled flight. Reaching saturation limits does not necessarily indicate poor performance. Therefore, the constraints of this section should be considered a guide rather than a rule for selecting proportional feedback control gains. Other control design techniques, such as successive loop closure or root locus can be used to improve the selection of the proportional and derivative feedback gains.

14.6 Modifications for Flying Wing Airplanes

Flying wing airplanes, such as the Boomerang Warbler of Section 8.1, have elevons instead of an elevator and ailerons. The word “elevon” is a combination of the words elevator and aileron; elevons perform both the functions of the elevator and the ailerons. There is a left elevon command: $\delta_{v,l}$, and a right elevon command: $\delta_{v,r}$. They are combinations of the elevator command δ_e and aileron command δ_a as follows:

$$\delta_{v,l} = \delta_e + \delta_a \quad (14.37)$$

$$\delta_{v,r} = -\delta_e + \delta_a \quad (14.38)$$

$$(14.39)$$

The elevon commands are saturated to ± 1 , i.e., $\delta_{v,l} \in [-1, 1]$ and $\delta_{v,r} \in [-1, 1]$. This mapping of the elevator and aileron commands to the elevon commands is just one possible relationship. Other linear or nonlinear combinations are possible as well.

Chapter 15

C++ Crash Course 1

Contents

15.1	Installing Visual Studio	452
15.2	Create, Compile, and Run a C++ Program	452
15.2.1	Explanation of the Hello World Code	455
15.3	Common Engineering Calculations	456
15.3.1	More Data Types in C++	457
15.4	Functions that Return One Variable	457
15.4.1	Do It Yourself	458
15.5	Functions that Return Multiple Variables	459
15.5.1	Do It Yourself	460
15.6	Basic Arrays and Matrices	460
15.6.1	Do It Yourself	463

This crash courses for learning C++ teaches the following concepts:

1. How to create, compile, and run a C++ program in Visual Studio
2. How and why to include libraries in your C++ programs
3. How to declare variables and what data types are
4. How to write outputs to the console window
5. How to perform common engineering calculations
6. How to write functions that can return one variable
7. How to use pointers to enable functions to return multiple variables
8. How to create basic arrays and matrices
9. How to perform basic calculations with arrays and matrices

15.1 Installing Visual Studio

The C++ Crash Courses in this book use Visual Studio. If needed, download the Community version of Visual Studio for free at the website <https://visualstudio.microsoft.com/downloads/> (accessed April 2023). Download the installer by clicking the **Free Download** button on the website. Once the download completes, launch the installer and follow the instructions until you see the following options shown in Figure 15.1 (it may be for a newer version of Visual Studio than what is shown):

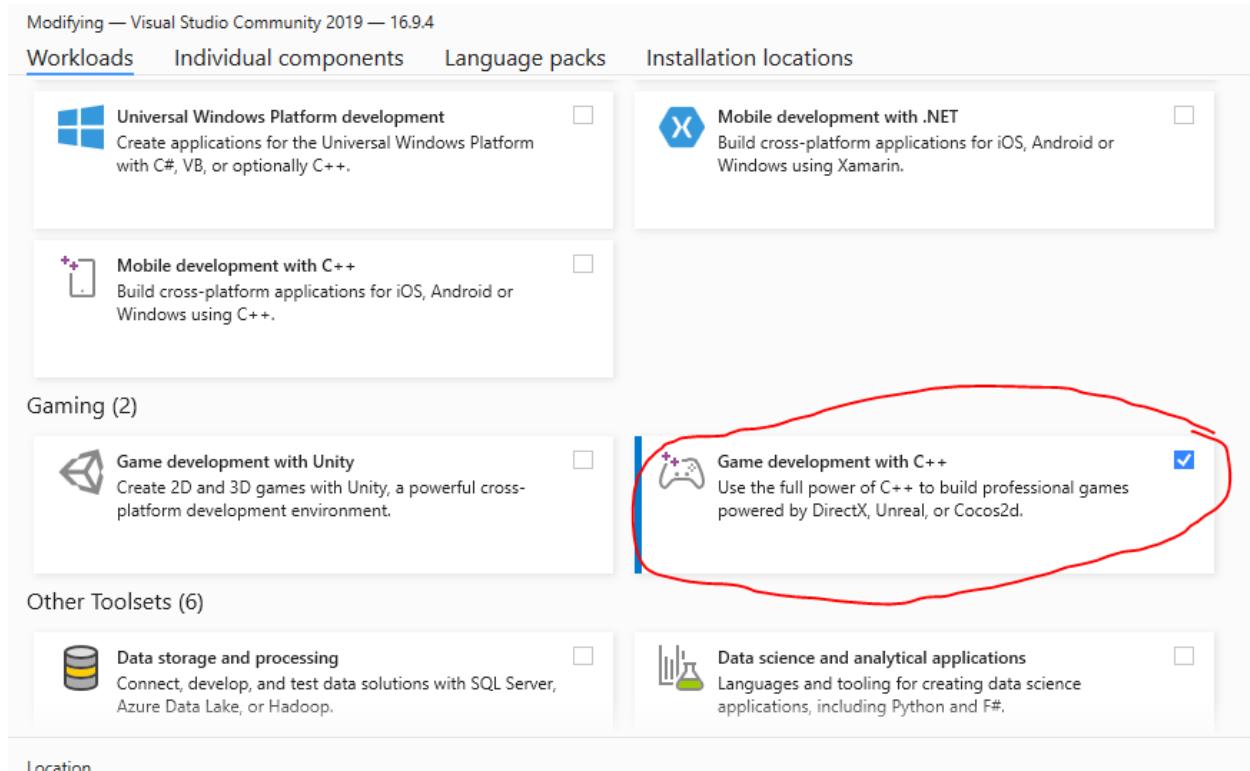


Figure 15.1 When installing Visual C++, select Game development with C++.

Select Game development with C++ by placing a checkmark in its box. Also make sure that checkmarks are in the same Optional boxes as shown in the figure. If you have already installed Visual Studio, you can modify it by running the Visual Studio Installer. Then click Modify. If you do not see the Modify option, select More, then Modify.

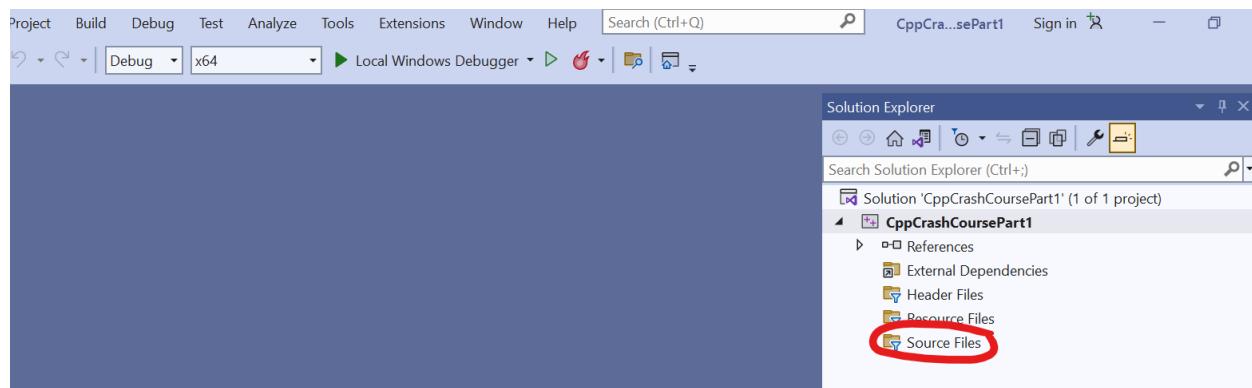
15.2 Create, Compile, and Run a C++ Program

The following steps will walk you through creating, compiling, and running one of the most basic C++ programs: “Hello World”. The examples of this chapter were generated using Microsoft Visual Studio Community 2022 (64-bit) Version 17.3.3. Different versions may have different steps.

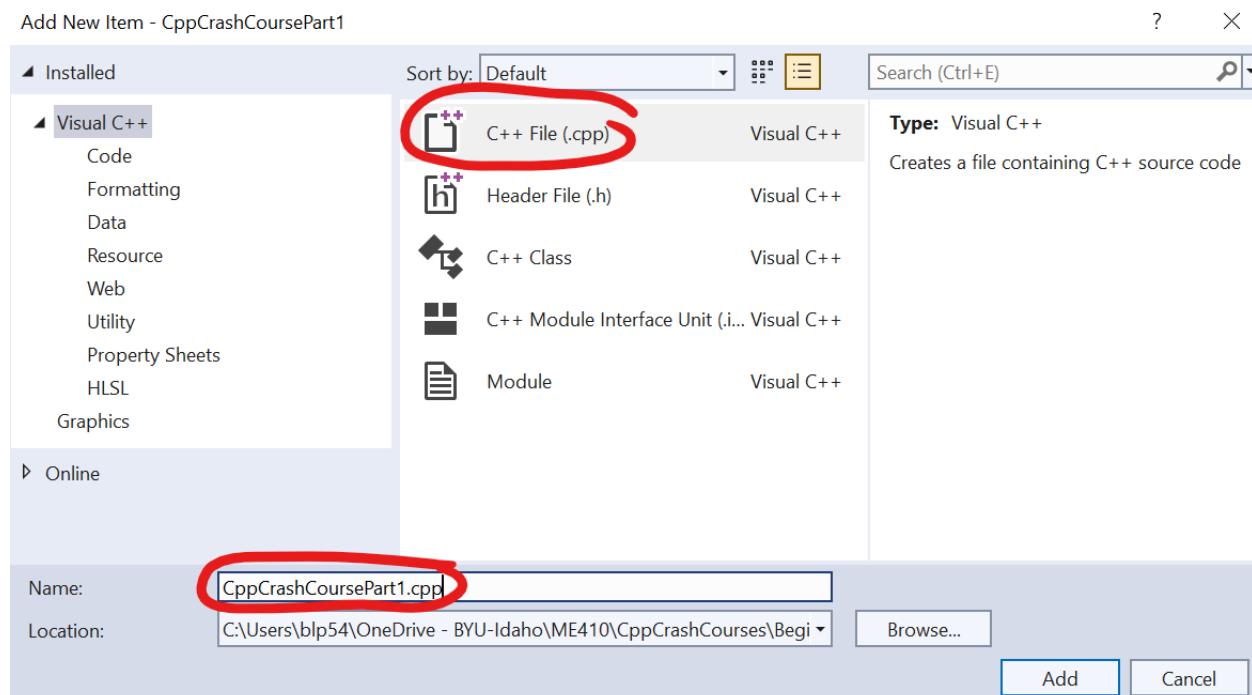
- Open Visual Studio
- Select Create a new project

15.2 Create, Compile, and Run a C++ Program

- Select Empty Project and click Next
- Click the three dots ... next to the folder location, and navigate to a folder location where you would like to save the project
- Name the project, for example, CppCrashCoursePart1
- Check the box next to Place solution and project in the same directory
- Select Create
- Right click on Source Files in the Solution Explorer window and select Add » New Item



- In the Add New Item window, select C++ File (.cpp), change the Name to (for example) CppCrashCoursePart1.cpp and click Add

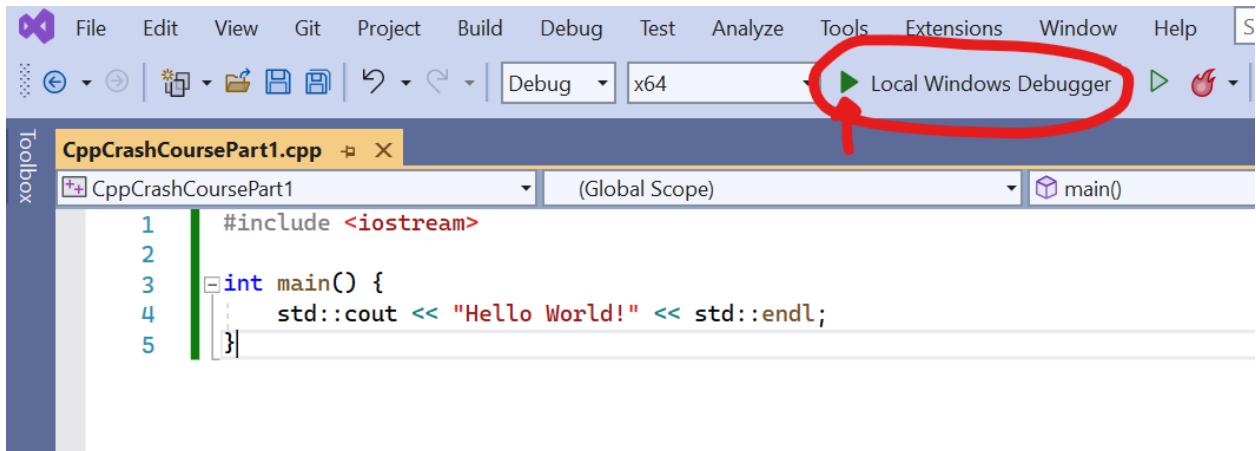


- A new window should open with the CppCrashCoursePart1.cpp file that you just added. (If not, you can open it by double-clicking CppCrashCoursePart1.cpp under Source Files in the Solution Explorer)
- Instead of copying the code in this assignment, it is recommended that you type it. Typing it helps you learn better, and may prevent compiler errors that are caused by incompatible font formats between this file and Visual Studio.
- Type the following code in the CppCrashCoursePart1.cpp file:

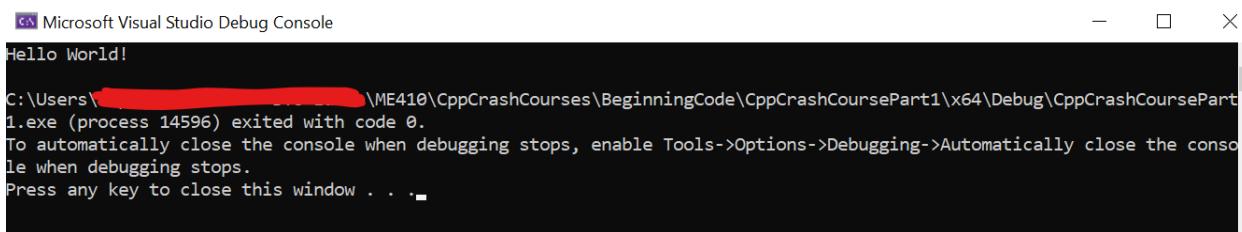
```
#include <iostream> //Add this command to use std::cout and std::endl

int main() { //The MAIN function is the required entry point for a C++ program
    std::cout << "Hello World!" << std::endl; //COUT prints "Hello World!" to the console.
                                                //ENDL moves the cursor to a new line.
    return 0; //The MAIN function should return an integer
}
```

- Then click the green play button next to Local Windows Debugger.



It should compile the code and produce a window that says Hello World! It should look something like



Congratulations! You have written and compiled a C++ program.

15.2 Create, Compile, and Run a C++ Program

15.2.1 Explanation of the Hello World Code

The Hello World code began with a command to include the iostream library:

```
#include <iostream>
```

This library allows the C++ program to use the commands cout and endl. The cout command prints letters, numbers, and other characters to the console window. The endl command is a carriage return that causes the cursor in the console window to move to the next line. A comment in C++ is made using two forward slashes //. Whatever follows the two forward slashes is not part of the compiled code. For example, the phrase //Add this command to use std::cout and std::endl that follows the #include <iostream> command is ignored by the C++ compiler. It is not part of the compiled program. Comments are helpful to let programmers write explanations about the code. The next line is the main() function:

```
int main() {}
```

Each C++ program needs an entry point, or code that is called first. For console programs like the one we created, the main() function is a required entry point. It is the function that C++ calls first. (Later we will create other functions, and they will be called from within the main() function). The code that is within the curly braces {} is part of the main() function. It is said to be within the *scope* of the main() function. The main() function returns an int data type. The int data type is an integer which is a whole number. Our program could have returned any whole number. In this example, it returned zero using the command

```
return 0;
```

The part of the program that printed Hello World to the console window was the line

```
std::cout << "Hello World!" << std::endl;
```

The cout command prints letters, numbers, and other characters to the console window. The endl command is a carriage return that causes the cursor in the console window to move to the next line. The cout command is part of the std namespace, meaning that cout will not work without using the identifier std:: or by using the command using namespace std. Understanding namespace is a more advanced topic that can be explored later. For now, just know that to use cout and endl, they must be used with the

std identifier.

15.3 Common Engineering Calculations

Engineers use math. To use many of the basic math operators, C++ code will need to include the library math.h. The math.h library includes at least the following math operations:

```
cos, sin, tan, acos, asin, atan, atan2, cosh, sinh, tanh, acosh, asinh, atanh, exp, frexp,
ldexp, log, log10, modf, exp2, expm1, ilogb, log1p, log2, logb, scalbn, scalbln, pow, sqrt,
cbrt, hypot, erf, erfc, tgamma, lgamma, ceil, floor, fmod, trunc, round, lround, llround,
rint, lrint, llrint, nearbyint, remainder, remquo, copysign, nan, nextafter, nexttoward,
fdim, fmax, fmin, fabs, abs, fma
```

To learn more about how to use these operators, do an Internet search using the keyword math.h. This section will practice using some of these math.h commands. Modify your CppCrashCoursePart1.cpp file as follows:

```
#include <iostream> //Add this command to use std::cout and std::endl
#include <math.h> //For math operations: sqrt, pow, abs, sin, cos, tan, asin, acos, atan, atan2

using namespace std; //allows us to remove std:: from std::cout and std::endl

int main() { //The MAIN function is the required entry point for a C++ program
    cout << "Hello World!" << endl; //COUT prints "Hello World!" to the console.
                                            //ENDL moves the cursor to a new line.

    //Perform very basic algebra such as add, subtract, multiply, and divide.
    //This does not require math.h
    double sum = 2.0 + 3.0;
    double difference = 2.0 - 3.0;
    double product = 2.0 * 3.0;
    double quotient = 2.0 / 3.0;
    cout << "sum = " << sum << endl; //Print the sum to the console window
    cout << "difference = " << difference << endl; //Print the difference to the console window
    cout << "product = " << product << endl; //Print the product to the console window
    cout << "quotient = " << quotient << endl; //Print the quotient to the console window

    //Perform math that does require the math.h library
    const double pi = 3.141592653589793238462643383279;
    double sin2pi = sin(2.0 * pi);
    double atan2_4_n2 = atan2(4.0, -2.0);
    double sixCubed = pow(6.0, 3.0);
    double sqrt_5 = sqrt(5.0);
    cout << "sin(2.0 * pi) = " << sin2pi << endl; //Print the sin of 2*pi to the console window
    cout << "atan2(4.0, -2.0) = " << atan2_4_n2 << endl; //Print the arctangent of 4/(-2)
    cout << "pow(6.0, 3.0) = " << sixCubed << endl; //Print 6^3 to the console window
```

15.4 Functions that Return One Variable

```
cout << "sqrt(5.0) = " << sqrt_5 << endl; //Print the square root of 5 to the console window  
return 0; //The MAIN function should return an integer  
}
```

Click the green play button next to Local Windows Debugger. Did the math operators work correctly? You may want to compare the result with what you get in MATLAB or some other program.

15.3.1 More Data Types in C++

You may have noticed some new keywords in the C++ code above. Many of these keywords tell the C++ compiler what type of data it needs to create memory for. The data type double in the phrase double sum = 2.0 + 3.0; indicates that sum is a floating point number with double precision. Floating point numbers have decimal places, unlike integers which are whole numbers. You may have also noticed the sum used the numbers 2.0 and 3.0 instead of 2 and 3. The C++ compiler automatically treats the numbers 2 and 3 like integers, but it treats 2.0 and 3.0 as double precision numbers. When writing code in C++, it can be extremely important to include the decimal on floating point numbers. Forgetting the decimal can lead to unexpected results that are difficult to debug.

Because of the keyword const, the phrase const double pi = 3.141592653589793238462643383279; defined pi as a constant that cannot be changed later in the C++ code. If an attempt was made to change pi to a different number later in the program, the compiler would throw an error.

The phrase using namespace std allows the C++ program to use the cout and endl commands directly without the std:: identifier.

15.4 Functions that Return One Variable

Functions in C++ require a function declaration, a function definition, and a call to the function. The declaration tells the compiler that the function exists. The definition contains the actual body of the function. The call to the function executes it. The function definition can double-count as the declaration if it is placed in the C++ code before it is called.

Modify your code to include the sind() function as shown below. The sind() function calculates the sine of an angle in degrees. The angle in degrees has a double precision data type, and the function returns a double precision number.

```
#include <iostream> //Add this command to use std::cout and std::endl  
#include <math.h> //For math operations: sqrt, pow, abs, sin, cos, tan, asin, acos, atan, atan2  
  
using namespace std; //allows us to remove std:: from std::cout and std::endl  
const double pi = 3.141592653589793238462643383279;  
  
//function declaration of sind()  
double sind(const double angle_degrees);
```

```

int main() { //The MAIN function is the required entry point for a C++ program
    cout << "Hello World!" << endl; //COUT prints "Hello World!" to the console.

    //Perform very basic algebra such as add, subtract, multiply, and divide.
    //This does not require math.h
    double sum = 2.0 + 3.0;
    double difference = 2.0 - 3.0;
    double product = 2.0 * 3.0;
    double quotient = 2.0 / 3.0;
    cout << "sum = " << sum << endl; //Print the sum to the console window
    cout << "difference = " << difference << endl; //Print the difference to the console window
    cout << "product = " << product << endl; //Print the product to the console window
    cout << "quotient = " << quotient << endl; //Print the quotient to the console window

    //Perform math that does require the math.h library
    double sin2pi = sin(2.0 * pi);
    double atan2_4_n2 = atan2(4.0, -2.0);
    double sixCubed = pow(6.0, 3.0);
    double sqrt_5 = sqrt(5.0);
    cout << "sin(2.0 * pi) = " << sin2pi << endl; //Print the sin of 2*pi to the console window
    cout << "atan2(4.0, -2.0) = " << atan2_4_n2 << endl; //Print the arctangent of 4/(-2)
    cout << "pow(6.0, 3.0) = " << sixCubed << endl; //Print 6^3 to the console window
    cout << "sqrt(5.0) = " << sqrt_5 << endl; //Print the square root of 5 to the console window

    //Function call to sind
    double sin_45 = sind(45.0);
    cout << "sin(45 degrees) = " << sin_45 << endl; //Print the result to the console

    return 0; //The MAIN function should return an integer
}

//function definition of sind()
double sind(const double angle_degrees) {
    double angle_radians = angle_degrees * pi / 180.0;
    return (sin(angle_radians));
}

```

Pay attention to the following: First, `const double pi = 3.141592653589793238462643383279;` was moved outside the scope of the `main()` function. This was necessary so that it could be used in both the `main()` and `sind()` functions. Before, `pi` was only defined within the curly braces {} of the `main()` function after the line on which it was declared.

Second, notice that the function declaration came before the function call. That allowed the code to place the function definition after the function call. Finally, notice that the `sind()` function returned the double-precision value calculated by `sin(angle_radians)`.

15.4.1 Do It Yourself

Create your own function named `cosd()`. It should take an input angle in degrees and output the cosine of the angle. Print the result to the C++ console window.

15.5 Functions that Return Multiple Variables

15.5 Functions that Return Multiple Variables

If a C++ function must return multiple variables, it must pass them by reference. C++ uses pointers, denoted with asterisks (*), and the & operator to pass variables by reference. When passing a variable by reference, instead of passing the variable to the function, the function call passes the memory location of the variable to the function using the & operator. The function uses a pointer (*) to change the variable in the memory location that was passed to it.

Modify your code as follows to create a function that can return multiple variables.

```
#include <iostream> //Add this command to use std::cout and std::endl
#include <math.h> //For math operations: sqrt, pow, abs, sin, cos, tan, asin, acos, atan, atan2

using namespace std; //allows us to remove std:: from std::cout and std::endl
const double pi = 3.141592653589793238462643383279;

//function declaration of sind()
double sind(const double angle_degrees);

//This is both the function declaration and definition. It must come before the function call
void manyOutputs(
    double* output1, //first output
    double* output2, //second output
    const double input1, //first input
    const double input2, //second input
    const double input3 //third input
) {
    //Access the variable in the memory location of output1 to change it
    *output1 = input1 + input2;

    //Access the variable in the memory location of output2 to change it
    *output2 = input2 * input3;
}

int main() { //The MAIN function is the required entry point for a C++ program
    cout << "Hello World!" << endl; //Print "Hello World!" to the console.

    //Perform very basic algebra such as add, subtract, multiply, and divide.
    //This does not require math.h
    double sum = 2.0 + 3.0;
    double difference = 2.0 - 3.0;
    double product = 2.0 * 3.0;
    double quotient = 2.0 / 3.0;
    cout << "sum = " << sum << endl; //Print the sum to the console window
    cout << "difference = " << difference << endl; //Print the difference to the console window
    cout << "product = " << product << endl; //Print the product to the console window
    cout << "quotient = " << quotient << endl; //Print the quotient to the console window

    //Perform math that does require the math.h library
    double sin2pi = sin(2.0 * pi);
    double atan2_4_n2 = atan2(4.0, -2.0);
```

```

double sixCubed = pow(6.0, 3.0);
double sqrt_5 = sqrt(5.0);
cout << "sin(2.0 * pi) = " << sin2pi << endl; //Print the sin of 2*pi to the console window
cout << "atan2(4.0, -2.0) = " << atan2_4_n2 << endl; //Print the arctangent of 4/(-2)
cout << "pow(6.0, 3.0) = " << sixCubed << endl; //Print 6^3 to the console window
cout << "sqrt(5.0) = " << sqrt_5 << endl; //Print the square root of 5 to the console window

//Function call to sind
double sin_45 = sind(45.0);
cout << "sin(45 degrees) = " << sin_45 << endl;

//Function call to manyOutputs()
double out1, out2;
manyOutputs(&out1, &out2, 1.0, 2.0, 3.0);
cout << "out1 = " << out1 << " and out2 = " << out2 << endl;

return 0; //The MAIN function should return an integer
}

//function definition of sind()
double sind(const double angle_degrees) {
    double angle_radians = angle_degrees * pi / 180.0;
    return (sin(angle_radians));
}

```

Pay attention to how the function call to `manyOutputs()` passed the memory locations (`&out1` and `&out2`) of the function outputs to the function `manyOutputs()`. The function referenced the values in the memory locations using the pointer commands `*output1 = input1 + input2;` and `*output2 = input2 * input3;` The function `manyOutputs()` never used the `return()` command. That is because it was of data type `void`. Void functions do not return anything directly, and can only return variables by passing them by reference.

15.5.1 Do It Yourself

Create a function named `divideAndMultiply()`. The function should have two inputs and two outputs. When calling the function, the two inputs should be the numbers 4 and 5. The two outputs should be the quotient (4/5) and product (4*5) of the inputs. Use the concept of passing by reference. Print the results to the console window.

15.6 Basic Arrays and Matrices

This section uses basic arrays and matrices. A later section will provide a more extensive framework for using matrices and arrays. C++ has many ways of working with arrays and matrices. This part of the tutorial teaches one of the most basic ways. Modify your code as follows:

15.6 Basic Arrays and Matrices

```
#include <iostream> //Add this command to use std::cout and std::endl
#include <math.h> //Math operations: sqrt, pow, abs, sin, cos, tan, asin, acos, atan, atan2

using namespace std; //allows us to remove std:: from std::cout and std::endl
const double pi = 3.141592653589793238462643383279;

//function declaration of sind()
double sind(const double angle_degrees);

//This is both the function declaration and definition. It must come before the function call
void manyOutputs(
    double* output1, //first output
    double* output2, //second output
    const double input1, //first input
    const double input2, //second input
    const double input3 //third input
) {
    //Access the variable in the memory location of output1 to change it
    *output1 = input1 + input2;

    //Access the variable in the memory location of output2 to change it
    *output2 = input2 * input3;
}

//Declare the matrixMultiply function C = A * B
void matrixMultiply(double C[2][4], const double A[2][3], const double B[3][4]);

int main() { //The MAIN function is the required entry point for a C++ program
    cout << "Hello World!" << endl; //COUT prints "Hello World!" to the console.
                                            //ENDL moves the cursor to a new line.

    //Perform very basic algebra such as add, subtract, multiply, and divide.
    //This does not require math.h
    double sum = 2.0 + 3.0;
    double difference = 2.0 - 3.0;
    double product = 2.0 * 3.0;
    double quotient = 2.0 / 3.0;
    cout << "sum = " << sum << endl; //Print the sum to the console window
    cout << "difference = " << difference << endl; //Print the difference to the console window
    cout << "product = " << product << endl; //Print the product to the console window
    cout << "quotient = " << quotient << endl; //Print the quotient to the console window

    //Perform math that does require the math.h library
    double sin2pi = sin(2.0 * pi);
    double atan2_4_n2 = atan2(4.0, -2.0);
    double sixCubed = pow(6.0, 3.0);
    double sqrt_5 = sqrt(5.0);
    cout << "sin(2.0 * pi) = " << sin2pi << endl; //Print the sin of 2*pi to the console window
    cout << "atan2(4.0, -2.0) = " << atan2_4_n2 << endl; //Print the arctangent of 4/(-2)
    cout << "pow(6.0, 3.0) = " << sixCubed << endl; //Print 6^3 to the console window
    cout << "sqrt(5.0) = " << sqrt_5 << endl; //Print the square root of 5 to the console window
```

```

//Function call to sind
double sin_45 = sind(45.0);
cout << "sin(45 degrees) = " << sin_45 << endl;

//Function call to manyOutputs()
double out1, out2;
manyOutputs(&out1, &out2, 1.0, 2.0, 3.0);
cout << "out1 = " << out1 << " and out2 = " << out2 << endl;

//Define two arrays with three elements each and calculate their cross product
float vec1[3] = { 1.0f, 2.0f, 3.0f };
float vec2[3] = { 3.0f, 2.0f, 1.0f };
float crossProduct[3];
crossProduct[0] = vec1[1] * vec2[2] - vec2[1] * vec1[2];
crossProduct[1] = vec1[2] * vec2[0] - vec2[2] * vec1[0];
crossProduct[2] = vec1[0] * vec2[1] - vec2[0] * vec1[1];
cout << "cross(vec1, vec2) = [ ";
for (float& elem : crossProduct) {
    cout << elem << " ";
}
cout << "]" << endl;

//Multiply a 2x3 matrix by a 3x4 matrix C = A * B
double A[2][3] = { {1.0,2.0,3.0},{3.0,2.0,1.0} };
double B[3][4] = { {1.0,2.0,3.0,4.0},{5.0,6.0,7.0,8.0},{9.0,10.0,11.0,12.0} };
double C[2][4];
matrixMultiply(C, A, B);
cout << "C = [ ";
for (unsigned int ii = 0; ii < 2; ii++) {
    for (int jj = 0; jj < 4; jj++) {
        cout << C[ii][jj] << " ";
    }
    if (ii < 1)
        cout << endl;
    else
        cout << "]" << endl;
}

return 0; //The MAIN function should return an integer
}

//function definition of sind()
double sind(const double angle_degrees) {
    double angle_radians = angle_degrees * pi / 180.0;
    return (sin(angle_radians));
}

//Define the matrixMultiply function C = A * B
void matrixMultiply(double C[2][4], const double A[2][3], const double B[3][4]) {
    for (int ii = 0; ii < 2; ii++) {
        for (int jj = 0; jj < 4; jj++) {
            C[ii][jj] = 0.0;
            for (int kk = 0; kk < 3; kk++) {

```

15.6 Basic Arrays and Matrices

```
    C[ii][jj] += A[ii][kk] * B[kk][jj];  
}  
}  
}  
}
```

Pay special attention to how matrices are passed to the function `matrixMultiply()`. This approach requires the matrix sizes to be included in the function declaration and definition. Passing matrices without specifying their sizes is beyond the scope of this assignment but will be taught in later C++ Crash Courses. The function definition of `matrixMultiply()` includes three nested for loops. The line `C[ii][jj] += A[ii][kk] * B[kk][jj];` is equivalent to $C_{ii,jj} = C_{ii,jj} + A_{ii,kk} * B_{kk,jj};$.

The for loop for printing the cross product includes the phrase `for (float& elem : crossProduct)`. This phrase is especially useful when the size of the array (`crossProduct` in this example) is unknown. It iterates through each element of the `crossProduct` and assigns the element to the floating point value `elem`. The `float` data type is for single precision floating point (decimal) numbers.

15.6.1 Do It Yourself

Create a function that can add two 2x2 matrices: $A = \{ \{1.0, 2.0\}, \{3.0, 4.0\} \}$ and $B = \{ \{5.0, 6.0\}, \{7.0, 8.0\} \}$. Print the result $C = A + B$; to the console window.

This concludes the first C++ crash course.

Chapter 16

C++ Crash Course 2

Contents

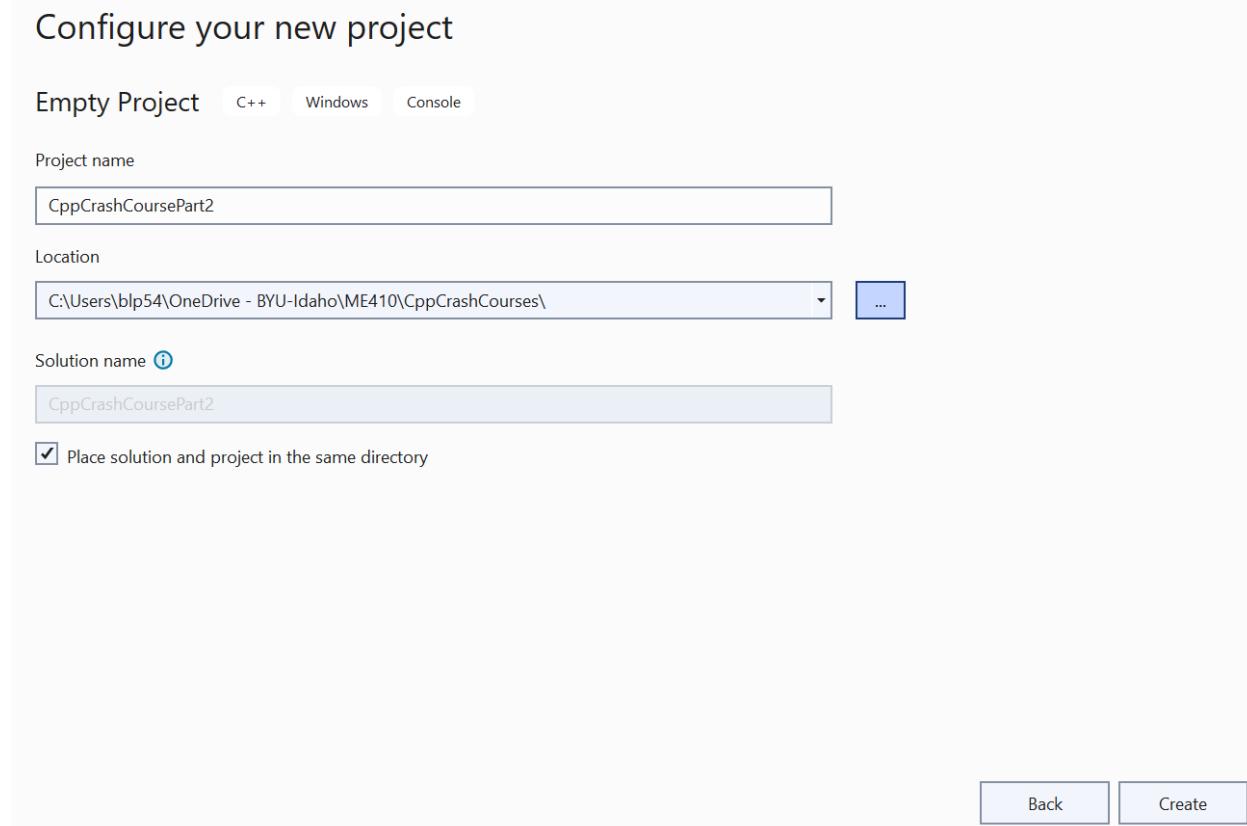
16.1 Creating a Program with Multiple Files	465
16.2 Header Files and Source Files	469
16.2.1 MyMatrixMath.cpp	469
16.2.2 MyMatrixMath.h	470
16.2.3 CppCrashCoursePart2.cpp	470
16.2.4 The vector Library	471
16.3 The MyMatrixMath Library	472
16.3.1 MyMatrixMath.h	472
16.3.2 MyMatrixMath.cpp	473
16.4 Solving $\dot{x} = Ax + Bu$	480
16.4.1 CppCrashCoursePart2.cpp	480
16.4.2 Do It Yourself	483

This is the second C++ crash course. It teaches the following concepts:

1. How to create, compile, and run a C++ program with multiple source files
2. How header files (.h) and source files (.cpp) differ
3. How to use the vector library when array or matrix sizes are unknown in advance
4. How to perform matrix operations like matrix inversions and matrix exponentials
5. How to solve the state-space equation $\dot{x} = Ax + Bu$

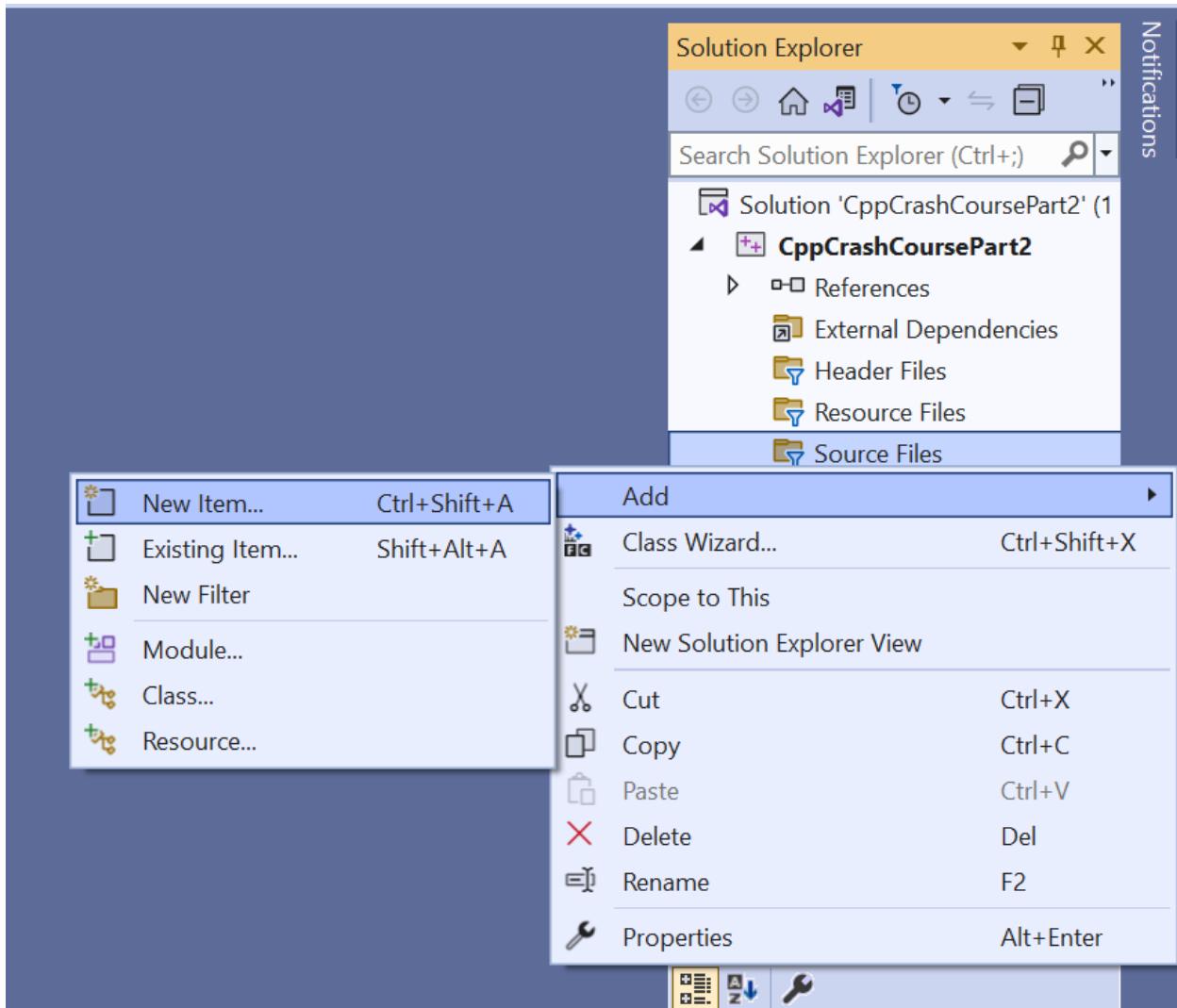
16.1 Creating a Program with Multiple Files

Open Visual Studio and select Create a New Project. Select the Empty Project option, and click Next. Change the Project name to CppCrashCoursePart2, choose the Location, checkmark the box Place solution and project in the same directory and click Create.

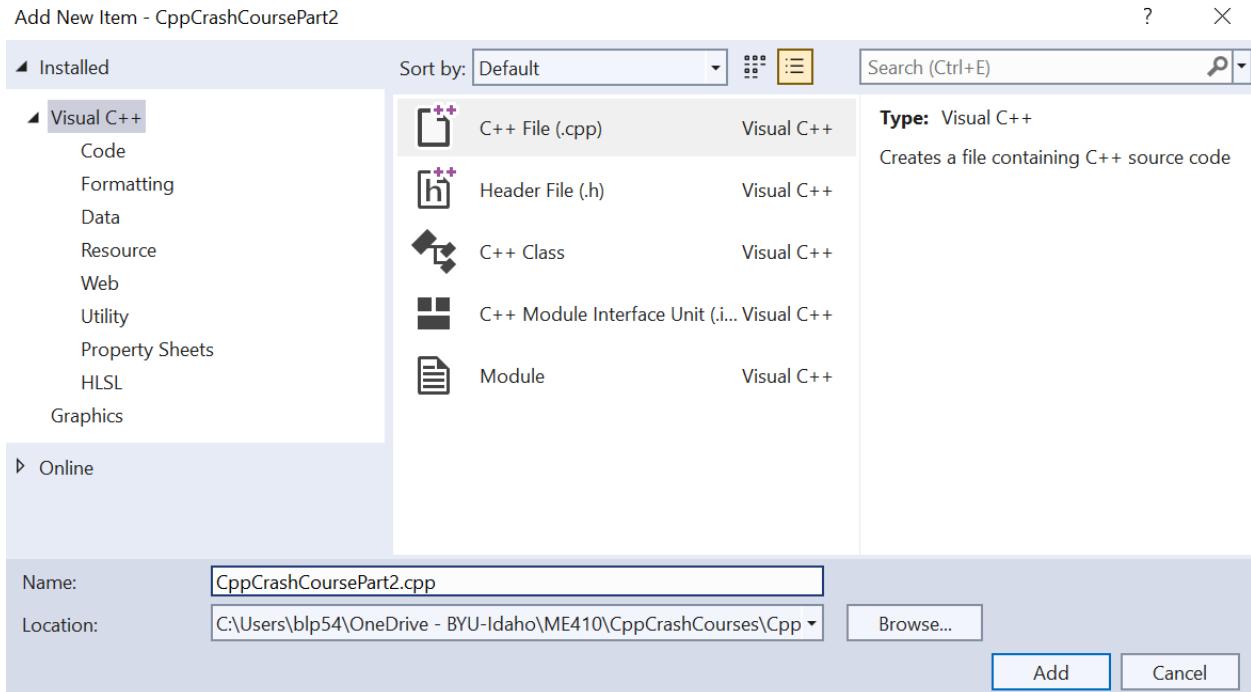


Right-click Source Files in the Solution Explorer, then select Add, then select New Item...

16.1 Creating a Program with Multiple Files



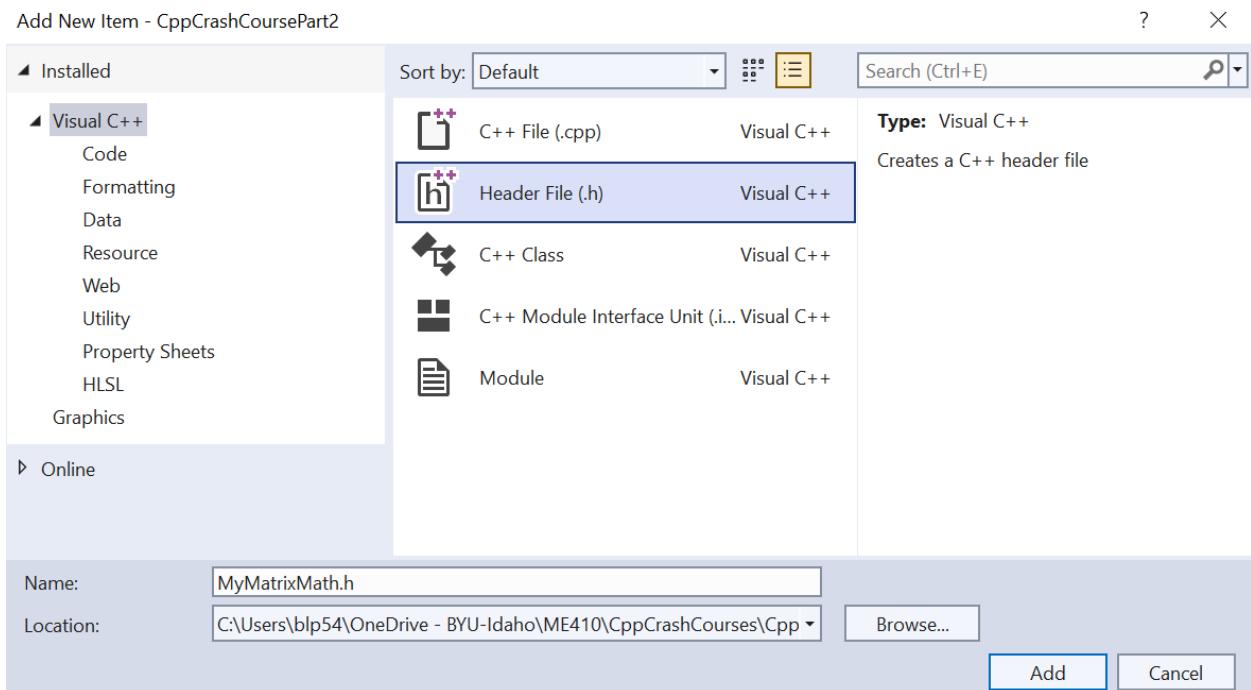
Select C++ File (.cpp), name the file CppCrashCoursePart2.cpp, and click Add.



The CppCrashCoursePart2.cpp file will be the main file in our program.

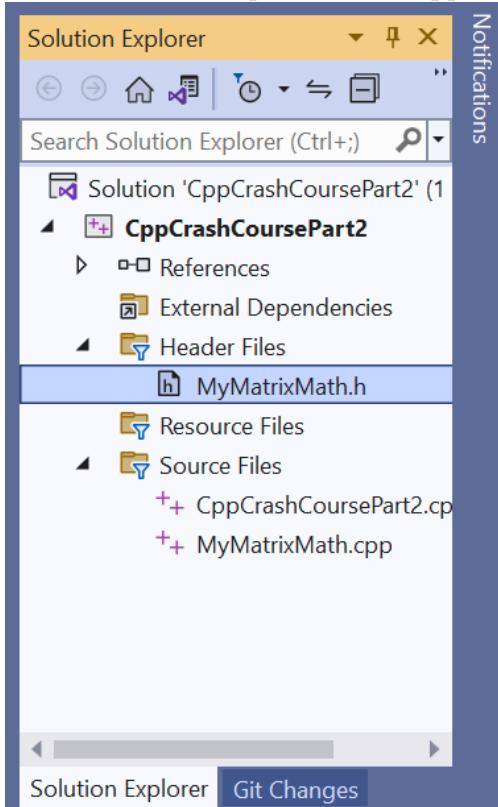
Repeat the steps above to add another source file named MyMatrixMath.cpp.

After the MyMatrixMath.cpp file has been added, right-click Header Files in the Solution Explorer, then select Add, then select New Item... This time, however, select Header File (.h), name the file MyMatrixMath.h, and click Add.



16.2 Header Files and Source Files

Your Solution Explorer should appear as follows:



If not, please review the previous steps until your Solution Explorer is correct.

16.2 Header Files and Source Files

In the first C++ Crash Course, you should have performed basic matrix calculations (add, multiply). Matrix sizes must be known in advance to use the basic matrix calculations of the first C++ Crash Course. The vector library will allow us to generalize our C++ code to work for matrices of any size. Instead of copying the code in this assignment, it is recommended that you type it. Typing it helps you learn better and may prevent compiler errors that are caused by incompatible font formats with Visual Studio.

Open the MyMatrixMath.cpp file by clicking on it in the Solution Explorer. Modify MyMatrixMath.cpp as follows:

16.2.1 MyMatrixMath.cpp

```
#include <iostream> //cout, endl
#include <vector> //vector
#include "MyMatrixMath.h" //Header file we create for matrix math

using namespace std; //To use cout and endl without std::cout and std::endl
```

```
//Function to print the matrix to the console window
extern void printMatrix(vector<vector<double>> A) {
    for (int ii = 0; ii < (int)A.size(); ii++) {
        for (int jj = 0; jj < (int)A[0].size(); jj++) {
            cout << A[ii][jj] << " ";
        }
        cout << endl; //Go to the next row of the matrix
    }
}
```

Modify the MyMatrixMath.h file as follows:

16.2.2 MyMatrixMath.h

```
#pragma once //ensure MyMatrixMath.h is compiled only once
#include <vector> //vector type

using namespace std; //To use vector without std::vector

// Declare the printMatrix function
extern void printMatrix(vector<vector<double>> A);
```

Modify the CppCrashCoursePart2.cpp file as follows:

16.2.3 CppCrashCoursePart2.cpp

```
#include <iostream> //cout, endl
#include <vector> //vector type
#include "MyMatrixMath.h" //Include our MyMatrixMath library

using namespace std; //To use cout, endl, and vector without std::

//The program's main function
int main() {
    //Declare and define the 3x2 matrix B
    vector<vector<double>> B{ {-4.0,2.0},{1.0,-0.6},{5.0,1.0} };

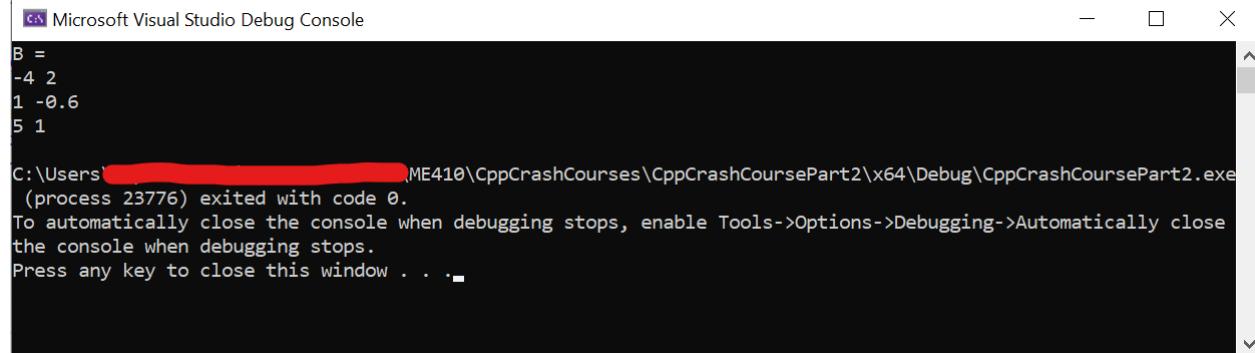
    //Print the matrix B using the MyMatrixMath library function
    cout << "B = " << endl;
    printMatrix(B);

    //Return 0 to indicate that the main function was successful
    return 0;
}
```

16.2 Header Files and Source Files

```
}
```

Click the green play arrow next to Local Windows Debugger to compile and run the code. If there are any compiler errors, check that you have followed all of the steps precisely, including making your Solution Explorer match the one presented above. If the code runs, you should get an output in the console window that is something like this:



The screenshot shows a Microsoft Visual Studio Debug Console window. The output is as follows:

```
Microsoft Visual Studio Debug Console
B =
-4 2
1 -0.6
5 1

C:\Users\██████████\ME410\CppCrashCourses\CppCrashCoursePart2\x64\Debug\CppCrashCoursePart2.exe
(process 23776) exited with code 0.
To automatically close the console when debugging stops, enable Tools->Options->Debugging->Automatically close
the console when debugging stops.
Press any key to close this window . . .
```

This program has three files: two source files (CppCrashCoursePart2.cpp, MyMatrixMath.cpp), and one header file (MyMatrixMath.h). The source file CppCrashCoursePart2.cpp is the main file because it includes the entry function to the program. In this case, the entry function is the int main() function. A program can have multiple source files, but only one can include the program's entry function. The int main() function is one type of entry function. Other types of entry functions exist. For example, Arduino programs use the setup() and loop() functions as entry functions that are compiled with C++ libraries containing source (.cpp) and header (.h) files. Matlab uses specific mex functions as the program's entry functions.

The header file MyMatrixMath.h contains function declarations but no function definitions. As a reminder, function declarations tell the compiler that the function exists. The function definition is the code that is executed when the function is called. Although it is possible to include both the function declarations and definitions in the header file, that is not the common practice. Rather, it is common coding practice to declare functions in a header file but write function definitions in a separate source file. The file MyMatrixMath.h declares to the compiler that the function printMatrix() exists. It lets the compiler know that the output data type of printMatrix() is extern void. The keyword extern means that the function printMatrix() is available to be used by other files that include the MyMatrixMath.h library, and void means that it does not return any outputs.

The source file MyMatrixMath.cpp purposefully shares the same name as its header file. It includes the function definitions for all functions that are declared in the header MyMatrixMath.h file. Any file that will use the MyMatrixMath functions must use the code #include "MyMatrixMath.h" to tell the compiler that the MyMatrixMath functions exists.

16.2.4 The vector Library

The vector library is very helpful when array and matrix sizes could vary. The two-dimensional matrix A is the input argument to the printMatrix() function in MyMatrixMath.cpp. Its data type is `vector<vector<double>>`. More precisely, A is a vector of vectors, i.e., A is a vector in which each element is a

vector of double precision numbers. The code `(int)A.size()` returns the number of rows in the A matrix as an integer. The code `(int)A[0].size()` returns the number of columns. With information about the number of rows and columns, the `printMatrix()` function can write two-dimensional matrices of any size to the console. In addition to `size()`, the vector library has many other useful functions. You can learn more about them at this website: <https://cplusplus.com/reference/vector/vector/> (accessed February 2023).

16.3 The MyMatrixMath Library

Modify your (.cpp) and (.h) files as shown in the code below. Try to understand how each function works.

16.3.1 MyMatrixMath.h

```
#pragma once
#include <vector> //vector type

using namespace std; //To use vector without std::vector

// Declare the printMatrix function
extern void printMatrix(vector<vector<double>> A);

// Function to multiply two matrices of any compatible sizes
extern vector<vector<double>> MatrixMultiply(const vector<vector<double>> A,
                                               const vector<vector<double>>);

//Function that returns the transpose of a matrix of any size C=transpose(A)
extern vector<vector<double>> transpose(const vector<vector<double>> A);

//Function that creates an nxn identity matrix I
extern vector<vector<double>> Identity(int n);

//This function splits the matrix "A" into two parts: rows above "index"
// are untouched, rows below "index" are searched for the row with the
// largest value in the "index" column. The row having the largest value
// in the "index" column is moved up to the "index" row. The other
// searched rows are moved below it.
extern vector<vector<double>> SortLargestRowToTop(
    const vector<vector<double>> A, const int index);

//This function is part of the row-reduction matrix solver that
// can create an upper triangular matrix.
// It multiplies one row by a factor and adds it to another
// to eliminate the value in "startCol" from the "ReduceRow"
extern vector<vector<double>> ReduceOneRow(const vector<vector<double>> A,
                                              const int KeepRow, const int startCol, const int ReduceRow);

//This function uses Gaussian Elimination to solve a matrix
// inversion problem that is given in Reduced Row Echelon form
//The A matrix must be in row echelon form [u,B] where
// u is an nxn upper triangular matrix and B is an nxp solution matrix
```

16.3 The MyMatrixMath Library

```
extern vector<vector<double>> SolveRowEchelonMatrix(
    const vector<vector<double>> A);

//This function solves the matrix problem A*x = B for x.
// It uses Gaussian elimination to find x = A^-1*B and returns x
// as "SolvedMat"
extern vector<vector<double>> LeftInverseSolve(
    const vector<vector<double>> A, const vector<vector<double>> B);

//This function calculates the inverse of the matrix A by solving the
// equation A*x=I for x = A^-1*I where I is the identity matrix
extern vector<vector<double>> Inverse(const vector<vector<double>> A);

//This function uses the power law to calculate the matrix exponential of A
// expm(A) = I + A + A^2/2 + A^3/3! + A^4/4! + ...
extern vector<vector<double>> expm(const vector<vector<double>> A);

//This function multiplies a matrix by a scalar B = scalar*A
extern vector<vector<double>> MatrixScalarMult(const vector<vector<double>> A,
    const double scalar);

//This function adds two matrices of the same size C = A+B
extern vector<vector<double>> MatrixAdd(const vector<vector<double>> A,
    const vector<vector<double>> B);
```

16.3.2 MyMatrixMath.cpp

```
#include <iostream> //cout, endl
#include <vector> //vector
#include "MyMatrixMath.h" //Header file we create for matrix math

using namespace std; //To use cout and endl without std::cout and std::endl

//Function to print the matrix to the console window
extern void printMatrix(vector<vector<double>> A) {
    for (int ii = 0; ii < (int)A.size(); ii++) {
        for (int jj = 0; jj < (int)A[0].size(); jj++) {
            cout << A[ii][jj] << " "; //Print the matrix to the console
        }
        cout << endl; //Go to the next row of the matrix
    }
}

// Function to multiply two matrices of any compatible sizes
extern vector<vector<double>> MatrixMultiply(
    const vector<vector<double>> A, //INPUT: C = A*B
    const vector<vector<double>> B //INPUT: C= A*B
) {
```

```

//Declare the output matrix C and give it the correct dimensions
vector<vector<double>> C(A.size(), vector<double>(B[0].size()));

//check that matrix inner dimensions are compatible
if (A[0].size() != B.size()) {
    //The matrices are not compatible
    cout << "Error: Number of columns of A must equal number of rows of B for C = A*B" << endl;
    throw 1; //Throw an error
}

//Perform the matrix multiplication
for (int ii = 0; ii < A.size(); ii++) {
    for (int jj = 0; jj < B[0].size(); jj++) {
        C[ii][jj] = 0.0;
        for (int kk = 0; kk < A[0].size(); kk++) {
            C[ii][jj] += A[ii][kk] * B[kk][jj];
        }
    }
}

//Return the matrix C=A*B
return C;
}

//Function that returns the transpose of a matrix of any size C=transpose(A)
extern vector<vector<double>> transpose(
    const vector<vector<double>> A
) {

//Declare the output matrix C and give it the correct dimensions
vector<vector<double>> C(A[0].size(), vector<double> (A.size()));

//Transpose the matrix by swapping rows with columns
for (int ii = 0; ii < C.size(); ii++) {
    for (int jj = 0; jj < C[0].size(); jj++) {
        C[ii][jj] = A[jj][ii];
    }
}

//Return the transposed matrix C=transpose(A)
return C;
}

//Function that creates an nxn identity matrix I
extern vector<vector<double>> Identity(int n) {

//Declare the variable I and set its dimensions to be nxn
vector<vector<double>> I(n, vector<double>(n, 0.0)); //initialize a nxn matrix to all 0.0
for (int ii = 0; ii < n; ii++) {
    //Set the diagonal elements to be ones
    I[ii][ii] = 1.0;
}
}

```

16.3 The MyMatrixMath Library

```
//Return the nxn identity matrix
return I;
}

//This function splits the matrix "A" into two parts: rows above "index"
// are untouched, rows below "index" are searched for the row with the
// largest value in the "index" column. The row having the largest value
// in the "index" column is moved up to the "index" row. The other
// searched rows are moved below it.
extern vector<vector<double>> SortLargestRowToTop(
    const vector<vector<double>> A,
    const int index
) {

    int n = (int)A.size(); //Number of rows in A
    int m = (int)A[0].size(); //Number of columns in A

    //Declare the output matrix C, and set its dimensions
    // to the dimensions of A
    vector<vector<double>> C(n, vector<double>(m));

    //Only search rows whose indices are larger than "index"
    int indexMax = index;
    //find the row in "A" with the largest value in the "index" column
    for (int ii = index+1; ii < n; ii++) {
        double absA = abs(A[ii][index]);
        double maxVal = abs(A[indexMax][index]);
        if (absA > maxVal) {
            //A larger value has been found, update the max row index
            indexMax = ii;
        }
    }

    //Move the largest to the "index" row
    for (int ii = 0; ii < n; ii++) {
        for (int jj = 0; jj < m; jj++) {
            if (ii < index)
                //Leave the row untouched if its index is smaller
                // than the "index" row
                C[ii][jj] = A[ii][jj];
            else if (ii < indexMax)
                //If a searched row had an "index" value smaller than the
                // largest value in the "index" column, move it below the
                // "index" row
                C[ii + 1][jj] = A[ii][jj];
            else if (ii == indexMax)
                //Make the row with the largest value in the "index" column
                // the "index" row
                C[index][jj] = A[ii][jj];
            else
                //These rows were already below the "index" row, and they
                // had smaller values than the largest value in the "index"
```

```

    // column.  They do not need to be moved.
    C[ii][jj] = A[ii][jj];
}

}

//Return the sorted matrix
return C;
}

//This function is part of the row-reduction matrix solver that
// can create an upper triangular matrix.
// It multiplies one row by a factor and adds it to another
// to eliminate the value in "startCol" from the "ReduceRow"
extern vector<vector<double>> ReduceOneRow(
    const vector<vector<double>> A,
    const int KeepRow,
    const int startCol,
    const int ReduceRow
) {

    //Find the factor to eliminate the value in "startCol" from "ReduceRow"
    double factor = -A[ReduceRow][startCol] / A[KeepRow][startCol];
    vector<vector<double>> C = A; //Set the reduced matrix to be the A matrix
    for (int jj = startCol; jj < A[KeepRow].size(); jj++) {
        //Remove the value in "startCol" from "ReduceRow"
        C[ReduceRow][jj] = factor * A[KeepRow][jj] + A[ReduceRow][jj];
    }

    //Return the reduced row
    return C;
}

//This function uses Gaussian Elimination to solve a matrix
// inversion problem that is given in Reduced Row Echelon form
//The A matrix must be in row echelon form [u,B] where
// u is an nxn upper triangular matrix and B is an nxp solution matrix
extern vector<vector<double>> SolveRowEchelonMatrix(
    const vector<vector<double>> A
) {

    int n = (int)A.size(); //Size of the square matrix
    int m = (int)A[0].size(); //# of columns of Row Echelon Matrix
    int p = m - n; //# of columns of the solution matrix

    //Check that solution matrix has at least one column
    if (p < 1) {
        cout << "Matrix Size Issue" << endl;
        throw 1;
    }

    //separate the augmented matrix into its upper triangular and
    //solution form
    vector<vector<double>> u(n, vector<double>(n)); //nxn upper triangular matrix
}

```

16.3 The MyMatrixMath Library

```
vector<vector<double>> B(n, vector<double>(p)); //nxp solution matrix

//Extract the upper triangular matrix u from the row echelon matrix A=[u,B]
for (int ii = 0; ii < n; ii++)
    for (int jj = 0; jj < n; jj++)
        u[ii][jj] = A[ii][jj];

//Extract the solution matrix B from the row echelon matrix A=[u,B]
for (int ii = 0; ii < n; ii++)
    for (int jj = 0; jj < p; jj++)
        B[ii][jj] = A[ii][jj + n];

//Declare the matrix C = u^(-1)*B, and set its dimensions
vector<vector<double>> C(n, vector<double>(p));

//Start at the bottom row, use back-substitution to solve for each value of C
//This is the heart of the algorithm
//(https://en.wikipedia.org/wiki/Gaussian_elimination)
for (int ii = 0; ii < n; ii++) {
    int aa = n - ii - 1;
    for (int jj = 0; jj < p; jj++) {
        C[aa][jj] = B[aa][jj];
        for (int kk = aa+1; kk < n; kk++) {
            C[aa][jj] -= u[aa][kk] * C[kk][jj];
        }
        C[aa][jj] /= u[aa][aa];
    }
}

//Return the solution C=u(^-1)*B
return C;
}

//This function solves the matrix problem A*x = B for x.
// It uses Gaussian elimination to find x = A^-1*B and returns x
// as "SolvedMat"
extern vector<vector<double>> LeftInverseSolve(
    const vector<vector<double>> A,
    const vector<vector<double>> B
) {

    //Make sure that the matrix sizes are compatible
    if (A.size() != A[0].size() ||
        A.size() != B.size()) {
        cout << "Error: Matrix size problems" << endl;
        throw 1;
    }

    //Get the matrix sizes. A is nxn, B is nxm
    int n = (int)A.size();
    int m = n + (int)B[0].size();

    //Put together a big matrix BigMat = [A,B]
```

```

// that will eventually be reduced to row echelon form [u,b]
// where u is an upper triangular matrix and b is a solution matrix
vector<vector<double>> BigMat(B.size(), vector<double>(m));
for (int ii = 0; ii < n; ii++) {
    for (int jj = 0; jj < m; jj++) {
        if (jj < n)
            BigMat[ii][jj] = A[ii][jj];
        else
            BigMat[ii][jj] = B[ii][jj - n];
    }
}

//Pivot the row with the largest first element to the top of the big matrix
vector<vector<double>> SortedMat = SortLargestRowToTop(BigMat, 0);

//Perform Gaussian elimination with partial pivoting
for (int ii = 0; ii < n-1; ii++) {
    for (int jj = ii+1; jj < n; jj++) {
        //Row reduction
        SortedMat = ReduceOneRow(SortedMat, ii, ii, jj);
    }
    //Row pivoting
    SortedMat = SortLargestRowToTop(SortedMat, ii + 1);
}

//Use back substitution to solve the [u,b] matrix
vector<vector<double>> SolvedMat = SolveRowEchelonMatrix(SortedMat);

//Return the solution: SolvedMat = A^-1*B
return SolvedMat;
}

//This function calculates the inverse of the matrix A by solving the
// equation A*x=I for x = A^-1*I
// where I is the identity matrix
extern vector<vector<double>> Inverse(const vector<vector<double>>&A) {
    return LeftInverseSolve(A, Identity((int)A.size()));
}

//This function uses the power law to calculate the matrix exponential of A
// expm(A) = I + A + A^2/2 + A^3/3! + A^4/4! + ...
extern vector<vector<double>> expm(const vector<vector<double>>&A) {
    //Get the size of the square A matrix
    int n = A.size();
    int m = A[0].size();
    if (m != n) {
        cout << "Matrix must be square " << endl;
        throw 1;
    }

    //Start with the identity matrix
    vector<vector<double>> eAin = Identity(n);
}

```

16.3 The MyMatrixMath Library

```
//Set the maximum number of elements in the power law summation
const int maxIter = 50;

//Declare a temporary matrix
vector<vector<double>> eAout(n, vector<double>(n));

//Start with the identity matrix
vector<vector<double>> expA = Identity(n);

for (int kk = 1; kk < maxIter; kk++) {
    //eAout = eAin * A = A^kk / (kk-1)!
    eAout = MatrixMultiply(eAin, A);
    for (int ii = 0; ii < n; ii++) {
        for (int jj = 0; jj < n; jj++) {
            //eAin = eAout / kk = A^kk / kk!
            eAin[ii][jj] = eAout[ii][jj] / ((double)kk);
            //expA = expA + A^kk / kk!
            expA[ii][jj] += eAin[ii][jj]; //expm(A)
        }
    }
}

//Return the matrix exponential of A
return expA;
}

//This function multiplies a matrix by a scalar B = scalar*A
extern vector<vector<double>> MatrixScalarMult(const vector<vector<double>> A,
    const double scalar) {

    int n = A.size();
    int m = A[0].size();
    vector<vector<double>> B(n, vector<double>(m));

    //Multiply each element of the matrix by the scalar
    for (int ii = 0; ii < n; ii++)
        for (int jj = 0; jj < m; jj++)
            B[ii][jj] = scalar * A[ii][jj];

    return B;
}

//This function adds two matrices of the same size C = A+B
extern vector<vector<double>> MatrixAdd(const vector<vector<double>> A,
    const vector<vector<double>> B) {

    //Declare the output matrix C and give it the correct dimensions
    vector<vector<double>> C(A.size(), vector<double>(A[0].size()));

    //check that matrix inner dimensions are compatible
    if (A.size() != B.size() || A[0].size() != B[0].size()) {
        //The matrices are not compatible
        cout << "Error: Matrix Sizes for C=A+B are Incompatible" << endl;
    }
}
```

```

    throw 1; //Throw an error
}

//Add the matrices element-by-element
for (int ii = 0; ii < C.size(); ii++)
    for (int jj = 0; jj < C[0].size(); jj++)
        C[ii][jj] = A[ii][jj] + B[ii][jj];

//Return the result C=A+B
return C;
}

```

Wow! Creating a C++ matrix math library is a lot of work!

16.4 Solving $\dot{x} = Ax + Bu$

The MyMatrixMath library can help solve dynamic equations like $\dot{x} = Ax + Bu$. To highlight this, consider the state-space system

$$\dot{x} = \begin{bmatrix} 0 & 1 \\ -1 & -2 \end{bmatrix}x + \begin{bmatrix} 0 \\ 1 \end{bmatrix}u \quad (16.1)$$

$$y = \begin{bmatrix} 1 & 0 \end{bmatrix}x \quad (16.2)$$

with the initial condition

$$x(0) = \begin{bmatrix} -2 \\ 1 \end{bmatrix} \quad (16.3)$$

the time-step

$$\Delta t = 0.1 \quad (16.4)$$

and the input sequence

$$u = \begin{bmatrix} 1, 2, 3, 4, 5 \end{bmatrix} \quad (16.5)$$

Example code is provided in the CppCrashCoursePart2.cpp file below to solve it.

16.4.1 CppCrashCoursePart2.cpp

```

#include <iostream> //cout, endl
#include <vector> //vector type
#include "MyMatrixMath.h" //Include our MyMatrixMath library

using namespace std; //To use cout, endl, and vector without std::

//The program's main function
int main() {

```

16.4 Solving $\dot{x} = Ax + Bu$

```
//Declare and define the A matrix
vector<vector<double>> A{ {0.0,1.0},{-1.0,-2.0} };

//Declare and define the B matrix
vector<vector<double>> B{ {0.0},{1.0} };

//Declare and define the C matrix
vector<vector<double>> C{ {1.0,0.0} };

//Declare and define the u vector
vector<double> u{ 1.0,2.0,3.0,4.0,5.0 };

//Declare and define the initial condition
vector<vector<double>> x{ {-2.0},{1.0} };

//Declare and define the time-step
double dt = 0.1;

//Create a temporary vector Adt = A*dt
vector<vector<double>> Adt = MatrixScalarMult(A, dt);

//Create a temporary vector Bdt = B*dt
vector<vector<double>> Bdt = MatrixScalarMult(B, dt);

//Create a temporary vector F = [A*dt,B*dt;zeros(1,3)];
vector<vector<double>> F(3, vector<double>(3));
for (int ii = 0; ii < 3; ii++) {
    for (int jj = 0; jj < 3; jj++) {
        if (ii < 2)
            if (jj < 2)
                F[ii][jj] = Adt[ii][jj];
            else
                F[ii][jj] = Bdt[ii][0];
        else
            F[ii][jj] = 0.0;
    }
}

//Declare the Fd matrix and give it the correct dimensions 3x3
vector<vector<double>> Fd(3, vector<double>(3));

//Calculate the Fd matrix: Fd = expm([A*dt, B*dt; zeros(1,3)])
Fd = expm(F);

//Declare the Ad matrix and give it the correct dimensions 2x2
vector<vector<double>> Ad(2, vector<double>(2));

//Extract the Ad matrix from the Fd Matrix
for (int ii = 0; ii < 2; ii++)
    for (int jj = 0; jj < 2; jj++)
        Ad[ii][jj] = Fd[ii][jj];
```

```

//Declare the Bd matrix and give it the correct dimensions 2x1
vector<vector<double>> Bd(2, vector<double>(1));

//Extract the Bd matrix from the Fd Matrix
for (int ii = 0; ii < 2; ii++)
    for (int jj = 0; jj < 1; jj++)
        Bd[ii][jj] = Fd[ii][jj+2];

//Allocate memory (5 elements) for the output
vector<vector<double>> y(1, vector<double>(5));

//Solve the state and output equations
for (int ii = 0; ii < u.size(); ii++) {
    //Declare a temporary matrix so the output data type is correct
    vector<vector<double>> yii(1, vector<double>(1));

    //Solve the output equation
    yii = MatrixMultiply(C, x);
    y[0][ii] = yii[0][0];

    //Solve the state equation
    x = MatrixAdd(MatrixMultiply(Ad, x), MatrixScalarMult(Bd, u[ii]));
}

//Print the output y
cout << "y = " << endl;
printMatrix(y);

//Return 0 to indicate that the main function was successful
return 0;
}

```

If everything was coded and set up correctly, the output to the console window is



The screenshot shows the Microsoft Visual Studio Debug Console window. The title bar says "Microsoft Visual Studio Debug Console". The console area displays the output of the program: "y = -2 -1.89548 -1.77901 -1.64474 -1.48808". The window has standard minimize, maximize, and close buttons at the top right.

The code below shows how the same problem would be solved in MATLAB.

```

clear
A = [0,1;-1,-2];
B = [0;1];
C = [1,0];
u = 1:5;
dt = 0.1;
x = [-2;1];
Fd = exp([A*dt,B*dt;zeros(1,3)]);
Ad = Fd(1:2,1:2);
Bd = Fd(1:2,3);

```

16.4 Solving $\dot{x} = Ax + Bu$

```
for ii = 1:5
    y(ii) = C*x;
    x = Ad*x+Bd*u(ii);
end
y
```

The MATLAB program used far fewer lines of code to solve the same state-space system. Did it give the same answer?

16.4.2 Do It Yourself

Create a new function named `getFd()` in the `MyMatrixMath.cpp` file that takes the matrices `A` and `B` and the time-step `dt` and returns the `Fd` matrix. The function should work for any correctly sized `A` and `B` matrices. Then, in the `CppCrashCoursePart2.cpp` file, write code that uses your new function to solve the following state-space system:

$$\dot{x} = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ -6 & -7 & -8 \end{bmatrix} x + \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} u \quad (16.6)$$

$$y = \begin{bmatrix} 1 & 0 & 0 \end{bmatrix} x + 0.8u \quad (16.7)$$

with the initial condition

$$x(0) = \begin{bmatrix} -2 \\ -1 \\ 3 \end{bmatrix} \quad (16.8)$$

the time-step

$$\Delta t = 0.1 \quad (16.9)$$

and the input sequence

$$u = \begin{bmatrix} 4, 3, 2, 1, 0, -1 \end{bmatrix} \quad (16.10)$$

If available, use MATLAB to check that your C++ program solved the state-space equations correctly. This is the end of C++ Crash Course 2.

Chapter 17

C++ Crash Course 3

Contents

17.1 Classes in C++	486
17.2 The MyMatrixClass Class	486
17.2.1 MyMatrixClass.h	487
17.2.2 MyMatrixClass.cpp	488
17.2.3 CppCrashCoursePart3.cpp	491
17.2.4 How it Works	492
17.2.5 Do It Yourself	493
17.3 Getting and Setting Private Members	493
17.3.1 MyMatrixClass.h	494
17.3.2 MyMatrixClass.cpp	495
17.4 Solving $\dot{x} = Ax + Bu$	500
17.4.1 CppCrashCoursePart3.cpp	500
17.4.2 Do It Yourself	502

This third C++ Crash Course teaches the following concepts:

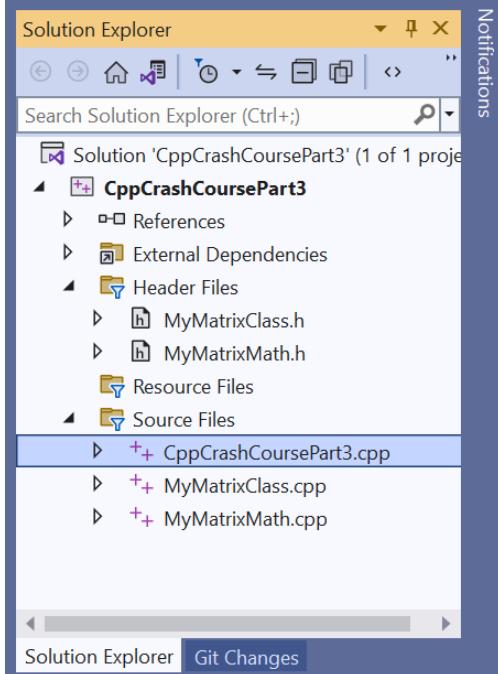
1. How to use classes in C++ for object-oriented programming
2. How to overload operators in C++ classes
3. How to create and use class member functions
4. How to solve the state-space system $\dot{x} = Ax + Bu$

This course builds on the previous two C++ crash courses. You will need to use some of the computer code that you generated in the past courses. As you have done in the past, create a new solution in Visual Studio. Add header and source files until your Solution Explorer includes the following files:

- MyMatrixClass.h
- MyMatrixClass.cpp

- MyMatrixMath.h (Same as C++ Crash Course 2)
- MyMatrixMath.cpp (Same as C++ Crash Course 2)
- CppCrashCoursePart3.cpp

Your solution explorer should appear as shown below:



You should have already created the files MyMatrixMath.h and MyMatrixMath.cpp in C++ Crash Course 2. If not, you will need to complete that crash course before doing this one.

17.1 Classes in C++

Classes in C++ enable object-oriented programming. Objects have attributes and methods. Methods are also called class member functions. For example, a circle is an object. Its attributes could include its radius, its origin, its fill-color, and its edge-color. A circle's methods could include the equation to calculate its area or the equation to calculate its circumference. Creating a circle class in C++ defines a new data type. As a reminder, double, int, float, char, and vector are examples of data types. Any variables declared by the circle data type would have all the attributes and methods of the circle object that are defined in the circle class.

17.2 The MyMatrixClass Class

In this C++ crash course, you will create a new matrix class named MyMatrixClass. MyMatrixClass defines a new data type for matrices. Matrices in MyMatrixClass have the following attributes:

- nRows

17.2 The MyMatrixClass Class

- nCols
- myMatrix

The attribute nRows is an int data type that defines the number of rows in the matrix. The int nCols defines the number of columns in the matrix. The attribute myMatrix is of data type `vector<vector<double>>`. It contains all the data in the rows and columns of the matrix.

MyMatrixClass objects will have the following methods:

- Constructor
- Destructor
- Matrix addition
- Matrix subtraction
- Matrix multiplication
- Matrix inversion solver
- Matrix-scalar division
- **Matrix-scalar multiplication (you create this one)**
- Matrix element accessing
- Matrix print
- **Matrix transpose (you create this one)**

Instead of copying the code in this assignment, it is recommended that you type it. Typing it helps you learn better and may prevent compiler errors that are caused by incompatible font formats with Visual Studio. MyMatrixClass is declared in the MyMatrixClass.h file:

17.2.1 MyMatrixClass.h

```
#pragma once //Tell the compiler to only compile this file once

#include <vector> //vector
using namespace std; //To use vector, cout, and endl without std::

//Declare the MyMatrixClass class
class MyMatrixClass
{
public:
    //Constructor for the MyMatrixClass object that initializes every element
    // of the matrix to init
    MyMatrixClass(int rows, int cols, double init);
```

```

//Alternate constructor for the MyMatrixClass that allows it to
// be initialized to a specific matrix A
MyMatrixClass(vector<vector<double>> A);

//MyMatrixClass destructor to delete the class when the program ends
~MyMatrixClass();

//Overload operators to perform matrix operations more easily
MyMatrixClass operator + (MyMatrixClass& B); //Matrix+Matrix addition
MyMatrixClass operator - (MyMatrixClass& B); //Matrix-Matrix subtraction
MyMatrixClass operator * (MyMatrixClass& B); //Matrix*Matrix multiplication
MyMatrixClass operator | (MyMatrixClass& B); //A|B = A^(-1)*B Matrix Solve

//Overload operators to make matrix / scalar operations easier
MyMatrixClass operator + (double& b); //Matrix+scalar addition
MyMatrixClass operator - (double& b); //Matrix-scalar subtraction
MyMatrixClass operator / (double& b); //Matrix/scalar division

//Overload operators to make accessing elements of a matrix easier
double& operator () (int n, int m);

//Create a function to print the matrix
void print();

private:
    //Matrix attributes
    vector<vector<double>> myMatrix; //The matrix data
    int nRows; //Number of rows in the matrix
    int nCols; //Number of columns in the matrix
};

```

17.2.2 MyMatrixClass.cpp

```

#include <vector> //vector
#include "MyMatrixClass.h" //Where MyMatrixClass is declared
#include "MyMatrixMath.h" //MatrixAdd, MatrixMultiply, LeftInverseSolve
// MatrixScalarMult, printMatrix

using namespace std; //To use vector, cout, and endl without std::

//Constructor for the MyMatrixClass object that initializes every element
// of the matrix to init
MyMatrixClass::MyMatrixClass(int rows, int cols, double init)
{
    //Set the number of rows and columns in the matrix
    this->nRows = rows;
    this->nCols = cols;
}

```

17.2 The MyMatrixClass Class

```
//Initialize all values of the matrix to init
this->myMatrix.resize(rows); //Give the matrix the right # of rows
for (int ii = 0; ii < rows; ii++) {
    this->myMatrix[ii].resize(cols); //Give the matrix the right # of columns
    for (int jj = 0; jj < cols; jj++) {
        this->myMatrix[ii][jj] = init;
    }
}
}

//Alternate constructor for the MyMatrixClass that allows it to
//be initialized to a specific matrix A
MyMatrixClass::MyMatrixClass(vector<vector<double>> A)
{
    //Set the number of rows and columns in the matrix
    this->nRows = A.size();
    this->nCols = A[0].size();

    //Initialize all values of the matrix to init
    this->myMatrix.resize(this->nRows); //Give the matrix the right # of rows
    for (int ii = 0; ii < this->nRows; ii++) {
        this->myMatrix[ii].resize(this->nCols); //Give the matrix the right # of columns
        for (int jj = 0; jj < this->nCols; jj++) {
            //Copy A to myMatrix
            this->myMatrix[ii][jj] = A[ii][jj];
        }
    }
}

//Destructor...Deletes the MyMatrixClass object when the program terminates
MyMatrixClass::~MyMatrixClass()
{
}

//Matrix addition operator overload method
MyMatrixClass MyMatrixClass::operator+(MyMatrixClass& B)
{
    //Create the output matrix C=A+B and initialize it to zero
    MyMatrixClass C(nRows, nCols, 0.0);

    //Perform the matrix addition
    C.myMatrix = MatrixAdd(this->myMatrix, B.myMatrix);

    //Return the output
    return C;
}

//Matrix subtraction operator overload method
MyMatrixClass MyMatrixClass::operator-(MyMatrixClass& B)
{
    //Create the output matrix C=A-B and initialize it to zero
    MyMatrixClass C(nRows, nCols, 0.0);
```

```

//Perform the matrix subtraction
for (int ii = 0; ii < nRows; ii++)
    for (int jj = 0; jj < nCols; jj++)
        C.myMatrix[ii][jj] = this->myMatrix[ii][jj] - B.myMatrix[ii][jj];

//Return the result C=A-B
return C;
}

//Matrix multiplication operator overload method
MyMatrixClass MyMatrixClass::operator*(MyMatrixClass& B)
{
    //Create the output matrix C=A*B and initialize it to zero
    MyMatrixClass C(this->nRows, B.nCols, 0.0);

    //Perform the matrix multiplication
    C.myMatrix = MatrixMultiply(this->myMatrix, B.myMatrix);

    //Return the result C=A*B
    return C;
}

//Matrix inversion solver operator overload method
MyMatrixClass MyMatrixClass::operator|(MyMatrixClass& B)
{
    //Create the output matrix C=A\B (C=A^-1*B) and initialize it to zero
    MyMatrixClass C(this->nRows, B.nCols, 0.0);

    //Solve the matrix equation to get C = A^(-1)*B
    C.myMatrix = LeftInverseSolve(this->myMatrix, B.myMatrix);

    //Return the result C=A\B
    return C;
}

//Matrix+scalar addition operator overload method
MyMatrixClass MyMatrixClass::operator+(double& b)
{
    //Create the output matrix C and initialize it to zero
    MyMatrixClass C(nRows, nCols, 0.0);

    //Perform the matrix+scalar addition
    for (int ii = 0; ii < nRows; ii++)
        for (int jj = 0; jj < nCols; jj++)
            C.myMatrix[ii][jj] = this->myMatrix[ii][jj] + b;

    //Return the result C = A+b
    return C;
}

//Matrix-scalar subtraction operator overload method
MyMatrixClass MyMatrixClass::operator-(double& b)

```

17.2 The MyMatrixClass Class

```
{  
    //Create the output matrix C and initialize it to zero  
    MyMatrixClass C(nRows, nCols, 0.0);  
  
    //Perform the matrix-scalar subtraction  
    for (int ii = 0; ii < nRows; ii++)  
        for (int jj = 0; jj < nCols; jj++)  
            C.myMatrix[ii][jj] = this->myMatrix[ii][jj] - b;  
  
    //Return the result C = A-b  
    return C;  
}  
  
//Matrix/scalar division operator overload method  
MyMatrixClass MyMatrixClass::operator/(double& b)  
{  
    //Create the output matrix C and initialize it to zero  
    MyMatrixClass C(nRows, nCols, 0.0);  
  
    //Perform the matrix/scalar division  
    for (int ii = 0; ii < nRows; ii++)  
        for (int jj = 0; jj < nCols; jj++)  
            C.myMatrix[ii][jj] = this->myMatrix[ii][jj] / b;  
  
    //Return the result C = A/b  
    return C;  
}  
  
//Operator overload method that allows access to matrix elements A(n,m)  
double& MyMatrixClass::operator()(int n, int m)  
{  
    //Return the value in row n column m of the matrix  
    return this->myMatrix[n][m];  
}  
  
//Class member function to print the matrix to the console  
void MyMatrixClass::print()  
{  
    //Use the printMatrix function to print the matrix to the console  
    printMatrix(this->myMatrix);  
}
```

17.2.3 CppCrashCoursePart3.cpp

```
#include <iostream> //cout, endl  
#include "MyMatrixClass.h" //MyMatrixClass  
using namespace std; //To use cout and endl without std::
```

```

int main()
{
    //Declare and define the A and B matrices
    MyMatrixClass A({
        {1.0,-2.0,3.0,4.0},
        {5.0,6.0,-7.0,8.0},
        {9.0,10.0,11.0,-12.0},
        {13.0,14.0,15.0,16.0} });
    MyMatrixClass B({
        {1.0,5.0},
        {2.0,6.0},
        {3.0,7.0},
        {4.0,8.0} });

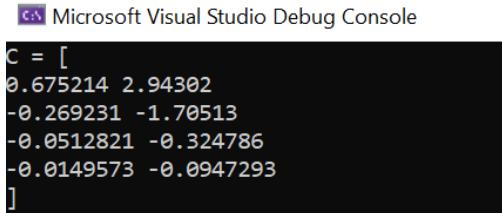
    //Solve C = A^-1 * B
    MyMatrixClass C = A | B;

    //Print the output matrix C
    cout << "C = [" << endl;
    C.print();
    cout << "]" << endl;

    return 0;
}

```

Compile and run the code. The console window should appear as follows:



Microsoft Visual Studio Debug Console

```
C = [
0.675214 2.94302
-0.269231 -1.70513
-0.0512821 -0.324786
-0.0149573 -0.0947293
]
```

If not, fix any errors and make sure your Solution Explorer includes the files discussed above.

17.2.4 How it Works

We will discuss certain parts of the files listed above. The `MyMatrixClass.h` file declares the `MyMatrixClass` class. The class has public and private attributes and methods. Private attributes and methods can only be accessed by `MyMatrixClass`. For example, they can be accessed within the file `MyMatrixClass.cpp`, but they cannot be accessed from `CppClassCrashCourse3.cpp` or `MyMatrixMath.cpp`. Public attributes and methods can be accessed within other files that have `#include "MyMatrixClass.h"`.

The functions `MyMatrixClass::MyMatrixClass(int rows, int cols, double init)` and `MyMatrixClass::MyMatrixClass(vector<vector<double> A)` are called constructors. A class can have multiple constructors. The constructor builds the object using the arguments passed to it. For example, the constructor builds a matrix with `rows` rows and `cols` columns and initializes all values of the matrix to the same value: `init`. On the other hand, the constructor `MyMatrixClass::MyMatrixClass(vector<vector<double> A)` builds a matrix equal to the matrix `A` passed to it.

17.3 Getting and Setting Private Members

The function `MyMatrixClass:: MyMatrixClass()` is called a destructor. It deletes the `MyMatrixClass` objects when the program terminates.

Because `MyMatrixClass` defines a new data type, if we want to use operators, we must define what they do. Defining what an operator does is called operator overloading. For example, matrix multiplication is different than scalar multiplication. Row elements of one matrix are multiplied by column elements of the other. The `*` operator is overloaded in the `MyMatrixClass` method `MyMatrixClass MyMatrixClass::operator*(MyMatrixClass& B)`.

You may have noticed that operators can be overloaded multiple times, causing different results. For example, the `+` operator is overloaded twice: once in `MyMatrixClass MyMatrixClass::operator+(MyMatrixClass& B)` and again in `MyMatrixClass MyMatrixClass::operator+(double& b)`. In the first method `MyMatrixClass MyMatrixClass::operator+(MyMatrixClass& B)`, it is overloaded for matrix + matrix addition. In the second method `MyMatrixClass MyMatrixClass::operator+(double& b)`, it is overloaded for matrix + scalar addition. If, in `CppCrashCoursePart3.cpp`, a matrix is added to another matrix the compiler will automatically use the method `MyMatrixClass MyMatrixClass::operator+(MyMatrixClass& B)`. If a matrix is added to a scalar, the compiler will automatically use the method `MyMatrixClass MyMatrixClass::operator+(double& b)` instead.

You may have noticed the keyword `this->`. It just means that the attribute belongs to `MyMatrixClass`. For example, `this->myMatrix` means that `myMatrix` belongs to `MyMatrixClass`. Using `this->myMatrix` is not strictly necessary within `MyMatrixClass.cpp`. We could simply write `myMatrix` instead.

Study the code carefully to understand how it works. Try out the different constructors and operators by calling them from `CppCrashCoursePart3.cpp`. See if you can understand how each works. If you need additional help, there are many good tutorials on C++ classes and object-oriented-programming (see for example https://www.w3schools.com/cpp/cpp_oop.asp, accessed Dec. 2022).

17.2.5 Do It Yourself

Modify the `MyMatrixClass` class in the following two ways:

1. Add an operator overload method to perform matrix-scalar multiplication of the matrix `A` and scalar `b` simply by using the `*` operator, i.e., $C = A * b$. The method must call the `MatrixScalarMult` function defined in `MyMatrixMath.cpp`. Use the matrix `MyMatrixClass A(1.0,-2.0,3.0,4.0);` and the scalar double `b = 0.5;`. Print the result to the console.
2. Create a new class member function named `transpose()`. It should cause a matrix `A` to be transposed by the code: `A.transpose()`. Use the matrix `MyMatrixClass A(1.0,-2.0,3.0,4.0);`. Print `A.transpose()` to the console.

17.3 Getting and Setting Private Members

It can sometimes be useful to grant access to the private members of a class. As a reminder, the private members of the `MyMatrixClass` class were `nRows`, `nCols`, and `myMatrix`. The following code shows changes to `MyMatrixClass.h` and `MyMatrixClass.cpp` to grant other files and functions access to either get or set the private members of the `MyMatrixClass` objects. Specifically, the class member function `getRows()` grants read-only (but not writing) privileges to the private member `nRows`. The function

`getCols()` grants read-only access to `nCols`, and `getMatrix()` grants read-only access to `myMatrix`. The class member function `setMatrix()` grants write-only access to the `myMatrix` private member.

Two functions were added to the `MyMatrixClass.cpp` file to show how these get and set functions are used. The function `matExp(A,B,Δt)` calculates the state-space solution matrix exponential F_d :

$$\begin{aligned} F_d &= \exp\left(\begin{bmatrix} A\Delta t & B\Delta t \\ \mathbf{0} & \mathbf{0} \end{bmatrix}\right) \\ &= \begin{bmatrix} A_d & B_d \\ \mathbf{0} & \mathbf{I} \end{bmatrix} \end{aligned}$$

where \mathbf{I} is the identity matrix, A_d is the discrete state-transition matrix, and B_d is the discrete input-transition matrix.

The function `stateSpace(& A_d, & B_d, A, B, Δt)` calls the `matExp(A,B,Δt)` function and returns the discrete state-transition matrices A_d and B_d .

The updated code for `MyMatrixClass.h` and `MyMatrixClass.cpp` is provided below:

17.3.1 MyMatrixClass.h

```
#pragma once //Tell the compiler to only compile this file once
#include <vector> //vector
#include "MyMatrixMath.h"
using namespace std; //To use vector, cout, and endl without std::
//Declare the MyMatrixClass class
class MyMatrixClass
{
public:
    //Constructor for the MyMatrixClass object that initializes every element
    // of the matrix to init
    MyMatrixClass(int rows, int cols, double init);
    //Alternate constructor for the MyMatrixClass that allows it to
    // be initialized to a specific matrix A
    MyMatrixClass(vector<vector<double>> A);
    //MyMatrixClass destructor to delete the class when the program ends
    ~MyMatrixClass();

    //Overload operators to perform matrix operations more easily
    MyMatrixClass operator + (MyMatrixClass& B); //Matrix+Matrix addition
    MyMatrixClass operator - (MyMatrixClass& B); //Matrix-Matrix subtraction
    MyMatrixClass operator * (MyMatrixClass& B); //Matrix*Matrix multiplication
    MyMatrixClass operator | (MyMatrixClass& B); //A|B = A^(-1)*B Matrix Solve

    //Overload operators to make matrix / scalar operations easier
    MyMatrixClass operator + (double& b); //Matrix+scalar addition
    MyMatrixClass operator - (double& b); //Matrix-scalar subtraction
    MyMatrixClass operator * (double& b); //Matrix-scalar multiplication
    MyMatrixClass operator / (double& b); //Matrix/scalar division
```

17.3 Getting and Setting Private Members

```
//Overload operators to make accessing elements of a matrix easier
double& operator () (int n, int m);

//Create a function to print the matrix
void print();
MyMatrixClass transpose();

//Get and Set functions
int getRows();
int getCols();
vector<vector<double>> getMatrix();
void setMatrix(vector<vector<double>> A);

private:
    //Matrix attributes
    vector<vector<double>> myMatrix; //The matrix data
    int nRows; //Number of rows in the matrix
    int nCols; //Number of columns in the matrix
};

//Function to calculate the Fd Matrix: Fd = expm(A*dt,B*dt;zeros(p,n+p));
MyMatrixClass matExp(MyMatrixClass A, MyMatrixClass B, const double dt);

//Function to calculate and return the state-transition matrices Ad and Bd
void stateSpace(MyMatrixClass& Ad, MyMatrixClass& Bd, MyMatrixClass A,
                MyMatrixClass B, const double dt);
```

17.3.2 MyMatrixClass.cpp

```
#include <vector> //vector
#include "MyMatrixMath.h" //MatrixAdd, MatrixMultiply, LeftInverseSolve
#include "MyMatrixClass.h" //Where MyMatrixClass is declared

// MatrixScalarMult, printMatrix
using namespace std; //To use vector, cout, and endl without std::
//Constructor for the MyMatrixClass object that initializes every element
// of the matrix to init
MyMatrixClass::MyMatrixClass(int rows, int cols, double init)
{
    //Set the number of rows and columns in the matrix
    this->nRows = rows;
    this->nCols = cols;

    //Resize the matrix to the correct size
    if ((int)this->myMatrix.size() != this->nRows)
        this->myMatrix.resize(this->nRows); //Give the matrix the right # of rows
    for (int ii = 0; ii < this->nRows; ii++) {
        if ((int)this->myMatrix[ii].size() != this->nCols)
```

```

        this->myMatrix[ii].resize(this->nCols); //Give the matrix the right # of columns
    }

    //Initialize all values of the matrix to init
    for (int ii = 0; ii < this->nRows; ii++) {
        for (int jj = 0; jj < this->nCols; jj++) {
            this->myMatrix[ii][jj] = init;
        }
    }
}

//Alternate constructor for the MyMatrixClass that allows it to
//be initialized to a specific matrix A
MyMatrixClass::MyMatrixClass(vector<vector<double>> A)
{
    //Set the number of rows and columns in the matrix
    this->nRows = (int)A.size();
    this->nCols = (int)A[0].size();

    //Resize myMatrix to the correct size
    if (this->myMatrix.size() != A.size())
        this->myMatrix.resize(A.size()); //Give the matrix the right # of rows
    for (int ii = 0; ii < A.size(); ii++) {
        if (this->myMatrix[ii].size() != A[0].size())
            this->myMatrix[ii].resize(A[0].size()); //Give the matrix the right # of columns
    }

    //Initialize all values of the matrix to init
    for (int ii = 0; ii < this->nRows; ii++) {
        for (int jj = 0; jj < this->nCols; jj++) {
            //Copy A to myMatrix
            this->myMatrix[ii][jj] = A[ii][jj];
        }
    }
}

//Destructor...Deletes the MyMatrixClass object when the program terminates
MyMatrixClass::~MyMatrixClass()
{
}

//Matrix addition operator overload method
MyMatrixClass MyMatrixClass::operator+(MyMatrixClass& B)
{
    //Create the output matrix C=A+B and initialize it to zero
    MyMatrixClass C(nRows, nCols, 0.0);
    //Perform the matrix addition
    C.myMatrix = MatrixAdd(this->myMatrix, B.myMatrix);
    //Return the output
    return C;
}

//Matrix subtraction operator overload method
MyMatrixClass MyMatrixClass::operator-(MyMatrixClass& B)

```

17.3 Getting and Setting Private Members

```
{  
    //Create the output matrix C=A-B and initialize it to zero  
    MyMatrixClass C(nRows, nCols, 0.0);  
    //Perform the matrix subtraction  
    for (int ii = 0; ii < nRows; ii++)  
        for (int jj = 0; jj < nCols; jj++)  
            C.myMatrix[ii][jj] = this->myMatrix[ii][jj] - B.myMatrix[ii][jj];  
    //Return the result C=A-B  
    return C;  
}  
//Matrix multiplication operator overload method  
MyMatrixClass MyMatrixClass::operator*(MyMatrixClass& B)  
{  
    //Create the output matrix C=A*B and initialize it to zero  
    MyMatrixClass C(this->nRows, B.nCols, 0.0);  
    //Perform the matrix multiplication  
    C.myMatrix = MatrixMultiply(this->myMatrix, B.myMatrix);  
    //Return the result C=A*B  
    return C;  
}  
//Matrix inversion solver operator overload method  
MyMatrixClass MyMatrixClass::operator|(MyMatrixClass& B)  
{  
    //Create the output matrix C=A\B (C=A^-1*B) and initialize it to zero  
    MyMatrixClass C(this->nRows, B.nCols, 0.0);  
    //Solve the matrix equation to get C = A^(-1)*B  
    C.myMatrix = LeftInverseSolve(this->myMatrix, B.myMatrix);  
    //Return the result C=A\B  
    return C;  
}  
//Matrix+scalar addition operator overload method  
MyMatrixClass MyMatrixClass::operator+(double& b)  
{  
    //Create the output matrix C and initialize it to zero  
    MyMatrixClass C(nRows, nCols, 0.0);  
    //Perform the matrix+scalar addition  
    for (int ii = 0; ii < nRows; ii++)  
        for (int jj = 0; jj < nCols; jj++)  
            C.myMatrix[ii][jj] = this->myMatrix[ii][jj] + b;  
    //Return the result C = A+b  
    return C;  
}  
//Matrix-scalar subtraction operator overload method  
MyMatrixClass MyMatrixClass::operator-(double& b)  
{  
    //Create the output matrix C and initialize it to zero  
    MyMatrixClass C(nRows, nCols, 0.0);  
    //Perform the matrix-scalar subtraction  
    for (int ii = 0; ii < nRows; ii++)  
        for (int jj = 0; jj < nCols; jj++)  
            C.myMatrix[ii][jj] = this->myMatrix[ii][jj] - b;  
    //Return the result C = A-b  
    return C;
```

```

}

MyMatrixClass MyMatrixClass::operator*(double& b)
{
    //Create the output matrix C and initialize it to zero
    MyMatrixClass C(this->nRows, this->nCols, 0.0);

    //Perform the matrix/scalar multiplication
    C.myMatrix = MatrixScalarMult(this->myMatrix, b);

    return C;
}
//Matrix/scalar division operator overload method
MyMatrixClass MyMatrixClass::operator/(double& b)
{
    //Create the output matrix C and initialize it to zero
    MyMatrixClass C(nRows, nCols, 0.0);
    //Perform the matrix/scalar division
    for (int ii = 0; ii < nRows; ii++)
        for (int jj = 0; jj < nCols; jj++)
            C.myMatrix[ii][jj] = this->myMatrix[ii][jj] / b;
    //Return the result C = A/b
    return C;
}
//Operator overload method that allows access to matrix elements A(n,m)
double& MyMatrixClass::operator()(int n, int m)
{
    //Return the value in row n column m of the matrix
    return this->myMatrix[n][m];
}
//Class member function to print the matrix to the console
void MyMatrixClass::print()
{
    //Use the printMatrix function to print the matrix to the console
    printMatrix(this->myMatrix);
}

MyMatrixClass MyMatrixClass::transpose()
{
    //Create the output matrix C and initialize it to zero
    MyMatrixClass C(nCols, nRows, 0.0);

    for (int ii = 0; ii < nCols; ii++)
        for (int jj = 0; jj < nRows; jj++)
            C.myMatrix[ii][jj] = this->myMatrix[jj][ii];

    return C;
}

int MyMatrixClass::getRows()
{
    return this->nRows;
}

```

17.3 Getting and Setting Private Members

```
int MyMatrixClass::getCols()
{
    return this->nCols;
}

vector<vector<double>> MyMatrixClass::getMatrix()
{
    return this->myMatrix;
}

void MyMatrixClass::setMatrix(vector<vector<double>> A)
{
    this->myMatrix = A;
}

MyMatrixClass matExp(MyMatrixClass A, MyMatrixClass B, const double dt)
{
    //ensure that the matrix sizes are compatible
    if (A.getRows() != A.getCols() || A.getRows() != B.getRows()) {
        throw 1;
    }

    MyMatrixClass Fd(A.getRows() + B.getCols(), A.getRows() + B.getCols(), 0.0);
    for (int ii = 0; ii < Fd.getRows(); ii++) {
        for (int jj = 0; jj < Fd.getCols(); jj++) {
            if (ii < A.getRows() && jj < A.getCols()) {
                Fd(ii, jj) = A(ii, jj) * dt;
            }
            else if (ii < A.getRows()) {
                Fd(ii, jj) = B(ii, jj-A.getCols()) * dt;
            }
            else {
                //do nothing
            }
        }
    }

    MyMatrixClass expFd(Fd.getRows(), Fd.getCols(), 0.0);
    expFd.setMatrix(expm(Fd.getMatrix()));

    return expFd;
}

void stateSpace(MyMatrixClass& Ad, MyMatrixClass& Bd, MyMatrixClass A,
    MyMatrixClass B, const double dt)
{
    //ensure that Ad and Bd are the correct sizes
    if (Ad.getRows() != A.getRows()
        || Ad.getCols() != A.getCols()
        || Bd.getRows() != B.getRows()
        || Bd.getCols() != B.getCols()) {
        throw 1;
    }
}
```

```

MyMatrixClass Fd = matExp(A, B, dt);

for (int ii = 0; ii < Ad.getRows(); ii++) {
    for (int jj = 0; jj < Fd.getCols(); jj++) {
        if (jj < Ad.getCols()) {
            Ad(ii, jj) = Fd(ii, jj);
        }
        else {
            Bd(ii, jj - Ad.getCols()) = Fd(ii, jj);
        }
    }
}
}

```

17.4 Solving $\dot{x} = Ax + Bu$

The following CppCrashCoursePart3.cpp code uses the updated MyMatrixClass.h and MyMatrixClass.cpp libraries to solve the following state-space system:

$$\dot{x} = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ -6 & -7 & -8 \end{bmatrix} x + \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} u \quad (17.1)$$

$$y = \begin{bmatrix} 1 & 0 & 0 \end{bmatrix} x + 0.8u \quad (17.2)$$

with the initial condition

$$x(0) = \begin{bmatrix} -2 \\ -1 \\ 3 \end{bmatrix} \quad (17.3)$$

the time-step

$$\Delta t = 0.1 \quad (17.4)$$

and the input sequence

$$u = [4, 3, 2, 1, 0, -1] \quad (17.5)$$

17.4.1 CppCrashCoursePart3.cpp

```

#include <iostream> //cout, endl
#include "MyMatrixClass.h" //MyMatrixClass
#include <vector>
using namespace std; //To use cout and endl without std::
int main()
{
    //Set the A and B matrices

```

17.4 Solving $\dot{x} = Ax + Bu$

```
MyMatrixClass A({ {0.0,1.0,0.0},{0.0,0.0,1.0},{-6.0,-7.0,-8.0} });
MyMatrixClass B(3, 1, 0.0);
B(2, 0) = 1.0;

//Set the initial condition for x
MyMatrixClass x(3, 1, 0.0);
x(0, 0) = -2.0;
x(1, 0) = -1.0;
x(2, 0) = 3.0;

//Set the time-step
double dt = 0.1;

//Set the input sequence
vector<double> u = { 4.0,3.0,2.0,1.0,0.0,-1.0 };

//Get the distrete state-transition matrices Ad and Bd
MyMatrixClass Ad(A.getRows(), A.getCols(), 0.0);
MyMatrixClass Bd(B.getRows(), B.getCols(), 0.0);
stateSpace(Ad, Bd, A, B, dt);

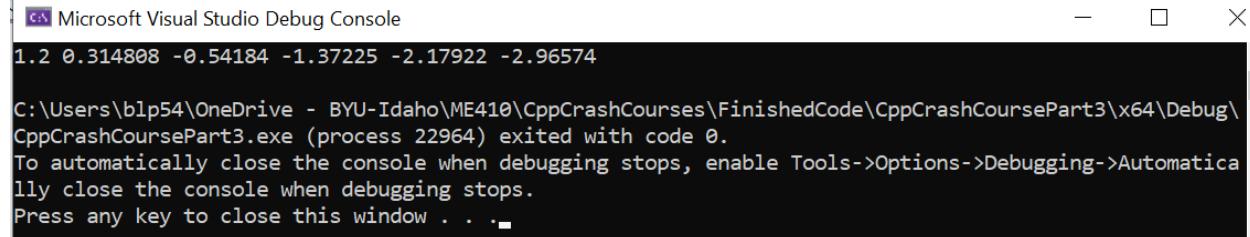
//Allocate memory to store the output sequence
int N = (int)u.size();
MyMatrixClass y(1, N, 10.0);

//run the FOR loop to simulate the state-space system
for (int ii = 0; ii < N; ii++) {
    //Store the output
    y(0, ii) = x(0, 0) + 0.8 * u[ii];

    //Solve x = Ad*x+Bd*u
    MyMatrixClass Bdu = Bd * u[ii];
    x = Ad * x;
    x = x + Bdu;
}

//Print the result
y.print();
}
```

The output printed to the console window is shown below:



Microsoft Visual Studio Debug Console

```
1.2 0.314808 -0.54184 -1.37225 -2.17922 -2.96574

C:\Users\blp54\OneDrive - BYU-Idaho\ME410\CppCrashCourses\FinishedCode\CppCrashCoursePart3\x64\Debug\
CppCrashCoursePart3.exe (process 22964) exited with code 0.
To automatically close the console when debugging stops, enable Tools->Options->Debugging->Automatically close the console when debugging stops.
Press any key to close this window . . .
```

Compared to C++ Crash Course 2, did MyMatrixClass make solving this problem easier, or at least reduce the number of lines of code required?

17.4.2 Do It Yourself

Modify your MyMatrixClass.h and MyMatrixClass.cpp libraries as was done above. Strive to understand each line of code. Use the updated libraries to solve the following state-space system:

$$\dot{x} = \begin{bmatrix} 0 & 1 \\ -6 & -7 \end{bmatrix} x + \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} u \quad (17.6)$$

$$y = \begin{bmatrix} 1 & 0 \\ 1 & -1 \end{bmatrix} x \quad (17.7)$$

with the initial condition

$$x(0) = \begin{bmatrix} -2 \\ -1 \end{bmatrix} \quad (17.8)$$

the time-step

$$\Delta t = 0.1 \quad (17.9)$$

and the input sequence

$$u = \begin{bmatrix} 4 & 3 & 2 & 1 & 0 & -1 \\ 1 & 2 & 3 & 4 & 5 & 6 \end{bmatrix} \quad (17.10)$$

Check your work by solving the same state-space system in MATLAB, python, or another programming language. This is the end of C++ Crash Course 3.

Chapter 18

Using C++ With MATLAB

Contents

18.1 Multiplying Matrices with mex Functions	503
18.2 Mex Functions With Multiple Files	509
18.3 Using Debugging Tools	514

After installing Visual C++ (see Section 15.1), you are ready to configure MATLAB to compile C++ code. Open MATLAB, and in the Command Window, type `mex -setup C++`. If the C++ compiler is installed correctly, the Command Window should say something like MEX configured to use ‘Microsoft Visual C++ 2022’ for C++ language compilation.

MATLAB can compile other coding languages into MATLAB Executable (mex) files. This can be a great benefit. For example, MATLAB can compile C++ code into mex functions. The compiled C++ code usually runs faster than MATLAB code, but C++ code does not have readily accessible and easy-to-use graphing, matrix math, symbolic math, etc. capabilities like MATLAB does. Also, code that is implemented in microcontrollers is often written in languages other than MATLAB. MATLAB can compile the code for these microcontrollers so it can be tested in the MATLAB and Simulink modeling environments before being flashed onto the microcontrollers.

18.1 Multiplying Matrices with mex Functions

This section will create MATLAB and C++ files. It is important that all the files for a mex program are saved in the same directory and that MATLAB’s working directory is set to that file location. This section will use the folder location named “MultiplyMatricesWithMexCPP”.

This section will create a mex function that uses C++ with MATLAB to multiply two matrices. Open Visual Studio, and select the option to Open a local folder. Navigate to the folder “MultiplyMatricesWithMexCPP” and select it.

In the Solution Explorer in Visual Studio, right click the folder MultiplyMatricesWithMexCPP and click Add New Item... Click Visual C++, click C++ File (.cpp), and then click Open. Change the name of the file that was created to `mexMatrixMultiply.cpp`. Add the following C++ code to the file `mexMatrixMultiply.cpp`. Instead of copying and pasting the code, it is recommended that you type it. Typing it helps you learn better, and may prevent compiler errors that are caused by incompatible font

formats between this file and Visual Studio. Read the comments in the code and try to understand how it works.

mexMatrixMultiply.cpp

```

//Include the Matlab Executable (mex) header files to use the following:
//mex, MexFunction, matlab::mex::Function, matlab::data,
//matlab::mex::ArgumentList, getNumberOfElements(), MATLABEngine(),
//TypedArray, ArrayFactory
#include "mex.hpp"
#include "mexAdapter.hpp"

#include <vector> //vector

using namespace matlab::data; //To use TypedArray, ArrayFactory, ArrayType, Array
                           //ArrayDimensions,
                           //without matlab::data::
using matlab::mex::ArgumentList; //To use ArgumentList without matlab::mex::
using namespace std; //To use vector without std::

//Fill the matrix M with the array data in the vector M_array
void MatlabInputArrayToMatrix(vector<vector<double>>& M, size_t nRows,
                             TypedArray<double> M_array)
{
    int ii = 0; //Row index
    int jj = 0; //Column index
    for (auto& elem : M_array) {
        M[ii][jj] = elem; //Populate the A matrix
        ii++;
        if (ii > nRows - 1) {
            //Go to the next column
            ii = 0;
            jj++;
        }
    }
}

//Convert the matrix OutMat into a Matlab output array Out_Array
matlab::data::TypedArray<double> MatrixToOutputMatlabArray(vector<vector<double>> OutMat)
{
    //Get the output matrix sizes for the Matlab output array
    size_t nRows = OutMat.size();
    size_t nCols = OutMat[0].size();
    vector<double> out(nRows * nCols);

    //Create the output Matlab array
    matlab::data::ArrayFactory factory;
    //Convert to a Matlab Array
    matlab::data::TypedArray<double> Out_Array =
        factory.createArray<double>({ nRows,nCols }, { 0.0 });
}

```

18.1 Multiplying Matrices with mex Functions

```
//Put the elements from the OutMat matrix into the Out_Array
int ii = 0;
int jj = 0;
for (auto& elem : Out_Array) {
    //Set the Out_Array elements to the OutMat elements
    elem = OutMat[ii][jj];
    ii++;
    if (ii > nRows - 1) {
        //Go to the next column
        ii = 0;
        jj++;
    }
}

//Return the Matlab output array
return Out_Array;
}

//Set up the matrix M to have nRows and nCols
void setMatrixSize(vector<vector<double>>& M, size_t nRows,
size_t nCols)
{
    //Resize the matrix M
    M.resize(nRows);
    for (int ii = 0; ii < nRows; ii++) {
        M[ii].resize(nCols);
    }
}

//Create a matrix multiplication function
vector<vector<double>> MultiplyMat(vector<vector<double>> A,
vector<vector<double>> B)
{
    //Declare the output matrix C and give it the correct dimensions
    vector<vector<double>> C(A.size(), vector<double>(B[0].size()));

    //check that matrix inner dimensions are compatible
    if (A[0].size() != B.size()) {
        //The matrices are not compatible
        throw 1;
    }

    //Perform the matrix multiplication
    for (int ii = 0; ii < A.size(); ii++) {
        for (int jj = 0; jj < B[0].size(); jj++) {
            C[ii][jj] = 0.0;
            for (int kk = 0; kk < A[0].size(); kk++) {
                C[ii][jj] += A[ii][kk] * B[kk][jj];
            }
        }
    }
}
```

```

//Return the matrix C=A*B
return C;
}

//The class MexFunction which inherits from matlab::mex::Function
// is required for mex functions that use C++
class MexFunction : public matlab::mex::Function {
public:

    //Declare the inputs from Matlab and the Outputs from C++ as
    // ArgumentList variables
    void operator()(ArgumentList outputs, ArgumentList inputs) {
        vector<vector<double>> C; //The output matrix C = A*B

        //Get the input matrix dimensions
        ArrayDimensions sizeA = inputs[0].getDimensions(); //Dims of A
        ArrayDimensions sizeB = inputs[1].getDimensions(); //Dims of B
        //Set the row and column sizes for the A and B matrices
        const size_t rowsA = sizeA[0];
        const size_t colsA = sizeA[1];
        size_t rowsB = sizeB[0];
        size_t colsB = sizeB[1];

        vector<vector<double>> A; //Declare the input matrix A
        //Resize the matrix A
        setMatrixSize(A, rowsA, colsA);
        //Get the data for the A matrix from the input
        TypedArray<double> A_Array = std::move(inputs[0]);
        //Put the data from A_Array into the A matrix
        MatlabInputArrayToMatrix(A, rowsA, A_Array);

        vector<vector<double>> B;//Declare the input matrix B
        //Resize the matrix B
        setMatrixSize(B, rowsB, colsB);
        //Get the data for the B matrix from the input
        TypedArray<double> B_Array = std::move(inputs[1]);
        //Put the data from B_Array into the B matrix
        MatlabInputArrayToMatrix(B, rowsB, B_Array);

        //Do the matrix multiplication
        C = MultiplyMat(A, B);

        //Convert the output matrix C into a Matlab output array C_Array
        matlab::data::TypedArray<double> C_Array = MatrixToOutputMatlabArray(C);

        //Return the output array C_Array to Matlab
        outputs[0] = C_Array;
    }
};

```

Open a new tab in a Matlab editor, and save the file as runmeMultiplyMatricesWithMexCPP.m. Make sure that it is saved in the same folder location as mexMatrixMultiply.cpp. Type the following code:

18.1 Multiplying Matrices with mex Functions

runmeMultiplyMatricesWithMexCPP.m

```
close all
clear all
clc

%% compile the C++ mex function
mex -g mexMatrixMultiply.cpp

%% Run the function

A = randi(9,[2,4]);
B = randi(9,[4,3]);
C = mexMatrixMultiply(A,B)
A*B
```

Running the code in runmeMultiplyMatricesWithMexCPP.m compiles the mexMatrixMultiply.cpp file into a Matlab mex function. This was done by the line of code: mex -g mexMatrixMultiply.cpp. The -g option compiles it with debug symbols so that a debugger can step through the code if Visual Studio is attached to Matlab. Attaching Visual Studio to Matlab and debugging is a topic that will be discussed later in this chapter. Once it is compiled, the mex function can be called directly from Matlab, as long as the Matlab path includes the directory containing the mex function. The Matlab program runs the compiled mex program by calling the function using the line C = mexMatrixMultiply(A,B). Notice that the name of the function is the same name as the .cpp file: mexMatrixMultiply. If there are multiple files, which is the topic of the next section, the name of the Matlab function will match the name of the first .cpp file after the mex command: mex -g.

We will now discuss the code in the mexMatrixMultiply.cpp file. Many of the commands, data types, namespaces, and functions are original because they are defined in MATLAB's header files "mex.hpp" and "mexAdapter.hpp". We include these MATLAB executable libraries using the commands:

```
#include "mex.hpp"
#include "mexAdapter.hpp"
```

To learn more about these header files and their contents, visit the website <https://github.com/alecjacobson/matlab/tree/master/extern/include>. Only MATLAB can compile code with these .hpp files unless the compiler path in Visual Studio (or a different C++ development environment) is pointed to them. The namespace matlab::data is defined in the header files, and using namespace matlab::data; allows the program to use the mex defined functions TypedArray, ArrayFactory, ArrayType, Array, ArrayDimensions without first typing matlab::data:: in front of them. The command using matlab::mex::ArgumentList; is similar in that it allows the code to use the ArgumentList data type without first writing matlab::mex:: in front of it. Writing using namespace std; allows the code to use the vector data type without first writing std::::

The next part of the code is a function that we create to fill a matrix (actually a vector of vectors of double precision numbers) named M with the data in the Matlab array M_array:

```

void MatlabInputArrayToMatrix(vector<vector<double>>& M, size_t nRows,
TypedArray<double> M_array)
{
    int ii = 0; //Row index
    int jj = 0; //Column index
    for (auto& elem : M_array) {
        M[ii][jj] = elem; //Populate the A matrix
        ii++;
        if (ii > nRows - 1) {
            //Go to the next column
            ii = 0;
            jj++;
        }
    }
}

```

The matrix M is created elsewhere and is passed by reference to the function `MatlabInputArrayToMatrix` using the `&` operator. The Matlab array `M_array` is a one-dimensional array that contains all the data for the columns and rows of the M matrix. The command `for (auto& elem : M_array)` is a for loop that iterates through each element in `M_array`. The command `M[ii][jj] = elem;` copies the elements from the `M_array` vector into the rows and columns of the matrix M . The `auto` keyword automatically detects the data type of the elements in `M_array`. It is useful when the data type is complicated or unknown. The `ii++;` line causes the matrix to move to the next row in the column. When the column is full, the code in the `if (ii > nRows - 1)` statement iterates to the next column of the M matrix.

The next function in the code is also one that we create, but it does exactly the opposite of the `MatlabInputArrayToMatrix` function. It converts a matrix `OutMat` of type `vector<vector<double>>` to a one-dimensional Matlab array `Out_Array` of type `TypedArray`. The data type `TypedArray` is defined by the `mex.hpp` header file.

The function `setMatrixSize` is one that we create. It allocated computer memory to store the data for the matrix M , which is passed to it by reference. As was done in Section 16.3, the function `MultiplyMat` uses three nested for loops to perform the matrix multiplication $C = A * B$.

The interface between Matlab and C++ is created by the `MexFunction` class, which inherits from the `matlab::mex::Function` class. The parentheses operator is overloaded in the public space of the `MexFunction` class by the command `void operator()(ArgumentList outputs, ArgumentList inputs)`. Overloading this operator allows Matlab to send variables to the C++ file as inputs. It also allows the C++ code to return variables to Matlab as outputs. Both inputs and outputs are of data type `ArgumentList`. Notice that in the `runmeMultiplyMatricesWithMex.CPP.m` file, that the function `C = mexMatrixMultiply(A,B)` has two inputs: A and B . It has one output C . The `inputs` variable is a pointer to A and B . Specifically, the command `TypedArray<double> A_Array = std::move(inputs[0]);` creates a one-dimensional Matlab array that contains the data in the A matrix. The command `TypedArray<double> B_Array = std::move(inputs[1]);` does the same for the B matrix. The C++ command `MatlabInputArrayToMatrix(A, rowsA, A_Array);` copies the data from `A_Array` into the rows and columns of `vector<vector<double>> A;`, and the command `MatlabInputArrayToMatrix(B, rowsB, B_Array);` does the same for the B matrix. If there was a third input, it could be accessed by referencing `inputs[2]`. The command `C = MultiplyMat(A, B);` multiplies the

18.2 Mex Functions With Multiple Files

A and B matrices. The resulting matrix is C. The command `matlab::data::TypedArray<double> C_Array = MatrixToOutputMatlabArray(C);` copies the data from the C matrix into a Matlab array C_Array, and the code `outputs[0] = C_Array;` returns C_Array to Matlab. Because Out_Array was created to have nRows and nCols, C_Array is returned to Matlab having the correct dimensions. If there was a second output, it would be returned to Matlab using the code `outputs[1]`.

18.2 Mex Functions With Multiple Files

This section accomplishes the same thing as the previous section. The difference is that it uses multiple C++ files instead of only one. The purpose is to demonstrate how to create and compile mex (Matlab Executable) functions from multiple files.

In this example, the code from the last section's `mexMatrixMultiply.cpp` file is divided into the following seven files:

1. `MultiplyMat.h`
2. `MultiplyMat.cpp`
3. `MatlabInputArrayToMatrix.h`
4. `MatlabInputArrayToMatrix.cpp`
5. `MatrixToOutputMatlabArray.h`
6. `MatrixToOutputMatlabArray.cpp`
7. `mexMatlabInterface.cpp`

There is also one Matlab file: `runmeMexMultipleFiles.m`. All of the files are contained in the same directory. The Matlab file `runmeMexMultipleFiles.m` is provided first:

```
%% compile the C++ mex function
mex -g MultiplyMat.cpp ...
mexMatlabInterface.cpp ...
MatlabInputArrayToMatrix.cpp ...
MatrixToOutputMatlabArray.cpp

%% Run the function
A = randi(9,[2,4]);
B = randi(9,[4,3]);
C = MultiplyMat(A,B)
A*B
```

The command to compile all of the C++ files is `mex -g MultiplyMat.cpp mexMatlabInterface.cpp MatlabInputArrayToMatrix.cpp MatrixToOutputMatlabArray.cpp`. Notice that when Matlab calls the compiled C++ function, the name of the function is `MultiplyMat`. That is because the first file after `mex -g` was `MultiplyMat.cpp`. Here are the seven C++ files:

MultiplyMat.h

```
#pragma once

#include <vector> //vector
using namespace std; //To use vector without std::

//Declare the MultiplyMat function
extern vector<vector<double>> MultiplyMat(vector<vector<double>> A,
                                              vector<vector<double>> B);
```

MultiplyMat.cpp

```
#include "MultiplyMat.h" //Include the MultiplyMat header file

//Create a matrix multiplication function
extern vector<vector<double>> MultiplyMat(vector<vector<double>> A,
                                              vector<vector<double>> B)
{
    //Declare the output matrix C and give it the correct dimensions
    vector<vector<double>> C(A.size(), vector<double>(B[0].size()));
    //check that matrix inner dimensions are compatible
    if (A[0].size() != B.size()) {
        //The matrices are not compatible
        throw 1;
    }
    //Perform the matrix multiplication
    for (int ii = 0; ii < A.size(); ii++) {
        for (int jj = 0; jj < B[0].size(); jj++) {
            C[ii][jj] = 0.0;
            for (int kk = 0; kk < A[0].size(); kk++) {
                C[ii][jj] += A[ii][kk] * B[kk][jj];
            }
        }
    }
    //Return the matrix C=A*B
    return C;
}
```

18.2 Mex Functions With Multiple Files

MatlabInputArrayToMatrix.h

```
#pragma once

#include "mex.hpp" //To use TypedArray
#include <vector> //vector

using namespace matlab::data; //To use TypedArray without matlab::data::
using namespace std; //To use vector without std::

//Declare the function
extern void MatlabInputArrayToMatrix(vector<vector<double>>& M, size_t nRows,
    TypedArray<double> M_array);
```

MatlabInputArrayToMatrix.cpp

```
#include "MatlabInputArrayToMatrix.h"

//Fill the matrix M with the array data in the vector M_array
extern void MatlabInputArrayToMatrix(vector<vector<double>>& M, size_t nRows,
    TypedArray<double> M_array)
{
    int ii = 0; //Row index
    int jj = 0; //Column index
    for (auto& elem : M_array) {
        M[ii][jj] = elem; //Populate the A matrix
        ii++;
        if (ii > nRows - 1) {
            //Go to the next column
            ii = 0;
            jj++;
        }
    }
}
```

MatrixToOutputMatlabArray.h

```
#pragma once

#include "mex.hpp" //To use TypedArray
#include <vector> //vector
```

```

using namespace matlab::data; //To use TypedArray without matlab::data::
using namespace std; //To use vector without std::

//Declare the function
extern matlab::data::TypedArray<double> MatrixToOutputMatlabArray(
    vector<vector<double>> OutMat);

```

MatrixToOutputMatlabArray.cpp

```

#include "MatrixToOutputMatlabArray.h"

//Convert the matrix OutMat into a Matlab output array Out_Array
extern matlab::data::TypedArray<double> MatrixToOutputMatlabArray(
    vector<vector<double>> OutMat)
{
    //Get the output matrix sizes for the Matlab output array
    size_t nRows = OutMat.size();
    size_t nCols = OutMat[0].size();
    vector<double> out(nRows * nCols);
    //Create the output Matlab array
    matlab::data::ArrayFactory factory;
    //Convert to a Matlab Array
    matlab::data::TypedArray<double> Out_Array =
        factory.createArray<double>({ nRows,nCols }, { 0.0 });
    //Put the elements from the OutMat matrix into the Out_Array
    int ii = 0;
    int jj = 0;
    for (auto& elem : Out_Array) {
        //Set the Out_Array elements to the OutMat elements
        elem = OutMat[ii][jj];
        ii++;
        if (ii > nRows - 1) {
            //Go to the next column
            ii = 0;
            jj++;
        }
    }
    //Return the Matlab output array
    return Out_Array;
}

```

18.2 Mex Functions With Multiple Files

mexMatlabInterface.cpp

```
//Include the Matlab Executable (mex) header files to use the following:  
//mex, MexFunction, matlab::mex::Function, matlab::data,  
//matlab::mex::ArgumentList, getNumberOfElements(), MATLABEngine(),  
//TypedArray, ArrayFactory  
#include "mex.hpp"  
#include "mexAdapter.hpp"  
#include "MatrixToOutputMatlabArray.h" //To use MatrixToOutputMatlabArray()  
#include "MatlabInputArrayToMatrix.h" //To use MatlabInputArrayToMatrix()  
#include "MultiplyMat.h" //Include the MultiplyMat header file  
#include <vector> //vector  
using namespace matlab::data; //To use TypedArray, ArrayFactory, ArrayType, Array  
//ArrayDimensions,  
//without matlab::data::  
using matlab::mex::ArgumentList; //To use ArgumentList without matlab::mex::  
using namespace std; //To use vector without std::  
  
//Set up the matrix M to have nRows and nCols  
void setMatrixSize(vector<vector<double>>& M, size_t nRows,  
    size_t nCols)  
{  
    //Resize the matrix M  
    M.resize(nRows);  
    for (int ii = 0; ii < nRows; ii++) {  
        M[ii].resize(nCols);  
    }  
}  
  
//The class MexFunction which inherits from matlab::mex::Function  
// is required for mex functions that use C++  
class MexFunction : public matlab::mex::Function {  
public:  
    //Declare the inputs from Matlab and the Outputs from C++ as  
    // ArgumentList variables  
    void operator()(ArgumentList outputs, ArgumentList inputs) {  
        vector<vector<double>> C; //The output matrix C = A*B  
        //Get the input matrix dimensions  
        ArrayDimensions sizeA = inputs[0].getDimensions(); //Dims of A  
        ArrayDimensions sizeB = inputs[1].getDimensions(); //Dims of B  
        //Set the row and column sizes for the A and B matrices  
        const size_t rowsA = sizeA[0];  
        const size_t colsA = sizeA[1];  
        size_t rowsB = sizeB[0];  
        size_t colsB = sizeB[1];  
        vector<vector<double>> A; //Declare the input matrix A  
        //Resize the matrix A  
        setMatrixSize(A, rowsA, colsA);  
        //Get the data for the A matrix from the input  
        TypedArray<double> A_Array = std::move(inputs[0]);  
        //Put the data from A_Array into the A matrix  
        MatlabInputArrayToMatrix(A, rowsA, A_Array);
```

```

vector<vector<double>> B;//Declare the input matrix B
//Resize the matrix B
setMatrixSize(B, rowsB, colsB);
//Get the data for the B matrix from the input
TypedArray<double> B_Array = std::move(inputs[1]);
//Put the data from B_Array into the B matrix
MatlabInputArrayToMatrix(B, rowsB, B_Array);
//Do the matrix multiplication
C = MultiplyMat(A, B);
//Convert the output matrix C into a Matlab output array C_Array
matlab::data::TypedArray<double> C_Array = MatrixToOutputMatlabArray(C);
//Return the output array C_Array to Matlab
outputs[0] = C_Array;
}
};

```

Take some time to compare the code in this section to the previous section. Both sections use the same code to accomplish the same purpose of matrix multiplication. However, this section divided the code into multiple files whereas the previous section used only one C++ file. The code was explained in the previous section, and it is not repeated here.

18.3 Using Debugging Tools

Visual Studio can attach to MATLAB processes during runtime. This can enable you to use Visual Studio's debugging capabilities to find errors as you walk line-by-line through your compiled mex function C++ code. At each step, you can watch how variables are assigned new values. This section introduces how to use these debugging capabilities.

To use the runtime debugger, your code must compile successfully. If your code fails to compile, fix the errors that are printed to MATLAB's Command Window after running the `mex -g ...` command. The `-g` option enables use of the debugging tools. If the code compiles, but crashes MATLAB or Visual Studio when you try to run it, it is usually because the program is attempting to access memory that has not been allocated. You can find the source of this error by using the debugging tools. If code runs without crashing the program but produces the wrong values, the debugging tools can help fix these problems as well.

To use the debugging tools, first clear all the mex functions by running the command `clear functions` or `clear all` in MATLAB's Command Window. Then compile the code using the `mex -g ...` command. Open the C++ file that you want to debug in Visual Studio. Click on the `Debug` tab in Visual Studio and select `Attach to Process` (see Figure 18.1).

In the box that pops up, click the "Select..." button next to the "Attach to:" box. Then set the code type to "Native", see Figure 18.2 and click "OK".

In the "Available processes" option to select the **MATLAB.exe** process and click **Attach**, see Figure 18.3.

Now Visual Studio should be attached to the MATLAB process. You can now use the debugging tools in Visual Studio. *Make sure not to run the functions `clear all` or `clear functions` in MATLAB after attaching to Visual Studio. If you do, it's not a big problem, but you will need to reattach Visual Studio to MATLAB.*

A break-point in the code can freeze the process at a specified line in the code during runtime. To

18.3 Using Debugging Tools

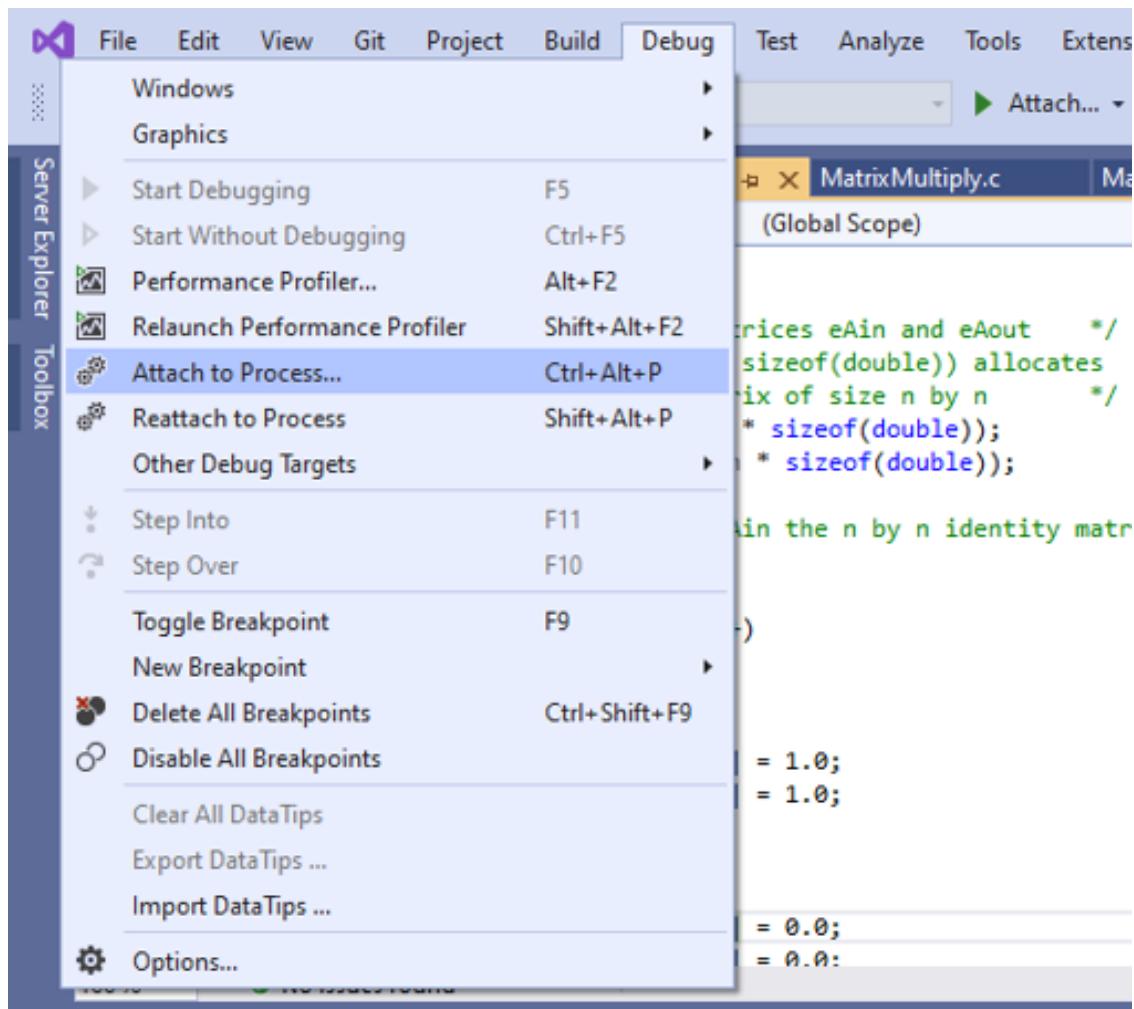


Figure 18.1 Select “Attach to Process...”

set a break-point in Visual Studio, click on the bar on the left of the line numbers (see Figure 18.4). The break-point may say that it will not be hit. That is likely because the MATLAB mex function is not currently being run.

In MATLAB, run the mex function, but be sure not to run the functions **clear all** or **clear functions**. The process should stop at the break-point that was set in Visual Studio. If the program crashes without stopping at the break-point, try setting the break-point at an earlier line in the C++ code, before any array operations are performed.

The debugging tools shown in Figure 18.5 include **Continue**, **Step Into**, **Step Over**, and **Step Out**. Use the debugging tools to step through the code. The autos box shows the values of the variables that are on the line of code that is being executed. As you step through the code, watch how these values change. Doing so can help you detect and correct mistakes.

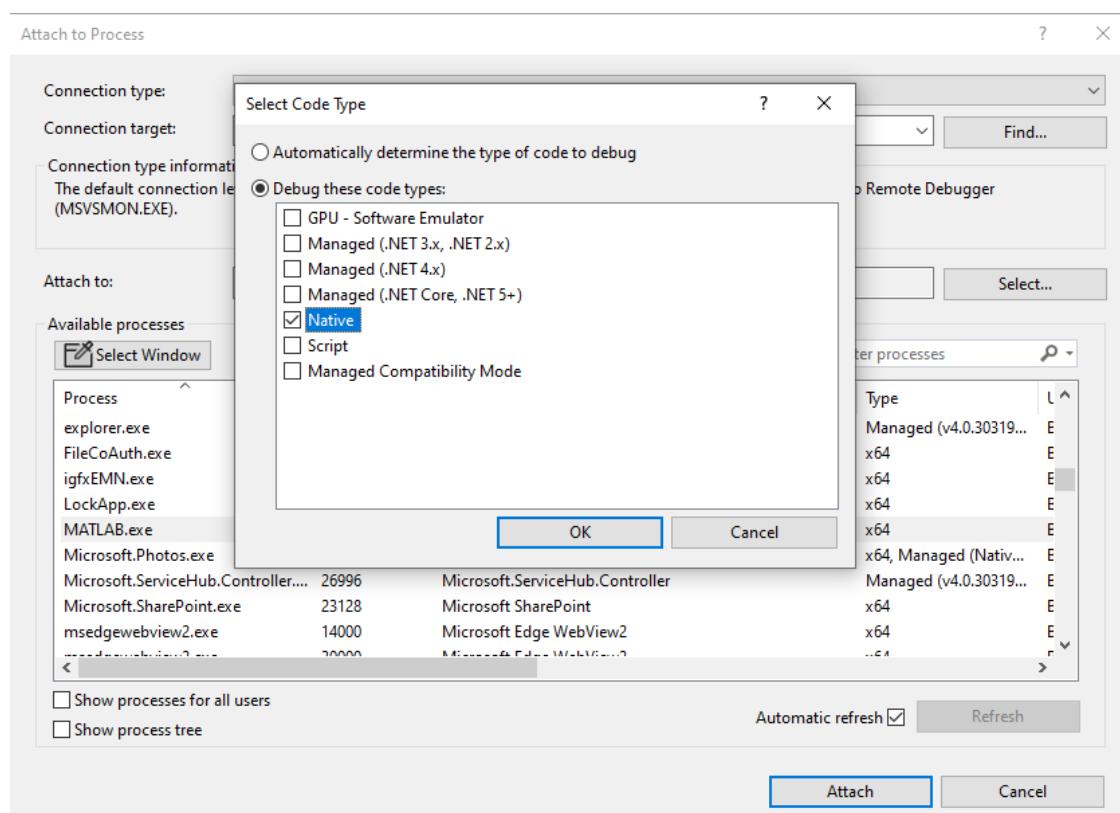


Figure 18.2 Select “Native” code type

18.3 Using Debugging Tools

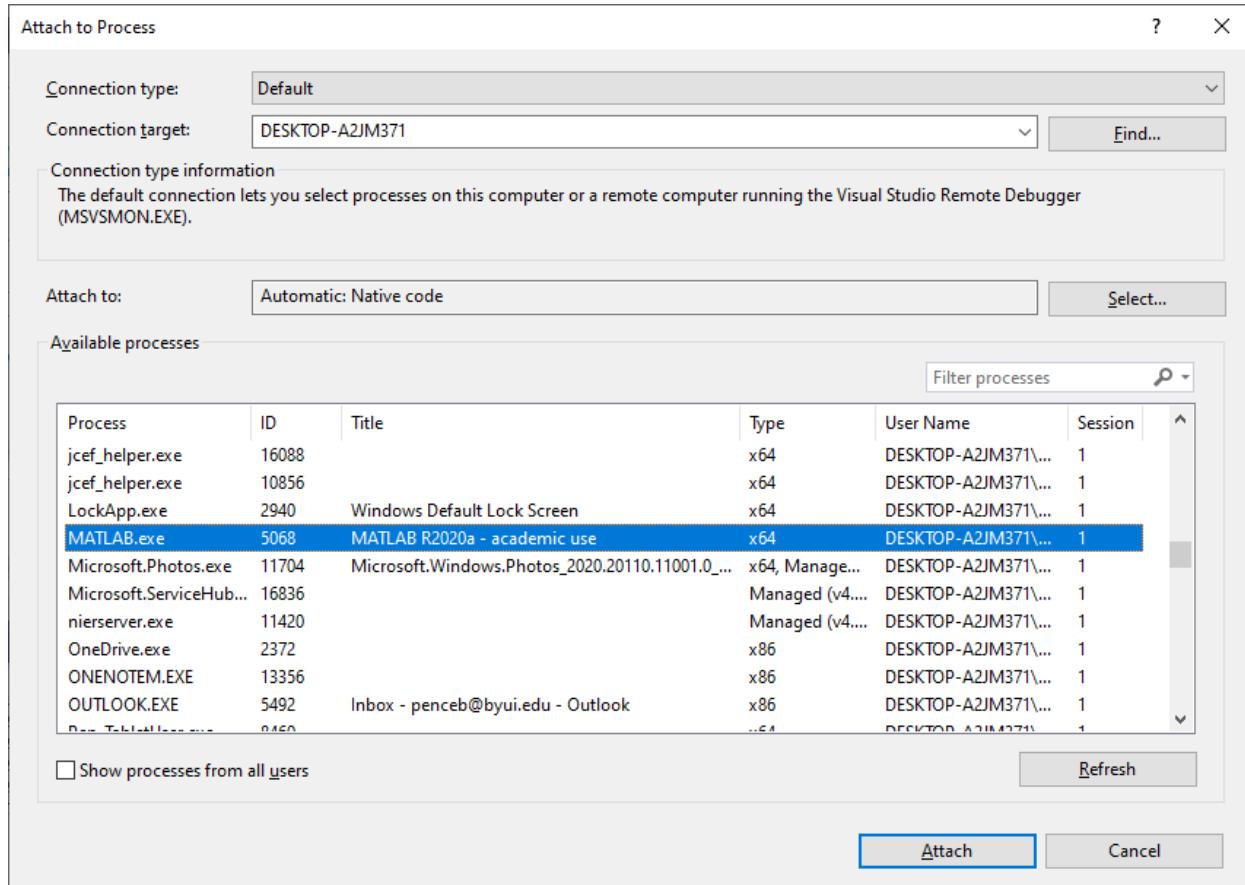


Figure 18.3 Select “MATLAB.exe” and click “Attach”

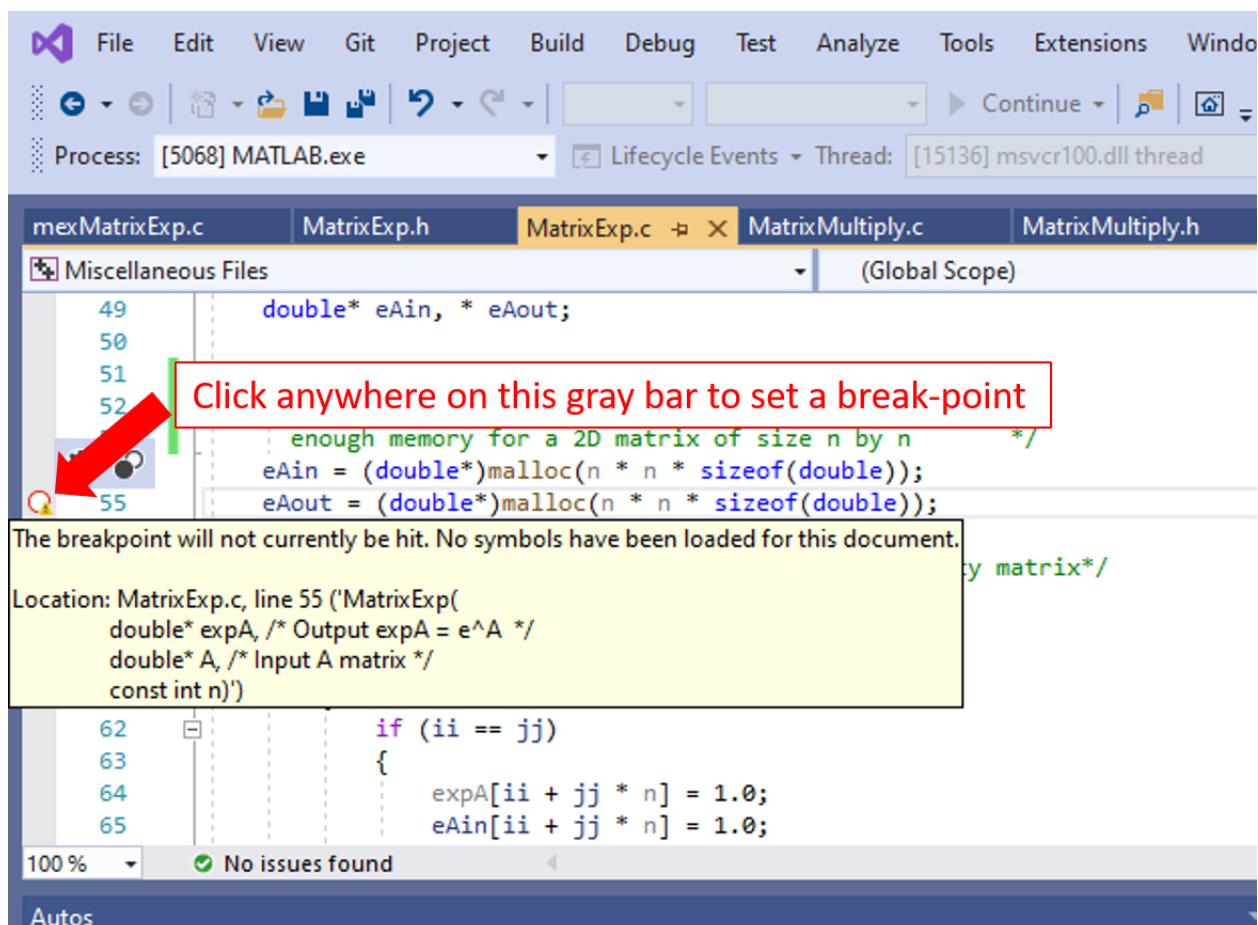


Figure 18.4 Click next to the line number to set a break-point

18.3 Using Debugging Tools

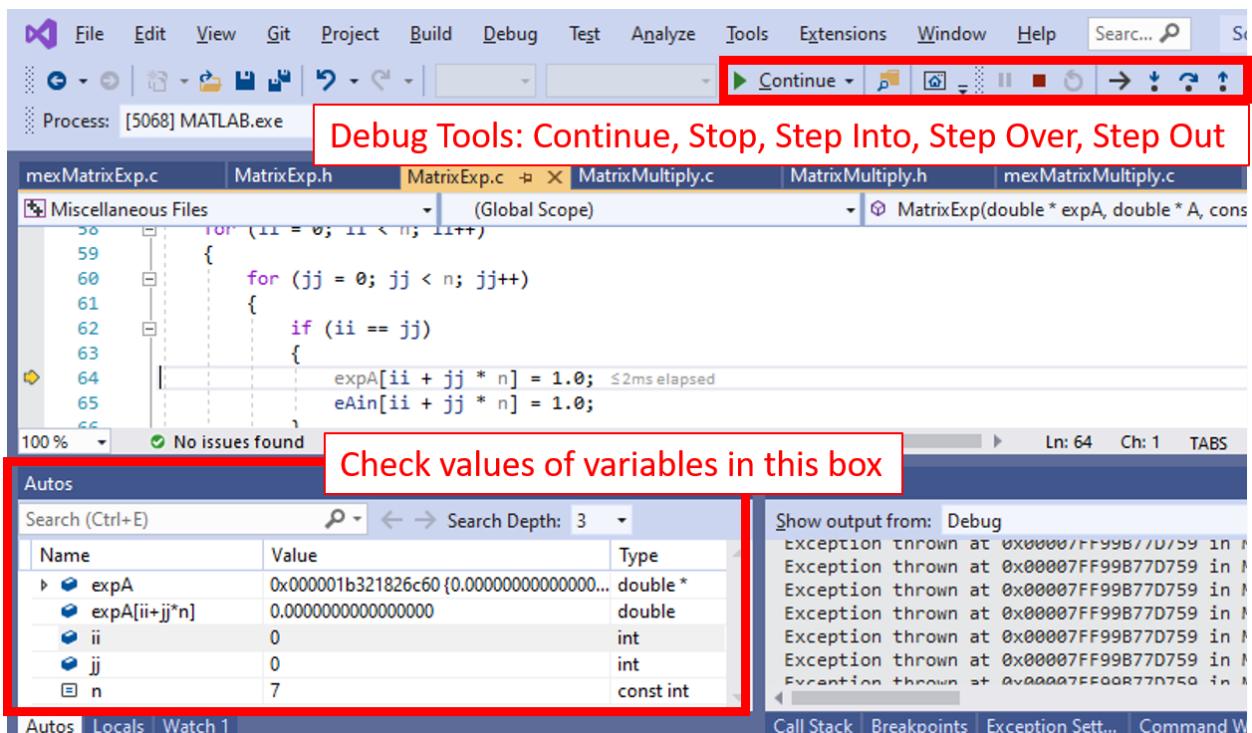


Figure 18.5 The debugging tools and autos box help detect errors

Chapter 19

Creating Custom Arduino Libraries

Contents

19.1 Programming the Raspberry Pi Pico with the Arduino IDE	521
19.2 Creating Arduino Libraries	523

This chapter explains how to create Arduino libraries. The Raspberry Pi Pico microcontroller and many other microcontrollers can be programmed using the Arduino IDE (Integrated Development Environment), so learning how to write Arduino libraries can be invaluable. The Arduino website includes a guide to writing libraries which can be found here: <https://docs.arduino.cc/learn/contributions/arduino-creating-library-guide> (accessed April 2023). The sections below will also walk through basic steps to creating Arduino libraries.

19.1 Programming the Raspberry Pi Pico with the Arduino IDE

The Raspberry Pi Pico requires some minor setup before it can be programmed by the Arduino IDE. The steps are explained below:

- Open the Arduino IDE and go to File->Preferences.
- In the pop-up box, copy and paste the following URL into the “Additional Boards Manager URLs” field: https://github.com/earlephilhower/arduino-pico/releases/download/global/package_rp2040_index.json, see Figure 19.1. Make sure there are no spaces in the pasted URL.
- Click OK to close the dialog.
- Go to Tools->Boards->Board Manager in the IDE
- Type “pico” in the search box
- Install the package: Raspberry Pi Pico/RP2040 by Earle F. Philhower, III

After it is done installing, to use the Pico, go to tools->Board->Raspberry Pi RP2040 Boards->Raspberry Pi Pico. Make sure to select the right port, and if necessary, press the BOOTSEL button on the Pico before

and while connecting the USB cable. The Raspberry Pi Pico should be ready to program with the Arduino IDE.

You cannot brick the Raspberry Pi Pico with software. If ever necessary, there is a file available from Raspberry Pi called “flash_nuke.uf2” available at https://datasheets.raspberrypi.com/soft/flash_nuke.uf2. This file can be used to reset the flash memory. When you click on the link, it will download the file. You can copy and paste it into the directory that pops up when you plug in the Raspberry Pi Pico while holding down the reset button.

If the Raspberry Pi Pico causes a folder to pop up every time it is plugged in, and / or Arduino does not recognize the port it is plugged into, you may still be able to add your compiled Arduino code to the Raspberry Pi Pico. To do so, compile the Arduino sketch, then select Sketch -> Export Compiled Binary (or Alt+Ctrl+S). This will download the compiled Arduino code as a .uf2 file. Copy and paste the .uf2 file into the folder that pops up when the Raspberry Pi Pico is plugged in. This should do the same thing as flashing the Pico. It may also reset it to a state in which the Arduino IDE recognizes it as a board.

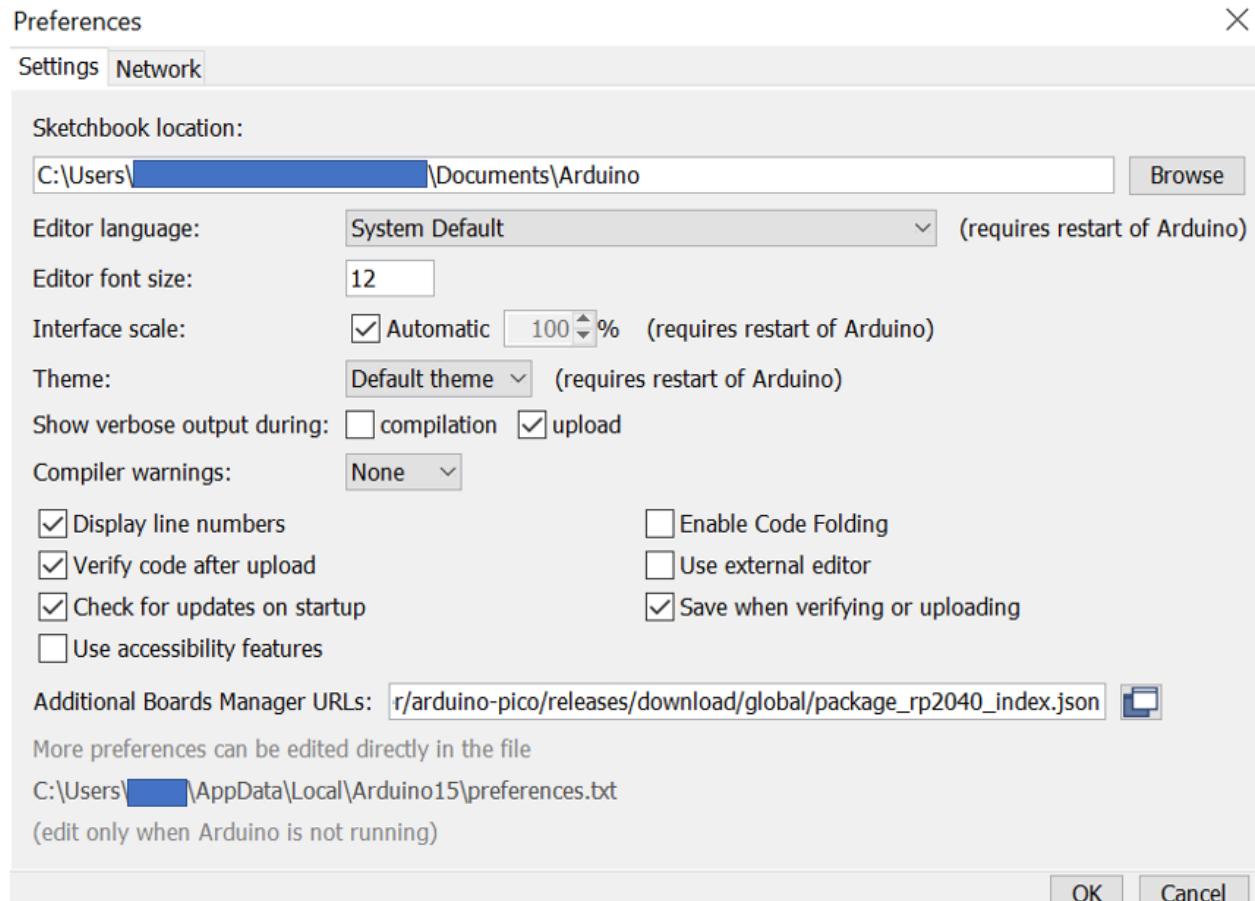


Figure 19.1 The Additional Boards Managers URLs box includes https://github.com/earlephilhower/arduino-pico/releases/download/global/package_rp2040_index.json, and the Sketchbook location: shows the path to find the Arduino libraries folder.

19.2 Creating Arduino Libraries

19.2 Creating Arduino Libraries

An Arduino library requires at least two files: a C++ header (.h) file and a C++ source (.cpp) file. Both must have the same name, and they must be inside a folder with the same name. Additional source and header files can be included as well.

For example, we will create an Arduino library called `Multiply3x3Matrix` that will multiply two 3x3 matrices and return the resulting matrix. The library will include four files:

1. `Multiply3x3Matrix.h`
2. `Multiply3x3Matrix.cpp`
3. `MatrixMultiply.h`
4. `MatrixMultiply.cpp`

All four files will be located in the same folder named `Multiply3x3Matrix`. The `Multiply3x3Matrix` folder must be located in the location that Arduino looks for libraries. The default location is usually `C:/Users/.../Documents/Arduino/libraries`. To find the Arduino libraries location on your computer, open the Arduino IDE, click `File»Preferences`. Under the `Settings` tab, the `Sketchbook` location: entry box shows the path to the libraries folder, see Figure 19.1. Note that it does not include the final `libraries` directory.

The four Arduino library files are provided below:

Multiply3x3Matrix.h

```
#pragma once

//Multiply two 3x3 matrices: C=A*B
extern void Multiply3x3Matrix(double C[3][3],
    const double A[3][3], const double B[3][3]);
```

Multiply3x3Matrix.cpp

```
#include "Multiply3x3Matrix.h"
#include "MatrixMultiply.h"

//Multiply two 3x3 matrices: C=A*B
extern void Multiply3x3Matrix(double C[3][3],
    const double A[3][3], const double B[3][3])
{
    mat3_mult(C, A, B);
}
```

MatrixMultiply.h

```
#pragma once

//Multiply two 3x3 matrices
void mat3_mult(double C[3][3], const double A[3][3], const double B[3][3]);

//Multiply two 4x4 matrices
void mat4_mult(double C[4][4], const double A[4][4], const double B[4][4]);
```

MatrixMultiply.cpp

```
#include "MatrixMultiply.h"

//Multiply two 3x3 matrices
const int Msize = 3;
void mat3_mult(double C[3][3], const double A[3][3], const double B[3][3]) {
    for (int ii = 0; ii < Msize; ii++) {
        for (int jj = 0; jj < Msize; jj++) {
            C[ii][jj] = 0.0;
            for (int kk = 0; kk < Msize; kk++) {
                C[ii][jj] += A[ii][kk] * B[kk][jj];
            }
        }
    }
}

//Multiply two 4x4 matrices
const int MatSize = 4;
void mat4_mult(double C[4][4], const double A[4][4], const double B[4][4]) {
    for (int ii = 0; ii < MatSize; ii++) {
        for (int jj = 0; jj < MatSize; jj++) {
            C[ii][jj] = 0.0;
            for (int kk = 0; kk < MatSize; kk++) {
                C[ii][jj] += A[ii][kk] * B[kk][jj];
            }
        }
    }
}
```

Unfortunately, as of April 2023, Arduino does not support the use of the C++ Standard Template Libraries (STL). Therefore, the `std::vector` and `std::string` classes, etc. are not available for use in writing Arduino libraries.

If the Arduino IDE was open during the creation of the Arduino library files above, it will not recognize them until the Arduino IDE is closed and reopened. Once it is reopened, the new Arduino library can be

19.2 Creating Arduino Libraries

found under Sketch » Include Library, see Figure 19.2.

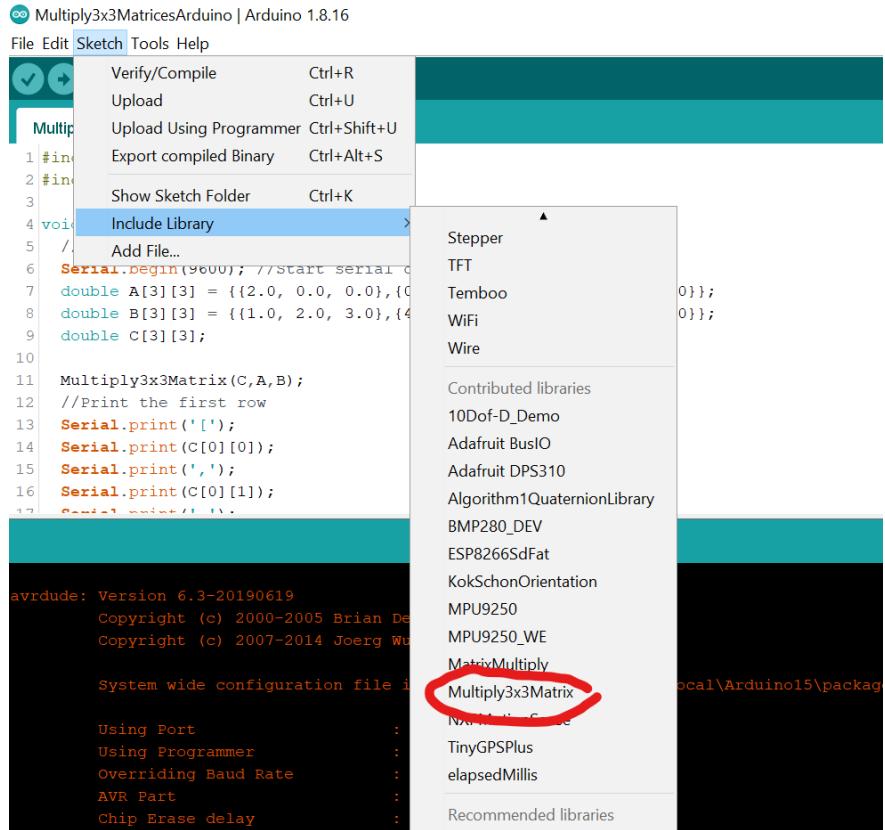


Figure 19.2 The Multiply3x3Matrix library is now in the Include Library directory.

The following Arduino code was used to test the Multiply3x3Matrix library:

Multiply3x3MatricesArduino.ino

```
#include <MatrixMultiply.h>
#include <Multiply3x3Matrix.h>

void setup() {
    // put your setup code here, to run once:
    Serial.begin(9600); //Start serial communication
    double A[3][3] = {{2.0, 0.0, 0.0},{0.0, 2.0, 0.0},{0.0, 0.0, 2.0}};
    double B[3][3] = {{1.0, 2.0, 3.0},{4.0, 5.0, 6.0},{7.0, 8.0, 9.0}};
    double C[3][3];

    Multiply3x3Matrix(C,A,B);
    //Print the first row
    Serial.print('[');
    Serial.print(C[0][0]);
```

```
Serial.print(',');
Serial.print(C[0][1]);
Serial.print(',');
Serial.print(C[0][2]);
Serial.println(';');
//Print the second row
Serial.print(C[1][0]);
Serial.print(',');
Serial.print(C[1][1]);
Serial.print(',');
Serial.print(C[1][2]);
Serial.println(';');
//Print the third row
Serial.print(C[2][0]);
Serial.print(',');
Serial.print(C[2][1]);
Serial.print(',');
Serial.print(C[2][2]);
Serial.println(']');
}

void loop() {
    // put your main code here, to run repeatedly:
}
```

This code was uploaded to an Arduino Uno. The result was displayed on the Serial Monitor, see Figure 19.3.

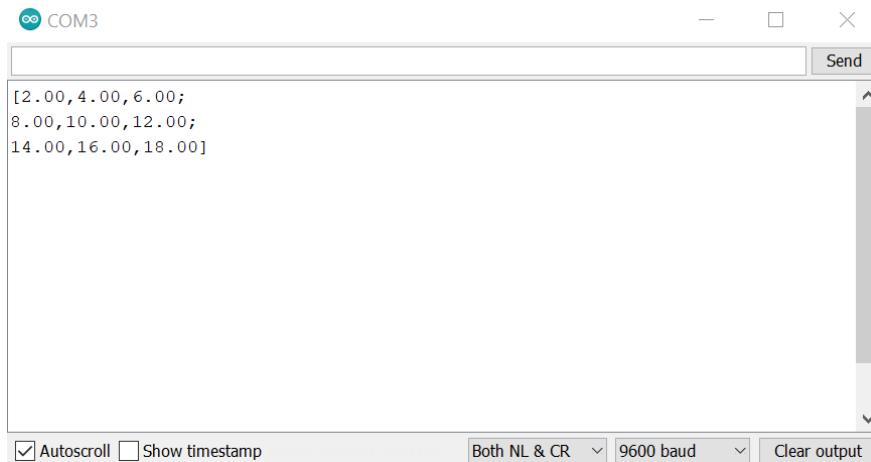


Figure 19.3 Result of the Arduino code in Multiply3x3MatricesArduino.ino shown on the Arduino Serial Monitor.

Chapter 20

NED: The MATLAB Flight Simulator

Contents

20.1 MATLAB Code for NED: The MATLAB Flight Simulator	528
---	-----

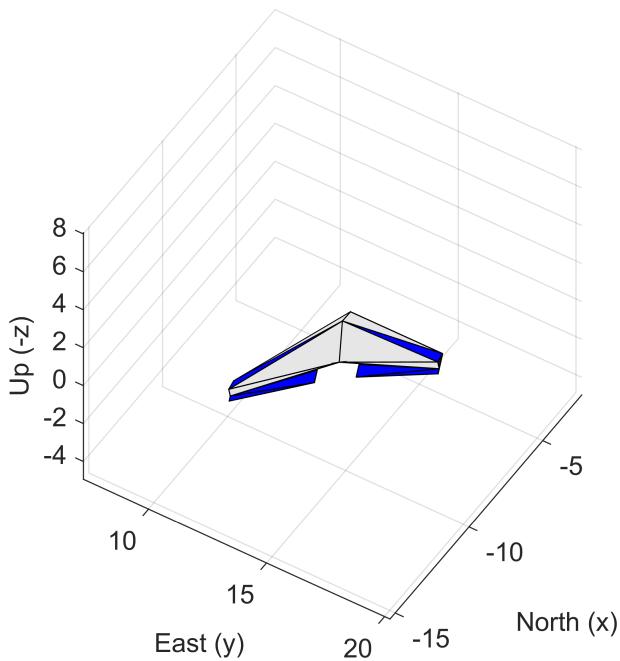


Figure 20.1 A screen capture from NED: The MATLAB Flight Simulator

This chapter provides code for NED: The MATLAB Flight Simulator. It simulates the flight of a fixed-wing airplane. The purpose of NED: The MATLAB Flight Simulator is to provide a development environment for writing control strategies. Students and engineers can write and test control strategies, orientation estimation algorithms, signal filters, and system ID programs. These programs can be tested

on the simulator before being implemented in practice. One benefit is that in the simulator, all the true signals are known and can be compared with estimated values. The simulator enables faster iterations to improve the programs. It facilitates development at a lower financial cost and with less risk of damage than directly applying them on real airplanes.

To run the simulator, copy and paste the MATLAB code into a MATLAB editor, and press run. The key-press events in Table 20.1 are used to control the airplane in the simulator.

Table 20.1 Keyboard keys for controlling the airplane

Key	Function	Description
w	Throttle $\delta_t = 1$	Full throttle
s	Throttle $\delta_t = 0$	No throttle
Up arrow ↑	Elevator $\delta_e = 1$	Pitch down
Down arrow ↓	Elevator $\delta_e = -1$	Pitch up
Right arrow →	Aileron $\delta_a = 1$	Roll right
Left arrow ←	Aileron $\delta_a = -1$	Roll left
d	Rudder $\delta_r = 1$	Yaw right
a	Rudder $\delta_r = -1$	Yaw left

20.1 MATLAB Code for NED: The MATLAB Flight Simulator

```
%% This is the runme file for NED, the MATLAB flight simulator
% Originally created 14 July, 2022, for Autonomous Control of
% Dynamic Systems

%Reset everything
close all
clear all
clc

%% Set the initial conditions for the airplane state and wind gust state
% The airplane state is
% x=[u;v;w;p;q;r;xI;yI;zI; e0;e1; e2;e3]
x = [2;0;0;0;0;0; 0; 0;-20;0.9763; 0;0.2164; 0];

% The wind gust state is xGust = [uwg;x1;x2;x3;x4;Va]
% Note: Va is the relative airspeed of the plane from the previous
% iteration
xGust = [0;0;0;0;0;1];

%% set the airplane constants in a structure named c
c.mass = 1; %[kg] airplane mass
c.Jxx = 0.12; %[kg m2] Moment of inertia about x-axis (nose)
c.Jyy = 0.2; %[kg m2] Moment of inertia about y-axis (right-wing)
c.Jzz = 0.18; %[kg m2] Moment of inertia about z-axis (down)
c.Jxz = 0.015; %[kg m2] xz Product of inertia
```

20.1 MATLAB Code for NED: The MATLAB Flight Simulator

```
c.rho = 1.27; %[kg/m3] air density
c.S = 1.5; %[m] wing span
c.c = 0.3; %[m] wing chord
c.A = c.S*c.c; %[m2] wing area
c.CL0 = -0.9;
c.CL1 = 2.5;
c.CL2 = 1;
c.CL3 = 11;
c.CL4 = 0.004;
c.CL_delta_e = 0.1; %[-] Elevator lift coefficient
c.CD0 = 0.06;
c.CD2 = 0.44;
c.CD_delta_e = 0.001; %[-] Elevator drag coefficient
c.Cy_beta = 0.015; %[-] Sideslip coefficient
c.Cy_delta_r = 0.001; %[-] Rudder sideslip coefficient
c.CTy_alpha = 0.15; %[-] Angle-of-attack pitch coefficient
c.CTy_delta_e = 0.1; %[-] Elevator pitch coefficient
c.bq = 0.5; %[N m s] Resistive pitch torque coefficient
c.C_delta_a = 0.03; %[-] Aileron roll torque coefficient
c.bp = 1; %[N m s] Resistive roll torque coefficient
c.CTz_beta = 0.005; %[-] Sideslip yaw torque coefficient
c.CTz_delta_r = 0.04; %[-] Rudder yaw torque coefficient
c.br = 0.5; %[N m s] Resistive yaw torque coefficient
c.n_max = 170; %[rev/s] maximum propeller speed
c.D = 10*0.0254; %[m] propeller diameter
c.alpha_b = 5; %[inch] propeller pitch

%% Set the wind model constants in a structure named wind
wind.ThereIsWind = 1; %[0 or 1] flag to turn wind on (1) or off (0)
wind.uw_North = -1; %[m/s] constant wind speed from south to north
wind.vw_East = 0.6; %[m/s] constant wind speed from west to east
wind.ww_Down = -0.1; %[m/s] constant downward wind speed
wind.Lu = 200; %[m] x-axis gust parameter
wind.Lv = wind.Lu; %[m] y-axis gust parameter
wind.Lw = 50; %[m] z-axis gust parameter
wind.sigma_u = 1.06; %[m/s] x-axis gust parameter
wind.sigma_v = wind.sigma_u; %[m/s] y-axis gust parameter
wind.sigma_w = 0.7; %[m/s] z-axis gust parameter

%% set other constants, initialize arrays
dt = 0.01;
t = []; %[s] time array
u = []; %[m/s] forward velocity
xI = []; yI = []; zI = []; %[m] inertial positions
delta_t = 0.8; %initial throttle position
delta_e = 0; %Initial elevator position
delta_a = 0; %Initial aileron position
delta_r = 0; %Initial rudder position
ii = 0; %initialize the iteration counter

%% Run the simulation
h_fig = figure('WindowState','maximized'); hold on
%Set up the figure to accept key-press events
set(h_fig,'KeyPressFcn',@setKey);
set(h_fig,'KeyReleaseFcn',@resetKey);
```

```

while x(9) < 0, ii = ii + 1;
%Store the outputs
xI = [xI;x(7)];
yI = [yI;x(8)];
zI = [zI;x(9)];
t = [t;ii*dt];
u = [u;x(1)];

if wind.ThereIsWind
    %Get the wind gusts and North-East-Down constant wind
    uvw_gust = WindGustOutputs(xGust,wind);
    ned_wind = [wind.uw_North; wind.vw_East; wind.ww_Down];

    %Calculate the airspeeds and wind-frame parameters
    [Va, alpha, beta, ur] = AirspeedVariables(x(1:3), ...
        ned_wind, ...
        uvw_gust, ...
        x(10:13));
else
    %Calculate the airspeeds and wind-frame parameters
    [Va, alpha, beta, ur] = AirspeedVariables(x(1:3), ...
        [0;0;0], ...
        [0;0;0], ...
        x(10:13));
end

%Calculate the gravity forces
[fxg,fyg,fzg] = gravityForces(c.mass,x(10:13));

%Calculate the propeller forces and torques
[fxT, TxQ] = PropellerThrustAndTorque(delta_t,ur,c);

%Calculate the aerodynamic forces and torques
[fxAero,fyAero,fzAero,TxAero,TyAero,TzAero] = ...
    AeroForcesAndTorques(x,Va,alpha,beta, ...
        delta_e,delta_a,delta_r,c);

%Combine all forces and torques
fx = fxAero + fxT + fxg;
fy = fyAero + fyg;
fz = fzAero + fzg;
Tx = TxAero + TxQ;
Ty = TyAero;
Tz = TzAero;

%Use Runge-Kutta to calculate the airplane motion
k1 = AirplaneDynamics(x, fx, fy, fz, Tx, Ty, Tz, c);
k2 = AirplaneDynamics(x+dt*k1, fx, fy, fz, Tx, Ty, Tz, c);
x = x+dt/2*(k1+k2);

%Normalize the quaternions
qNorm = norm(x(10:13));
x(10:13) = x(10:13) / qNorm;

%Use Euler to calculate the gust dynamics

```

20.1 MATLAB Code for NED: The MATLAB Flight Simulator

```
k1g = GustDynamics(xGust,wind);
xGust(1:end-1) = xGust(1:end-1) + dt*k1g;
xGust(end) = Va;

if ~mod(ii,10)
    clf(h_fig)
    hold on
    DrawAirplane(x(7),x(8),x(9),x(10),x(11),x(12),x(13))
    drawnow
end
end

% plot the outputs
figure, scatter3(xI,yI,-zI) %show the 3D path of the airplane
xlabel('North')
ylabel('East')
zlabel('Up')
grid on

figure, plot(t, u*2.237) %Show the forward speed of the airplane
xlabel('Time (s)')
ylabel('Forward Speed [mph]')
grid on

%% set the key when it is pressed
function setKey(H,E)
switch E.Key
    case 'a'
        assignin('base','delta_r',-0.4);
    case 's'
        assignin('base','delta_t',0);
    case 'd'
        assignin('base','delta_r',0.4);
    case 'w'
        assignin('base','delta_t',0.9);
    case 'uparrow'
        assignin('base','delta_e',0.4);
    case 'downarrow'
        assignin('base','delta_e',-0.4);
    case 'leftarrow'
        assignin('base','delta_a',-0.4);
    case 'rightarrow'
        assignin('base','delta_a',0.4);
    otherwise
end
end

%% reset the key when it is released
function resetKey(H,E)
% trim values
delta_e_trim = 0.065;
delta_a_trim = -0.04;
switch E.Key
    case 'a'
```

```

        assignin('base','delta_r',0);
case 's'
    assignin('base','delta_t',0);
case 'd'
    assignin('base','delta_r',0);
case 'w'
    assignin('base','delta_t',0);
case 'uparrow'
    assignin('base','delta_e',delta_e_trim);
case 'downarrow'
    assignin('base','delta_e',delta_e_trim);
case 'leftarrow'
    assignin('base','delta_a',delta_a_trim);
case 'rightarrow'
    assignin('base','delta_a',delta_a_trim);
otherwise
end
end

%% This function calculates the wind gust outputs
function uvw_gust = WindGustOutputs(xGust,wind)
Va = xGust(end);
Cvw = [Va/sqrt(3*wind.Lv), 1];
Cww = [Va/sqrt(3*wind.Lw), 1];
uvw_gust = [xGust(1);Cvw*xGust(2:3);Cww*xGust(4:5)];
end

%% This function calculates the inertial-to-body frame rotation matrix
function RI2b = R_I2b(q)

%Extract the components of the quaternion q
e0 = q(1);
e1 = q(2);
e2 = q(3);
e3 = q(4);

%Get the rotation matrix
RI2b = [e0^2+e1^2-e2^2-e3^2, 2*(e0*e3-e1*e2), 2*(e1*e3-e0*e2);...
         2*(e1*e2-e0*e3), e0^2-e1^2+e2^2-e3^2, 2*(e0*e1+e2*e3);...
         2*(e0*e2+e1*e3), 2*(e2*e3-e0*e1), e0^2-e1^2-e2^2+e3^2];
end

%% This function calculates Va, alpha, and beta
function [Va, alpha, beta, ur] = AirspeedVariables(...%
uvw, ned_wind, uvw_gust, q)

%get the Inertial-to-body frame rotation matrix
RI2b = R_I2b(q);

%get the wind velocities in the body-frame
uvw_wind = RI2b*ned_wind + uvw_gust;

%get the relative velocities
uvw_r = uvw - uvw_wind;

```

20.1 MATLAB Code for NED: The MATLAB Flight Simulator

```
%Magnitude of the relative airspeed
Va = sqrt(uvw_r(1)^2+uvw_r(2)^2+uvw_r(3)^2);

%alpha and beta
if Va > 0.1
    alpha = atan(uvw_r(3)/Va);
    beta = asin(max(-1,min(1,uvw_r(2)/Va)));
else
    alpha = 0;
    beta = 0;
end

%Get the relative forward velocity
ur = uvw_r(1);
end

%% This function calculates the gravity forces in the body frame
function [fxg,fyg,fzg] = gravityForces(m,q)
% [fxg,fyg,fzg] = gravityForces(m,q)
%
%OUTPUTs:
% fxg: (N) gravity force in the body-fixed x-direction
% fyg: (N) gravity force in the body-fixed y-direction
% fzg: (N) gravity force in the body-fixed z-direction
%
%INPUTS:
% m: (kg) airplane mass
% e0: (unitless) quaternion
% e1: (unitless) quaternion
% e2: (unitless) quaternion
% e3: (unitless) quaternion

%extract the quaternions
e0 = q(1);
e1 = q(2);
e2 = q(3);
e3 = q(4);

g = 9.81; %(m/s2) gravitational acceleration

mg = m*g; %(N) gravitational force

F = mg*[2*(e1*e3-e0*e2);...
          2*(e0*e1+e2*e3);...
          e0^2-e1^2-e2^2+e3^2];

%Gravity force components
fxg = F(1);
fyg = F(2);
fzg = F(3);
end

%% This function calculates the propeller thrust and torque
function [fxT, TxQ] = PropellerThrustAndTorque(delta_t,ur,c)
%[fxT, TxQ] = PropellerThrustAndTorque(delta_t,ur,c)
```

```

%
%OUTPUTS:
%fxT: (N) propeller thrust in the body-fixed x-axis
%TxQ: (N m) propeller torque around the body-fixed x-axis
%
%INPUTS:
%delta_t: (0-1) throttle command
%ur: (m/s) relative airspeed in the body-fixed x-direction
%c: A structure with at least the following items
%  c.n_max: (rev/s) Maximum propeller speed
%  c.D: (m) propeller diameter
%  c.alpha_b: (inch) propeller pitch
%  c.rho: (kg/m3) air density

n = c.n_max * (1-(1-delta_t)^2);

if n < 0.1
    %motor is off
    fxT = 0;
    TxQ = 0;
else
    J = ur / n / c.D;
    %prop torque
    zQ = 1/(1+exp(10*(0.16+0.05*c.alpha_b-J)));
    CQL = 0.0121-0.017*(J-0.1*(c.alpha_b-5));
    CQu = 0.0057+0.00125*(c.alpha_b-5)-0.0005*J;
    CQ = zQ*CQL+(1-zQ)*CQu;
    TxQ = c.rho*n^2*c.D^5*CQ;
    %prop thrust
    z = 1/(1+exp(20*(0.14+0.018*c.alpha_b-0.6*J)));
    CTL = 0.11-0.18*(J-0.105*(c.alpha_b-5));
    CTu = 0.1+0.009*(c.alpha_b-5)-0.065*J;
    CT = z*CTL+(1-z)*CTu;
    fxT = c.rho*n^2*c.D^4*CT;
end
end

%% This function calculates the aerodynamic forces and torques
function [fxAero,fyAero,fzAero,TxAero,TyAero,TzAero] = ...
    AeroForcesAndTorques(x,Va,alpha,beta, ...
    delta_e,delta_a,delta_r,c)
%[fxAero,fyAero,fzAero,TxAero,TyAero,TzAero] = ...
%    AeroForcesAndTorques(x,Va,alpha,beta, ...
%    delta_e,delta_a,delta_r,c)
%
%OUTPUTS:
% fxAero: (N) aerodynamic force in the body-fixed x-direction
% fyAero: (N) aerodynamic force in the body-fixed y-direction
% fzAero: (N) aerodynamic force in the body-fixed z-direction
% TxAero: (N m) aerodynamic torque around the body-fixed x-axis
% TyAero: (N m) aerodynamic torque around the body-fixed y-axis
% TzAero: (N m) aerodynamic torque around the body-fixed z-axis
%
%INPUTS:
% x: a vector of 13 state variables x=[u;v;w;p;q;r;xI;yI;zI;e0;e1;e2;e3]

```

20.1 MATLAB Code for NED: The MATLAB Flight Simulator

```
% Va: (m/s) airspeed
% alpha: (rad) angle of attack
% beta: (rad) side-slip angle
% delta_e: (-1 to 1) elevator command
% delta_a: (-1 to 1) aileron command
% delta_r: (-1 to 1) rudder command
% c: a struct with constants

faero = 0.5*c.rho*Va^2*c.A;

fxAero = -faero*(Drag(alpha,c)+c.CD_delta_e*delta_e);

fyAero = -faero*(c.Cy_beta*beta+c.Cy_delta_r*delta_r);

fzAero = -faero*(Lift(alpha,c)+c.CL_delta_e*delta_e);

TxAero = faero*c.S*c.C_delta_a*delta_a-c.bp*x(4);

TyAero = -faero*c.c*...
    (c.CTy_alpha*alpha+c.CTy_delta_e*delta_e) - c.br * x(5);
TzAero = faero*c.c*...
    (c.CTz_beta*beta+c.CTz_delta_r*delta_r)-c.bq*x(6);
end

%% Drag coefficient
function CD = Drag(alpha,c)
% These values were curve-fit from a drag coefficient graph
CD = c.CD0 + c.CD2 * alpha^2;
end

%% Lift coefficient
function CL = Lift(alpha,c)
% These values were curve-fit from a lift coefficient graph
CL = c.CL0 + c.CL1/ (c.CL2 + exp(-c.CL3 * (alpha + c.CL4)));
end

%% This function calculates the Runge-Kutta k-slopes for the airplane
function k = AirplaneDynamics(x, fx, fy, fz, Tx, Ty, Tz, c)
% k = AirplaneDynamics(x,fx, fy, fz, Tx, Ty, Tz, c)
%
%AirplaneDynamics calculates the runge kutta k variable for the airplane
% equations of motion give the current state x, the forces, and torques.
%k: the runge kutta k variable (k1, k2, k3, or k4)
%x: the state vector, it has the following 13 state variables:
% x=[u;v;w;p;q;r;xI;yI;zI;e0;e1;e2;e3]
% x(1): u (m/s) is the x-velocity of the airplane in the body frame
% x(2): v (m/s) is the y-velocity of the airplane in the body frame
% x(3): w (m/s) is the z-velocity of the airplane in the body frame
% x(4): p (rad/s) roll rate about the body frame x-axis
% x(5): q (rad/s) yaw rate about the body frame y-axis
% x(6): r (rad/s) pitch rate about the body frame z-axis
% x(7): xI (m) is the north position of the airplane in the inertial frame
% x(8): yI (m) is the east position of the airplane in the inertial frame
% x(9): zI (m) is the down position of the airplane in the inertial frame
% x(10): e0 (rad) is a quaternion. It is related to roll, pitch, and yaw
```

```
% x(11): e1 (rad) is a quaternion. It is related to roll, pitch, and yaw
% x(12): e2 (rad) is a quaternion. It is related to roll, pitch, and yaw
% x(13): e3 (rad) is a quaternion. It is related to roll, pitch, and yaw
%fx: (N) force in the x-direction of the body frame
%fy: (N) force in the y-direction of the body frame
%fz: (N) force in the z-direction of the body frame
%Tx: (N m) Torque about the x-axis in the body frame
%Ty: (N m) Torque about the y-axis in the body frame
%Tz: (N m) Torque about the z-axis in the body frame
%c: a structure of constant parameters with
%c.mass: (kg) mass of the airplane
%c.Jxx: (kg m2) moment of inertia about the body x-axis
%c.Jyy: (kg m2) moment of inertia about the body y-axis
%c.Jzz: (kg m2) moment of inertia about the body z-axis
%c.Jxz: (kg m2) Product of inertia

u=x(1);v=x(2);w=x(3);p=x(4);q=x(5);r=x(6);
e0=x(10);e1=x(11);e2=x(12);e3=x(13);

k13 = [r*v-q*w;p*w-r*u;q*u-p*v]+1/c.mass*[fx;fy;fz];

Jinv = 1/(c.Jxx*c.Jyy*c.Jzz-c.Jyy*c.Jxz^2) * ...
[c.Jyy*c.Jzz, 0, c.Jxz*c.Jyy;...
0, c.Jxx*c.Jzz-c.Jxz^2, 0;...
c.Jyy*c.Jxz, 0, c.Jxx*c.Jyy];
k46 = Jinv * ([q*r*(c.Jyy-c.Jzz)+p*q*c.Jxz+Tx;...
p*r*(c.Jzz-c.Jxx)+(r^2-p^2)*c.Jxz+Ty;...
p*q*(c.Jxx-c.Jyy)-q*r*c.Jxz + Tz]);

k79 = [e0^2+e1^2-e2^2-e3^2, 2*(e1*e2-e0*e3), 2*(e0*e2+e1*e3);...
2*(e0*e3+e1*e2), e0^2-e1^2+e2^2-e3^2, 2*(e2*e3-e0*e1);...
2*(e1*e3-e0*e2), 2*(e0*e1+e2*e3) e0^2-e1^2-e2^2+e3^2]*[u;v;w];

k1013 = 1/2*[-e1, -e2, -e3;...
e0, -e3, e2;...
e3, e0, -e1;...
-e2, e1, e0] * [p;q;r];

k = [k13;k46;k79;k1013];
end

%% This function calculates the Runge-Kutta k-slopes for the wind gusts
function k = GustDynamics(xGust,wind)
%k = GustDynamics(xGust,Va,wind)
%
%INPUTS:
%xGust: an array with xGust = [uwg; x1; x2; x3; x4; Va] where
%xGust(1): [m/s] uwg is the wind gusts in the body-frame x-axis
%xGust(2): x1 is a state variable for the wind gusts in the y-axis
%xGust(3): x2 is a state variable for the wind gusts in the y-axis
%xGust(4): x3 is a state variable for the wind gusts in the z-axis
%xGust(5): x4 is a state variable for the wind gusts in the z-axis
%xGust(6): [m/s] Va is the previous relative airspeed of the plane
%
%OUTPUT:
```

20.1 MATLAB Code for NED: The MATLAB Flight Simulator

```
%k: an array for Runge-Kutta or Euler solvers, k = f(x,u)

%Intermediate matrices
Auw = -xGust(end)/wind.Lu;
Buw = wind.sigma_u*sqrt(2*xGust(end)/pi/wind.Lu);

Avw = [0,1;-(xGust(end)/wind.Lv)^2,-2*xGust(end)/wind.Lv];
Bvw = [0;wind.sigma_v*sqrt(3*xGust(end)/pi/wind.Lv)];

Awv = [0,1;-(xGust(end)/wind.Lw)^2,-2*xGust(end)/wind.Lw];
Bww = [0;wind.sigma_w*sqrt(3*xGust(end)/pi/wind.Lw)];

%get k
k = [Auw*xGust(1)+Buw*randn;...
      Avw*xGust(2:3)+Bvw*randn;...
      Awv*xGust(4:5)+Bww*randn];
end

%% These functions draw the airplane
function DrawAirplane(xNorth,yEast,zDown,e0,e1,e2, e3)
%get the airplane vertices, faces, and colors
[Vertices, Faces, Colors] = AirplaneGraphics;

% transform vertices from North-East-Down to East-North-Up
%(for MATLAB rendering)
R_ENU = [0,1,0; ...Convert East to North
         1,0,0; ...Convert North to East
         0,0,-1]; % Convert Down to Up
Vertices = (R_ENU*Vertices)';

%Convert from the body reference frame to the inertial frame
%first rotate, then translate (north is east, east is north, -down is up)
Vertices = BodyToInertialRotation(Vertices, e0, e2, e1, -e3);
Vertices = BodyToInertialTranslation(Vertices, yEast, xNorth, -zDown);

%draw the airplane using the "patch" command
%To learn more about the patch command, type "help patch" in the command
% window
hold on
patch('Vertices', Vertices, 'Faces', Faces, ...
       'FaceVertexCData',Colors, 'FaceColor','flat');
%north is east, east is north, -down is up
xlabel('East (y)')
ylabel('North (x)')
zlabel('Up (-z)')
view(32,47) % set the view angle for the figure
axis([yEast-10,yEast+10,xNorth-10,xNorth+10,-zDown-10,-zDown+10]); %set axis limits
pbaspect([1 1 1]) %Set the aspect ratio
grid on
hold off
end

function RotatedPoints = BodyToInertialRotation(points, e0, e1, e2, e3)
%RotatedPoints = BodyToInertialRotation(points, e0, e1, e2, e3)
%
```

```
% points: location of points in the body frame that are to be
%
%get the quaternion body to inertial frame rotation matrix
Rb2I = [e0^2+e1^2-e2^2-e3^2, ...
    2*(e1*e2-e0*e3), ...
    2*(e0*e2+e1*e3); ...
    ...
    2*(e0*e3+e1*e2), ...
    e0^2-e1^2+e2^2-e3^2, ...
    2*(e2*e3-e0*e1); ...
    ...
    2*(e1*e3-e0*e2), ...
    2*(e0*e1+e2*e3), ...
    e0^2-e1^2-e2^2+e3^2];

%Rotate the points
RotatedPoints = (Rb2I*points)';
end

function translatedPoints = BodyToInertialTranslation(points, xNorth, yEast, zDown)
translatedPoints = zeros(size(points));
translatedPoints(:,1) = points(:,1)+xNorth;
translatedPoints(:,2) = points(:,2)+yEast;
translatedPoints(:,3) = points(:,3)+zDown;
end

function [V, F, colors] = AirplaneGraphics
% [V, F, colors] = AirplaneGraphics
%
% V is a matrix of vertices (3D location of vertices)
% F is a matrix of faces that use the vertices to define the corners
% colors is a matrix of colors corresponding to the faces
%
%Define the vertices of the airplane, its faces, and its colors

% Define the vertices (physical location of vertices)
% For the flying wing airplane
% the locations are [xNorth,yUp,zEast]
V = 10*[...
    0.08,0,0;...
    -0.14,0,0.5;...
    -0.2,0,0.5;...
    -0.22,0,0.5;...
    -0.19,0,0.1;...
    -0.14,0,0.1;...
    -0.12,0,0;...
    -0.14,0,-0.1;...
    -0.19,0,-0.1;...
    -0.22,0,-0.5;...
    -0.2,0,-0.5;...
    -0.14,0,-0.5;...
    0.03,0.03,0;...
    -0.18,0.015,0.5;...
    -0.18,0.015,-0.5;...
```

20.1 MATLAB Code for NED: The MATLAB Flight Simulator

```
    ] * [1,0,0;0,0,-1;0,1,0];
%define the faces
F = [1,2,7;...
    7,2,3;...
    3,5,6;...
    4,5,3;...
    9,10,11;...
    11,8,9;...
    11,12,7;...
    12,1,7;...
    1,2,13;...
    13,2,14;...
    13,14,7;...
    7,14,3;...
    7,11,15;...
    15,13,7;...
    15,12,13;...
    12,1,13;...
];
% define colors for each face
myblue = [0, 0, 1];
mygrey = [0.1,0.1,0.1];
mywhite = [0.9,0.9,0.9];

colors = [...  
    mygrey;...
    mygrey;...
    myblue;...
    myblue;...
    myblue;...
    myblue;...
    myblue;...
    mygrey;...
    mygrey;...
    mywhite;...
    myblue;...
    mywhite;...
    mywhite;...
    mywhite;...
    myblue;...
    mywhite;...
];
end
```


Chapter 21

C++ and Arduino Code for Autonomous Flight

Contents

21.1	function_QuaternionOrientationEstimator	541
21.1.1	function_QuaternionOrientationEstimator.h	541
21.1.2	function_QuaternionOrientationEstimator.cpp	543
21.2	function_Quaternion2Euler	550
21.2.1	function_Quaternion2Euler.h	550
21.2.2	function_Quaternion2Euler.cpp	550
21.3	function_GPSwaypointAlgorithm	550
21.3.1	function_GPSwaypointAlgorithm.h	550
21.3.2	function_GPSwaypointAlgorithm.cpp	551
21.4	function_FeedbackControlOfDelta_tear	552
21.4.1	function_FeedbackControlOfDelta_tear.h	552
21.4.2	function_FeedbackControlOfDelta_tear.cpp	552
21.5	Arduino Code for the Autopilot	554

This chapter provides C++ and Arduino code for autonomous flight. Some of the Arduino libraries are contributed by the Arduino community. Code for those libraries is available through online sources, such as Arduino reference documentation, and is not included in this book. The code of this chapter comes with no guarantees or warranties of any kind. Sensor and actuator calibration is required for this code to work, and hence, it is not simply plug-and-play. The purpose is to provide an example to help the reader develop his or her own autonomous flight strategies.

21.1 function_QuaternionOrientationEstimator

21.1.1 function_QuaternionOrientationEstimator.h

```
1 #pragma once
2
3 void function_QuaternionOrientationEstimator(
4     double quaternion_new[4], //OUTPUT: updated quaternion orientation
5     double geomagnetic_new[3], //OUTPUT: [uT] low-pass filtered estimate of ↵
       inertial frame geomagnetic vector
6     const double quaternion[4], //INPUT: previous quaternion orientation
7     const double geomagnetic[3], //INPUT: [uT] low-pass filtered estimate ↵
       of inertial frame geomagnetic vector
8     const double gyrometer[3], //INPUT: [rad/s] 3-axis gyro data with bias ↵
       removed
9     const double gravity[3], //INPUT: [any] x-front, y-right, z-down body- ↵
       frame gravity vector
10    const double magnetometer[3], //INPUT: [any] x-front, y-right, z-down ↵
       calibrated magnetometer signal any units, bias removed
11    const double dt, //INPUT: [s] time-step
12    const double beta, //INPUT: [< 1] small-valued positive tuning ↵
       parameter
13    const double tau_m //INPUT: [s] low-pass filter time constant for ↵
       geomagnetic vector
14 );
15
```

21.1 function_QuaternionOrientationEstimator

21.1.2 function_QuaternionOrientationEstimator.cpp

```
1 #include "function_QuaternionOrientationEstimator.h"
2 #include <math.h> //To use sqrt
3
4 //create the max function
5 static double max(const double a, const double b) {
6     if (a >= b) {
7         return a;
8     }
9     else {
10         return b;
11     }
12 }
13
14 //Inertial to body rotation matrix
15 static void Rotation_I2b(
16     double Ri2b[3][3], //OUTPUT: Rotation matrix from the inertial to      ↵
17     const double q[4] //INPUT: unit quaternion orientation
18 ) {
19     double e0 = q[0];
20     double e1 = q[1];
21     double e2 = q[2];
22     double e3 = q[3];
23
24     /*local variables*/
25     double e0_2 = e0 * e0;
26     double e1_2 = e1 * e1;
27     double e2_2 = e2 * e2;
28     double e3_2 = e3 * e3;
29     double e0e1 = e0 * e1;
30     double e0e2 = e0 * e2;
31     double e0e3 = e0 * e3;
32     double e1e2 = e1 * e2;
33     double e1e3 = e1 * e3;
34     double e2e3 = e2 * e3;
35
36     Ri2b[0][0] = (e0_2 + e1_2 - e2_2 - e3_2);
37     Ri2b[0][1] = 2.0f * (e0e3 + e1e2);
38     Ri2b[0][2] = 2.0f * (e1e3 - e0e2);
39     Ri2b[1][0] = 2.0f * (e1e2 - e0e3);
40     Ri2b[1][1] = (e0_2 - e1_2 + e2_2 - e3_2);
41     Ri2b[1][2] = 2.0f * (e0e1 + e2e3);
42     Ri2b[2][0] = 2.0f * (e0e2 + e1e3);
43     Ri2b[2][1] = 2.0f * (e2e3 - e0e1);
44     Ri2b[2][2] = (e0_2 - e1_2 - e2_2 + e3_2);
45
46 }
47
48 //Multiply a 3x3 matrix A with a 3x1 vector b: c = A*b
```

```
49 static void mat3_vec3_multiply(double c[3], //OUTPUT: 3x1 vector solution
50     to c=A*b
51     const double A[3][3], //INPUT: 3x3 matrix
52     const double b[3] //INPUT: 3x1 vector
53 ) {
54     for (int ii = 0; ii < 3; ii++) {
55         c[ii] = 0.0;
56         for (int kk = 0; kk < 3; kk++) {
57             c[ii] += A[ii][kk] * b[kk];
58         }
59     }
60 }
61 static void mat3_transpose(double AT[3][3], //OUTPUT: transposed matrix A^T
62     const double A[3][3]//INPUT: matrix to be transposed
63 ) {
64     for (int ii = 0; ii < 3; ii++) {
65         for (int jj = 0; jj < 3; jj++) {
66             AT[ii][jj] = A[jj][ii]; //Transpose the matrix
67         }
68     }
69 }
70
71 //calculate the cross-product c=axb
72 static void vec3_cross(double c[3], //OUTPUT: c = a x b
73     const double a[3], //INPUT
74     const double b[3] //INPUT
75 ) {
76     //Calculate the cross product c = a x b
77     c[0] = a[1] * b[2] - b[1] * a[2];
78     c[1] = -a[0] * b[2] + b[0] * a[2];
79     c[2] = a[0] * b[1] - b[0] * a[1];
80 }
81
82 void function_QuaternionOrientationEstimator(
83     double quaternion_new[4], //OUTPUT: updated quaternion orientation
84     double MI_geomagnetic_new[3], //OUTPUT: [uT] low-pass filtered
85     estimate of inertial frame geomagnetic vector
86     const double quaternion[4], //INPUT: previous quaternion orientation
87     const double MI_geomagnetic[3], //INPUT: [uT] low-pass filtered
88     estimate of inertial frame geomagnetic vector
89     const double gyrometer[3], //INPUT: [rad/s] 3-axis gyro data with bias
90     removed
91     const double gravity[3], //INPUT: [any] x-front, y-right, z-down body-
92     frame gravity vector
93     const double magnetometer[3], //INPUT: [any] x-front, y-right, z-down
94     calibrated magnetometer signal any units, bias removed
95     const double dt, //INPUT: [s] time-step
96     const double beta, //INPUT: [< 1] small-valued positive tuning
```

```
    parameter
92     const double tau_m //INPUT: [s] low-pass filter time constant for      ↵
93     geomagnetic vector
94 }
95 //ensure that the quaternion is normalized
96 double q[4];
97 double q_den = sqrt(quaternion[0] * quaternion[0] +
98                      quaternion[1] * quaternion[1] +
99                      quaternion[2] * quaternion[2] +
100                     quaternion[3] * quaternion[3]);
101 if (q_den < 0.00001) {
102     //The quaternion needs to be reset because its magnitude is      ↵
103     basically zero
104     q[0] = 1.0;
105     q[1] = 0.0;
106     q[2] = 0.0;
107     q[3] = 0.0;
108 } else {
109     for (int ii = 0; ii < 4; ii++) {
110         //normalize the quaternion
111         q[ii] = quaternion[ii] / q_den;
112     }
113 }
114
115 //Get the inertial-to-body frame rotation matrix
116 double RI2b[3][3];
117 Rotation_I2b(RI2b, q);
118
119 //Get the body-to-inertial frame rotation matrix
120 double R_b2I[3][3];
121 mat3_transpose(R_b2I, RI2b);
122
123 //Get the raw estimate of the geomagnetic vector
124 double B[3];
125 mat3_vec3_multiply(B, R_b2I, magnetometer);
126
127 //Get the filtered and corrected estimate of the geomagnetic vector
128 double a = tau_m / (tau_m + dt); //Low-pass filter coefficient      ↵
129     (backward Euler)
130 MI_geomagnetic_new[0] = a * MI_geomagnetic[0] + (1.0 - a) * sqrt(B[0]      ↵
131     * B[0] + B[1] * B[1]);
132 MI_geomagnetic_new[1] = a * MI_geomagnetic[1]; //correct B[1] to be      ↵
133     zero
134 MI_geomagnetic_new[2] = a * MI_geomagnetic[2] + (1.0 - a) * B[2];
135
136 //Normalize to get the inertial-frame geomagnetic unit vector
137 double hat_mi[3];
```

```

...Estimator\function_QuaternionOrientationEstimator.cpp
135     double mI_den = max(0.1, sqrt(MI_geomagnetic[0] * MI_geomagnetic[0] +
136                             MI_geomagnetic[1] * MI_geomagnetic[1] +
137                             MI_geomagnetic[2] * MI_geomagnetic[2]));
138     for (int ii = 0; ii < 3; ii++) {
139         //Normalize the inertial frame geomagnetic vector
140         hat_mI[ii] = MI_geomagnetic[ii] / mI_den;
141     }
142
143     //get the quaternion's estimate of the body-frame magnetic unit vector
144     double hat_m[3];
145     mat3_vec3_multiply(hat_m, RI2b, hat_mI); //hat_m = RI2b*hat_mI
146
147     //get the quaternion's estimate of the body-frame gravity vector
148     double hat_g[3];
149     for (int ii = 0; ii < 3; ii++) {
150         hat_g[ii] = RI2b[ii][2]; //The gravity vector is the third column ↵
151         of RI2b
152     }
153
154     //Normalize the accelerometer's measurement of gravity
155     double gak[3];
156     double g_den = max(0.01, sqrt(gravity[0] * gravity[0] +
157                               gravity[1] * gravity[1] +
158                               gravity[2] * gravity[2]));
159     for (int ii = 0; ii < 3; ii++) {
160         //normalize the gravity vector
161         gak[ii] = gravity[ii] / g_den;
162     }
163
164     //Normalize the body-frame magnetometer signal
165     double mk[3];
166     double m_den = max(0.01, sqrt(magnetometer[0] * magnetometer[0] +
167                               magnetometer[1] * magnetometer[1] +
168                               magnetometer[2] * magnetometer[2]));
169     for (int ii = 0; ii < 3; ii++) {
170         //Normalize the magnetometer signal
171         mk[ii] = magnetometer[ii] / m_den;
172     }
173
174     //get the difference between the gravity vectors and magnetometer
175     //vectors
176     double dg[3];
177     double dm[3];
178     for (int ii = 0; ii < 3; ii++) {
179         dg[ii] = gak[ii] - hat_g[ii]; //difference between gravity vectors
180         dm[ii] = mk[ii] - hat_m[ii]; //difference between magnetic vectors
181     }
182
183     //get the cross products for the gravity and magnetic vectors

```



```
230     for (int ii = 0; ii < 4; ii++)
231         quaternion_new[ii] = qn[ii] / qn_den; //New estimate of the
232             quaternion orientation
233
234     //If any output is infinite or NaN, reset everything
235     if (isinf(quaternion_new[0]) || isnan(quaternion_new[0])
236         || isinf(quaternion_new[1]) || isnan(quaternion_new[1])
237         || isinf(quaternion_new[2]) || isnan(quaternion_new[2])
238         || isinf(quaternion_new[3]) || isnan(quaternion_new[3])
239         || isinf(MI_geomagnetic_new[0]) || isnan(MI_geomagnetic_new[0])
240         || isinf(MI_geomagnetic_new[1]) || isnan(MI_geomagnetic_new[1])
241         || isinf(MI_geomagnetic_new[2]) || isnan(MI_geomagnetic_new[2])
242     )
243     {
244         //Some problem happened, reset everything
245         //Reset quaternions
246         quaternion_new[0] = 1.0f;
247         for (unsigned int ii = 1; ii < 4; ii++)
248             quaternion_new[ii] = 0.0f;
249
250         //Reset the inertial-frame geomagnetic vector
251         MI_geomagnetic_new[0] = 20.7; //53*cosd(67)
252         MI_geomagnetic_new[1] = 0.0;
253         MI_geomagnetic_new[2] = 48.8; //53*sind(67)
254     }
255 }
```

21.2 function_Quaternion2Euler

21.2.1 function_Quaternion2Euler.h

```
#pragma once

//Declare the function
void function_Quaternion2Euler(double* roll,
    double* pitch, double* yaw, const double q[4]);
```

21.2.2 function_Quaternion2Euler.cpp

```
#include "function_Quaternion2Euler.h"
#include <math.h> //To use atan2 and asin
#define max(a,b) ((a<b)?b:a) //Macro that defines the max function
#define min(a,b) ((a>b)?b:a) //Macro that defines the min function
#define clip(minv, a, maxv) (max(minv, min(a, maxv))) //Clip function

//Define the function
void function_Quaternion2Euler(double* roll, double* pitch,
    double* yaw, const double q[4])
{
    *roll = atan2(2.0 * (q[0] * q[1] + q[2] * q[3]),
        q[0] * q[0] - q[1] * q[1] - q[2] * q[2] + q[3] * q[3]);
    *pitch = asin(clip(-1.0, 2.0 * (q[0] * q[2] - q[1] * q[3]), 1.0));
    *yaw = atan2(2.0 * (q[0] * q[3] + q[1] * q[2]),
        q[0] * q[0] + q[1] * q[1] - q[2] * q[2] - q[3] * q[3]);
}
```

21.3 function_GPSwaypointAlgorithm

21.3.1 function_GPSwaypointAlgorithm.h

```
#pragma once

void function_GPSwaypointAlgorithm(
    double dXI[3], //OUTPUT: [xI,yI,zI] (m) displacement to the target
    const double latitude, //INPUT: [lat](deg) current latitude angle
    const double longitude, //INPUT: [lon](deg) current longitude angle
    const double altitude, //INPUT: [-zI](m) current altitude
    const double target_lat, //INPUT: [lat](deg) target latitude
```

21.3 function_GPSwaypointAlgorithm

```
const double target_lon, //INPUT: [lon](deg) target longitude
const double target_alt //INPUT: [-zI](m) target altitude
);
```

21.3.2 function_GPSwaypointAlgorithm.cpp

```
#include "function_GPSwaypointAlgorithm.h"
#include <math.h>

void function_GPSwaypointAlgorithm(
    double dXI[3], //OUTPUT: [xI,yI,zI] (m) displacement to the target
    const double latitude, //INPUT: [lat](deg) current latitude angle
    const double longitude, //INPUT: [lon](deg) current longitude angle
    const double altitude, //INPUT: [-zI](m) current altitude
    const double target_lat, //INPUT: [lat](deg) target latitude
    const double target_lon, //INPUT: [lon](deg) target longitude
    const double target_alt //INPUT: [-zI](m) target altitude
)
{
    const double r = 6371000; // (m)Earth's radius
    const double pi = 3.14159265359;
    const double pi_180 = pi / 180.0;

    //convert to radians
    double lat = latitude * pi_180;
    double lon = longitude * pi_180;
    double latT = target_lat * pi_180;
    double lonT = target_lon * pi_180;

    //Calculate the change in longitude
    double abs_dlon = fabs(lonT - lon);
    double dlon;
    if (abs_dlon <= pi) {
        dlon = lonT - lon;
    }
    else {
        if (lonT > lon) {
            dlon = lonT - (lon + 2.0 * pi);
        }
        else {
            dlon = (2.0 * pi + lonT) - lon;
        }
    }
    //Calculate the displacement from the origin (m)
    dXI[0] = r * (latT - lat);
    if (dlon < 0) {
        dXI[1] = -r * acos((cos(dlon) - 1.0) * pow(cos(lat), 2) + 1);
    }
}
```

```

    else {
        dXI[1] = r * acos((cos(dlon) - 1.0) * pow(cos(lat), 2) + 1);

    }
    dXI[2] = -(target_alt - altitude);
}

```

21.4 function_FeedbackControlOfDelta_tear

21.4.1 function_FeedbackControlOfDelta_tear.h

```

#pragma once

void function_FeedbackControlOfDelta_tear(
    double* delta_t, // [0,1] throttle command
    double* delta_e, // [-1,1] elevator command
    double* delta_a, // [-1,1] aileron command
    double* delta_r, // [-1,1] rudder command
    const double dXI[3], // (m) x,y, and z displacements from the target
    const double gyro_x, // (rad/s) gyro x-axis angular velocity
    const double roll, // (rad) phi_x actual roll euler angle
    const double pitch, // (rad) theta_y actual pitch euler angle
    const double yaw, // (rad) psi_z actual yaw euler angle
    const double delta_t_d, // (0-1) desired throttle at cruise speed
    const double kp_throttle, // (-) k_{p,t} proportional throttle gain
    const double kp_roll, // (-) k_{p,phi} proportional roll gain
    const double kd_roll, // (-) k_{d,phi} derivative roll gain
    const double kp_pitch, // (-) k_{p,theta} proportional pitch gain
    const double kp_yaw, // (-) k_{p,psi} proportional yaw gain
    const double kp_rudder, // (-) k_{p,psi} proportional rudder yaw gain
    const double k_theta_psi, // (-) k_{theta,psi} proportional yaw-pitch
    const double max_roll, // [limit](rad) max roll angle limit
    const double max_pitch // [limit](rad) max pitch angle limit
);

```

21.4.2 function_FeedbackControlOfDelta_tear.cpp

```

#include "function_FeedbackControlOfDelta_tear.h"
#include <math.h>

#define max(a,b) ((a<b)?b:a) //Macro that defines the max function
#define min(a,b) ((a>b)?b:a) //Macro that defines the min function

```

21.4 function_FeedbackControlOfDelta_tear

```
static double sign(const double x) {
    if (x >= 0.0)
        return 1.0;
    else
        return -1.0;
}

static double clip(double minv, double a, double maxv) {
    if (a < minv)
        return minv;
    else if (a > maxv)
        return maxv;
    else
        return a;
}

static double shortestAngle(const double a2, const double a1) {
    double delta_alpha = acos(clip(-1.0,
        cos(a1) * cos(a2) + sin(a1) * sin(a2), 1.0));
    if (fabs(a2 - a1) - delta_alpha > 0.1) {
        return (-sign(a2 - a1) * delta_alpha);
    }
    else {
        return (a2 - a1);
    }
}

void function_FeedbackControlOfDelta_tear(
    double* delta_t, // [0,1] throttle command
    double* delta_e, // [-1,1] elevator command
    double* delta_a, // [-1,1] aileron command
    double* delta_r, // [-1,1] rudder command
    const double dXI[3], // (m) x,y, and z displacements from the target
    const double gyro_x, // (rad/s) gyro x-axis angular velocity
    const double roll, // (rad) phi_x actual roll euler angle
    const double pitch, // (rad) theta_y actual pitch euler angle
    const double yaw, // (rad) psi_z actual yaw euler angle
    const double delta_t_d, // (0-1) desired throttle at cruise speed
    const double kp_throttle, // (-) k_{p,t} proportional throttle gain
    const double kp_roll, // (-) k_{p,phi} proportional roll gain
    const double kd_roll, // (-) k_{d,phi} derivative roll gain
    const double kp_pitch, // (-) k_{p,theta} proportional pitch gain
    const double kp_yaw, // (-) k_{p,psi} proportional yaw gain
    const double kp_rudder, // (-) k_{p,psi} proportional rudder yaw gain
    const double k_theta_psi, // (-) k_{theta,psi} proportional yaw-pitch
    const double max_roll, // [limit] (rad) max roll angle limit
    const double max_pitch // [limit] (rad) max pitch angle limit
)
{
    // Calculate the desired roll, pitch, and yaw
    double roll_d = 0.0;
    double pitch_d = asin(clip(-1.0, -dXI[2] /
        sqrt(dXI[0] * dXI[0] + dXI[1] * dXI[1] + dXI[2] * dXI[2]), 1.0));
}
```

```
double yaw_d = atan2(dXI[1], dXI[0]);  
  
//Get the saturated desired pitch command  
double theta_y_r = clip(-max_pitch, pitch_d, max_pitch);  
  
//Get the difference between the commanded and actual pitch  
double Delta_theta;  
Delta_theta = shortestAngle(theta_y_r, pitch);  
  
//Yaw error  
double Delta_yaw = shortestAngle(yaw_d, yaw);  
  
//Get the absolute yaw angle error  
double abs_yaw = fabs(Delta_yaw);  
  
//saturated roll command due to yaw error  
double phi_x_r = clip(-max_roll, kp_yaw * Delta_yaw, max_roll);  
  
//roll error  
double Delta_phi = shortestAngle(phi_x_r, roll);  
  
//throttle command  
*delta_t = clip(0.0, kp_throttle * pitch_d + delta_t_d, 1.0);  
  
//elevator command  
*delta_e = clip(-1.0, kp_pitch * Delta_theta + k_theta_psi * abs_yaw, 1.0);  
  
//aileron command  
*delta_a = clip(-1.0, kp_roll * Delta_phi - kd_roll * gyro_x, 1.0);  
  
//rudder command  
*delta_r = clip(-1.0, kp_rudder * Delta_yaw, 1.0);  
}
```

21.5 Arduino Code for the Autopilot

```
1 //Include necessary libraries
2 #include <function_FeedbackControlOfDelta_tear.h>      //Elevator, Aileron, Rudder, Throttle commands
3 #include <function_GPSwaypointAlgorithm.h>                //Displacements from GPS waypoints
4 #include <function_Quaternion2Euler.h>                    //Convert quaternion to Euler orientation
5 #include <function_QuaternionOrientationEstimator.h>     //Sensor fusion to estimate quaternions
6 #include <SPI.h>
7 #include <SD.h>
8 #include <ICM20948_WE.h>        //ICM20948_WE.h: Gyro, Accelerometer, Magnetometer
9 #include <BMP280_DEV.h>        //BMP280_DEV: altitude, pressure, and temperature
10 #include <SoftwareSerial.h>       //SoftwareSerial: GPS communication
11 #include <TinyGPS.h>           //TinyGPS: GPS Latitude, Longitude, Altitude
12 #include <Servo.h>             //Servo: Servo motors and ESC
13
14
15 //name the file for the SD card
16 String filename = "FlightData001.csv";
17 //Say whether to use GPS data or not
18 bool useGPS = true;
19
20 //#####GPS Waypoint Displacements From Bottom-Bump Location#####
21 const int Nwaypoints = 7;
22 double targetLatVec[Nwaypoints] = { -0.0006, 0.0006, -0.0006, 0.0006, -0.0006, 0.0006, 0.0 };
23 double targetLonVec[Nwaypoints] = { 0.0006, 0.0006, -0.0006, -0.0006, 0.0006, 0.0006, 0.0 };
24 double targetAltVec[Nwaypoints] = { 20.0, 20.0, 20.0, 20.0, 10.0, 5.0, 1.0 };
25 //Flyaway boundaries (displacements from belly-bump position)
26 const double LongitudeWest = -0.0016; //deg
27 const double LongitudeEast = 0.0016; //deg
28 const double LatitudeNorth = 0.0016; //deg
29 const double LatitudeSouth = -0.0016; //deg
30 const double AltitudeMin = -60.0; //m
31 const double AltitudeMax = 60.0; //m
32 #####End of GPS Waypoints#####
33
34 #####Variables for GPS#####
35 TinyGPS gps;
36 const int rxPin = 1;
37 const int txPin = 0;
38 SoftwareSerial ss(rxPin, txPin);
39 static void readGPS(unsigned long ms);
```

```
40 float GPSlat = 0.0; //declare GPS latitude and longitude variables
41 float GPSlon = 0.0;
42 float GPSspeed = 0.0; //declare GPS speed (m/s)
43 float GPSAlt = 0.0; //Declare GPS altitude (m)
44 int waypointIndex = 0;
45 double target_lat = targetLatVec[0];
46 double target_lon = targetLonVec[0];
47 double target_alt = targetAltVec[0];
48 bool GPSisValid = false;
49 //#####End of Variables for GPS#####
50
51 //#####Variables for SD Card Reader#####
52 #if !defined(ARDUINO_ARCH_RP2040)
53 #error For RP2040 only
54 #endif
55 #if defined(ARDUINO_ARCH_MBED)
56 #define PIN_SD_MOSI PIN_SPI_MOSI
57 #define PIN_SD_MISO PIN_SPI_MISO
58 #define PIN_SD_SCK PIN_SPI_SCK
59 #define PIN_SD_SS PIN_SPI_SS
60 #else
61 #define PIN_SD_MOSI PIN_SPI0_MOSI
62 #define PIN_SD_MISO PIN_SPI0_MISO
63 #define PIN_SD_SCK PIN_SPI0_SCK
64 #define PIN_SD_SS PIN_SPI0_SS
65 #endif
66 #define _RP2040_SD_LOGLEVEL_ 0
67 //#####End of Variables for SD Card Reader#####
68
69 //#####Variables for 10 DOF ICM 20948 Sensor#####
70 #define ICM20948_ADDR 0x68 //define the address for the ICM 20948
71 //Create the IMU object and name it myIMU
72 ICM20948_WE myIMU = ICM20948_WE(ICM20948_ADDR);
73 //Calibration variables for the magnetometer
74 double magCalx = 0.0;
75 double magCaly = 0.0;
76 double magCalz = 0.0;
77 double magXmax[3] = { 0.0 };
78 double magYmax[3] = { 0.0 };
79 double magZmax[3] = { 0.0 };
80 double magXmin[3] = { 0.0 };
81 double magYmin[3] = { 0.0 };
82 double magZmin[3] = { 0.0 };
83
84 //Create the BMP280_DEV object, name it bmp280. I2C address is 0x77
85 BMP280_DEV bmp280(Wire1); //BMP communication on the secondary I2C port
86 //declare temperature, pressure, and altitude
87 float temperature, pressure, height_P;
88 //#####End of Variables for 10 DOF MPU9250 Sensor#####
```

```
89
90 //#####Variables for the Servo Motors#####
91 Servo LeftElevon;
92 Servo RightElevon;
93 const int LeftElevonPin = 8;      //Connect the left elevon to this pin
94 const int RightElevonPin = 13;    //Connect the right elevon to this pin
95 //#####End of Variables for the Servo Motors#####
96
97 //#####Variables for the Brushless Motor#####
98 Servo props;
99 const int propsPin = 9;           //Connect the ESC to this pin
100 const int ESC_MIN = 90;          //Minimum Servo Position Value for bi- ↵
101   directional ESC
102 const int ESC_MAX = 0;           //Maximum Servo position value for bi- ↵
103   directional ESC
104 const int ESC_REVERSE_MAX = 180; //Maximum servo position for reverse ↵
105   direction
106 //#####End of Variables for the Brushless Motor#####
107
108 //#####Variables for Altitude Complementary Filter#####
109 bool hpIsValid = false;          //indicate if the pressure altitude signal ↵
110   is valid
111 bool hGPSisValid = false;        //indicate if the GPS altitude signal is ↵
112   valid
113 int AltitudesValid = 0;          //0=not valid, 1=first time valid, 2 or ↵
114   more=valid for a long time
115 double hP;                      //low-pass filtered pressure altitude
116 double hGPS;                    //Low-pass filtered GPS altitude
117 const double tauAlt = 100.0;     //(s) complementary filter time-constant
118 double altitude = 0.0;           //(m) complementary filter prediction of ↵
119   altitude
120 //#####End of Variables for Altitude Complementary Filter#####
121
122 //#####Feedback control gains for delta_t,e,a,r#####
123 const double delta_t_d = 0.8;    ///(0-1) desired throttle ↵
124   at cruise speed
125 const double kp_throttle = 1.7;  //(-) k_{p,t}
126   proportional throttle gain
127 const double kp_roll = 1.5;      //(-) k_{p,phi}
128   proportional roll gain
129 const double kd_roll = 0.0;      //(-) k_{d,phi}
130   derivative roll gain
131 const double kp_pitch = -0.8;   //(-) k_{p,theta}
132   proportional pitch gain
133 const double kp_yaw = 1.0;       //(-) k_{p,psi}
134   proportional yaw gain
135 const double kp_rudder = 0.0;    //(-) k_{psi}
136   proportional rudder yaw gain
137 const double k_theta_psi = -0.5; //(-) k_{theta,psi}
```

```
    proportional yaw-pitch
124 const double max_roll = 50.0 * 3.14159 / 180.0; // [limit](rad) max roll ↵
    angle limit
125 const double max_pitch = 20.0 * 3.14159 / 180.0; // [limit](rad) max pitch ↵
    angle limit
126 // ##### End of Feedback control gains for ↵
    delta_t,e,a,r#####//#
127
128 // ##### Variables for Take-off#####
129 double takeOffLat = 0.0;
130 double takeOffLon = 0.0;
131 double takeOffAlt = 0.0;
132 // ##### End of Take-off Variables#####
133
134 // ##### Variables for the quaternion orientation ↵
    estimator#####
135 double quaternion[4] = { 1.0, 0.0, 0.0, 0.0 }; // (-) initial quaternion ↵
    orientation
136 double MI[3] = { 17.4, 0.0, 47.9 }; // (uT) local geomagnetic ↵
    vector
137 const double beta = 0.3; // (-) tuning parameter for ↵
    the Kok Schon correction
138 const double tau_m = 900.0; // (s) low pass filter time ↵
    constant for the geomagnetic vector
139 // ##### End of Variables for the quaternion orientation ↵
    estimator#####
140
141 // ##### State Machine Variables#####
142 bool nosebump = false; // Nose bumps indicate landing...turn off ↵
    props
143 const int calibration = 0; // Calibration state
144 const int preflight = 1; // Pre-flight state indicator
145 const int takeOff = 2; // Take-off state indicator
146 const int FlightMode = 33; // normal flying state indicator
147 const int landing = 4; // Landing state indicator
148 int FlightState = calibration; // Begin in calibration state
149 bool landCMD = false; // All waypoints hit, change to landing ↵
    mode
150 bool signalsAreValid = false; // Indicate if all signals are valid
151 // ##### End of State Machine Variables#####
152
153
154 // Additional variables
155 bool USBconnected = true;
156 double t_last = 0.0;
157
158 void setup() {
159     // ##### Set up the Servos#####
160     LeftElevon.attach(LeftElevonPin);
```

```
161 LeftElevon.write(90);
162 RightElevon.attach(RightElevonPin);
163 RightElevon.write(90);
164
165 //#####Set up the propellers#####
166 props.attach(propsPin, 1000, 2000);
167 props.write(ESC_MAX);
168 delay(2000);
169 props.write(ESC_REVERSE_MAX);
170 delay(2000);
171 props.write(ESC_MIN);
172 delay(2000);
173
174 delay(3000);
175
176 // Start serial communication
177 Serial.begin(9600);
178 delay(3000);
179 if (!Serial) {
180     USBconnected = false;
181 }
182 if (useGPS) {
183     //Start communication with the GPS module
184     ss.begin(9600);
185     unsigned long GPSdelay = 1000; //Time delay in ms
186     readGPS(GPSdelay);
187 }
188 //Begin I2C on the primary port
189 Wire.setSCL(5); //set primary I2C SCL to Pin GPIO5
190 Wire.setSDA(4); //set primary I2C SDA to Pin GPIO4
191 Wire.begin();
192 //Begin I2C on the secondary port
193 Wire1.setSCL(11); //set secondary I2C SCL to Pin GPIO11
194 Wire1.setSDA(10); //set secondary I2C SDA to Pin GPIO10
195 Wire1.begin(); //Begin I2C communication on the secondary I2C port
196 delay(2000);
197 if (!myIMU.init()) { //Start the MPU, if it fails, report an error
198     if (USBconnected) {
199         Serial.println("ICM20948 does not respond");
200     }
201 }
202 if (!myIMU.initMagnetometer()) { //Start the magnetometer, if it fails, ↵
203     if (USBconnected) {
204         Serial.println("Magnetometer does not respond");
205     }
206 }
207 delay(1000);
208 bmp280.begin(); // Default initialization, place the BMP280 into ↵
```

```
    SLEEP_MODE
209 //Set some BMP280 parameters
210 //bmp280.setPresOversampling(OVERSAMPLING_X4); // Set the pressure      ↵
211 //oversampling to X4
212 //bmp280.setTempOversampling(OVERSAMPLING_X1); // Set the temperature   ↵
213 //oversampling to X1
214 //bmp280.setIIRFilter(IIR_FILTER_4); // Set the IIR filter to setting 4
215 bmp280.setTimeStandby(TIME_STANDBY_2000MS); // Set the standby time to   ↵
216 // 2 seconds
217 // Start BMP280 continuous      ↵
218 //conversion in NORMAL_MODE
219
220 //Set some ICM-20948 filter and sampling parameters
221 myIMU.setAccOffsets(-16000.0, 16800.0, -16300.0, 16500.0, -16500.0,      ↵
222 // 16650.0); //Accelerometer offsets
223 myIMU.setAccDLPF(ICM20948_DLPF_5); //Digital low pass filter with a 10      ↵
224 // Hz bandwidth
225 myIMU.setGyrDLPF(ICM20948_DLPF_2); //Digital low pass filter with a      ↵
226 // 119.5 Hz Bandwidth
227 myIMU.setMagOpMode(AK09916_CONT_MODE_20HZ); //Continuous measurements,      ↵
228 // 20 Hz rate
229
230 delay(5000);
231 if (USBconnected) {
232     Serial.println("Done!");
233 }
234
235 #if defined(ARDUINO_ARCH_MBED)
236     if (USBconnected) {
237         Serial.print("Starting SD Card ReadWrite on MBED ");
238     }
239 #else
240     if (USBconnected) {
241         Serial.print("Starting SD Card ReadWrite on ");
242     }
243 #endif
244
245     if (USBconnected) {
246         Serial.println(BOARD_NAME);
247         Serial.print("Initializing SD card with SS = ");
248         Serial.println(PIN_SD_SS);
249         Serial.print("SCK = ");
250         Serial.println(PIN_SD_SCK);
251         Serial.print("MOSI = ");
252         Serial.println(PIN_SD_MOSI);
253         Serial.print("MISO = ");
254         Serial.println(PIN_SD_MISO);
255     }
256 }
```

```
249  if (!SD.begin(PIN_SD_SS)) {
250      if (USBconnected) {
251          Serial.println("Initialization failed!");
252      }
253  } else {
254      for (int ii = 0; ii < 1000; ii++) {
255          if (SD.exists(filename)) {
256              filename = "FlightData";
257              filename += String(ii);
258              filename += ".csv";
259          }
260      }
261  }
262  if (USBconnected) {
263      Serial.println("Initialization done.");
264  }
265
266 //create the SD card file and open it for writing
267 File dataFile = SD.open(filename, FILE_WRITE);
268
269 //If it opened correctly, write the header to it
270 if (dataFile) {
271     String headerString = "time,";
272     headerString +=
273         "gFx,gFy,gFz,wx,wy,wz,altitude,Bx,By,Bz,Azimuth,Pitch,Roll,Latitude,Longitude,GPSSAlt,delta_t,delta_e,delta_a,GPSSpeed";
274     dataFile.println(headerString);
275     dataFile.close();
276 } else {
277     if (USBconnected) {
278         Serial.println("Initialization failed to open the file");
279     }
280
281 t_last = 0.0;
282 }
283
284 void loop() {
285     myIMU.readSensor();
286     //Get the values from the accelerometer
287     xyzFloat accel = myIMU.getGValues();
288     //Get the values from the Gyrometer
289     xyzFloat gyro = myIMU.getGyrValues();
290     //Get the values from the Magnetometer
291     xyzFloat Mag = myIMU.getMagValues();
292
293     //put the data into arrays
294     double accelerometer[3];
295     double gyrometer[3];
```

```
296     double gravity[3];
297     double magnetometer[3];
298     gyrometer[0] = gyro.x * 3.14159 / 180.0;    // [rad/s]
299     gyrometer[1] = -gyro.y * 3.14159 / 180.0;   // [rad/s]
300     gyrometer[2] = -gyro.z * 3.14159 / 180.0;   // [rad/s]
301
302     accelerometer[0] = -accel.x;    // [g]
303     accelerometer[1] = accel.y;    // [g]
304     accelerometer[2] = accel.z;    // [g]
305
306     if (USBconnected) {
307         Serial.println(accelerometer[2], 4);
308     }
309
310     magnetometer[0] = (Mag.x - magCalx); // [uT]
311     magnetometer[1] = (Mag.y - magCaly); // [uT]
312     magnetometer[2] = (Mag.z - magCalz); // [uT]
313
314     // Get the time
315     double t = (double)millis() / 1000.0;
316     double dt = t - t_last;
317     t_last = t;
318
319     // Get the bmp280 measurements
320     // Temperature in Celsius, pressure in hectopascals, altitude in meters
321     bmp280.getMeasurements(temperature, pressure, height_P);
322     // Check if the pressure altitude is valid
323     if (height_P > 500.0f) {
324         hpIsValid = true;
325     } else {
326         hpIsValid = false;
327     }
328
329     double GPSspeed = 0.0; // (m/s)
330     // Get GPS measurements
331     unsigned long GPSdelay = 1; // Time delay in ms
332     readGPS(GPSdelay);
333     unsigned long age;
334     // Read the latitude and longitude
335     gps.f_get_position(&GPSlat, &GPSlon, &age);
336     // Read the GPS altitude (m)
337     GPSAlt = gps.f_altitude();
338     // Get the GPS speed (m/s)
339     GPSspeed = (double)gps.f_speed_mps(); // (m/s)
340
341     // Check that the GPS signal is valid
342     if (GPSlat < (LatitudeNorth + takeOffLat)
343         && GPSlat > (LatitudeSouth + takeOffLat)
344         && GPSlon > (LongitudeWest + takeOffLon)
```

```
345     && GPSlon < (LongitudeEast + takeOffLon)) {  
346         GPSisValid = true;  
347     } else {  
348         GPSisValid = false;  
349     }  
350  
351     //check that the GPS altitude is valid  
352     if (GPSAlt > AltitudeMin + takeOffAlt && GPSAlt < AltitudeMax +  
            takeOffAlt) {  
353         hGPSisValid = true;  
354     }  
355  
356     //Approximate gravity as the accelerometer signal  
357     gravity[0] = accelerometer[0];  
358     gravity[1] = accelerometer[1];  
359     gravity[2] = accelerometer[2];  
360  
361     if (hpIsValid && hGPSisValid && AltitudesValid < 3) {  
362         AltitudesValid = AltitudesValid + 1;  
363     }  
364  
365     if (AltitudesValid == 1) {  
366         hP = (double)height_P; //low-pass filtered pressure altitude  
367         hGPS = GPSAlt;           //Low-pass filtered GPS altitude  
368     }  
369  
370     //set the altitude  
371     altitude = (double)GPSAlt;  
372  
373     //Check if all waypoint signals are valid  
374     if (GPSisValid && (altitude > (AltitudeMin + takeOffAlt)) && (altitude < ?  
            (AltitudeMax + takeOffAlt))) {  
375         //Signals are valid and within the flight envelope  
376         signalsAreValid = true;  
377     } else {  
378         signalsAreValid = false;  
379     }  
380  
381     //Get the quaternion orientation  
382     function_QuaternionOrientationEstimator(  
383         quaternion, MI, quaternion, MI, gyrometer, gravity,  
384         magnetometer, dt, beta, tau_m);  
385  
386     //Get the Euler orientation  
387     double roll, pitch, yaw;  
388     function_Quaternion2Euler(&roll, &pitch, &yaw, quaternion);  
389  
390     //Check state-machine conditions  
391     if (FlightState == calibration) {
```

```
392     //Calibrate the magnetometers
393     bool bottombump = false;
394     if (gravity[2] > 1.8) {
395         bottombump = true;
396     }
397     if (bottombump
398         && GPSlat < 200.0
399         && GPSlon < 200.0
400         && altitude < 99000.0) {
401         //capture the current GPS location
402         takeOffLat = (double)GPSlat;
403         takeOffLon = (double)GPSlon;
404         takeOffAlt = altitude;
405
406         //Calibration is finished...move to a preflight state
407         FlightState = preflight;
408         //Set the magnetometer offsets
409         magCalx = (median(magXmax) + median(magXmin)) / 2.0;
410         magCaly = (median(magYmax) + median(magYmin)) / 2.0;
411         magCalz = (median(magZmax) + median(magZmin)) / 2.0;
412     }
413 } else if (FlightState == preflight) {
414     //Pre-flight condition
415     bool tailbump = false;
416     if (gravity[0] < -1.8) {
417         //The tail has been bumped indicating turn on propellers
418         tailbump = true;
419     }
420     if (tailbump && signalsAreValid) {
421         //Transition to take-off mode and turn on propellers
422         FlightState = takeOff;
423     }
424 } else if (FlightState == takeOff) {
425     //exit the take-off state when the airplane has flown
426     // a specified distance from the take-off GPS location
427     double dXI_TO[3];
428     function_GPSwaypointAlgorithm(dXI_TO, GPSlat, GPSlon, altitude,
429                                     takeOffLat, takeOffLon, altitude);
430
431     //get the distance from the initial take-off point
432     double dist_TO = sqrt(dXI_TO[0] * dXI_TO[0] + dXI_TO[1] * dXI_TO[1]);
433
434     if (dist_TO > 30.0) {
435         //transition to flying mode
436         FlightState = FlightMode;
437     }
438     if (gravity[0] > 1.8) {
439         //A Landing has been detected, turn off propellers
440         nosebump = true;
```

```
441     }
442     if (nosebump) {
443         //Turn off propellers for good
444         FlightState = landing;
445     }
446 } else if (FlightState == FlightMode) {
447     //Normal flying condition
448     if (gravity[0] > 1.8) {
449         //Landing has been detected, turn off propellers
450         nosebump = true;
451     }
452     if (nosebump || waypointIndex > Nwaypoints - 2) {
453         //Turn off propellers for good
454         FlightState = landing;
455     }
456 } else {
457     //landing condition
458 }
459
460 //get the control commands
461 double delta_T = 0.0;
462 double delta_e = 0.0;
463 double delta_a = 0.0;
464 double delta_r = 0.0;
465 if (FlightState == calibration) {
466     //Find the max and min magnetometer values
467     //x-axis
468     if (magnetometer[0] != 0.0) {
469         //compare with magXmax
470         if (magXmax[0] == 0.0 || magnetometer[0] > magXmax[0]) {
471             magXmax[0] = magnetometer[0];
472         } else if (magXmax[1] == 0.0 || magnetometer[0] > magXmax[1]) {
473             magXmax[1] = magnetometer[0];
474         } else if (magXmax[2] == 0.0 || magnetometer[0] > magXmax[2]) {
475             magXmax[2] = magnetometer[0];
476         } else { /*do nothing*/
477         }
478         //compare with magXmin
479         if (magXmin[0] == 0.0 || magnetometer[0] < magXmin[0]) {
480             magXmin[0] = magnetometer[0];
481         } else if (magXmin[1] == 0.0 || magnetometer[0] < magXmin[1]) {
482             magXmin[1] = magnetometer[0];
483         } else if (magXmin[2] == 0.0 || magnetometer[0] < magXmin[2]) {
484             magXmin[2] = magnetometer[0];
485         } else { /*do nothing*/
486         }
487     }
488     //y-axis
489     if (magnetometer[1] != 0.0) {
```

```
490     //compare with magYmax
491     if (magYmax[0] == 0.0 || magnetometer[1] > magYmax[0]) {
492         magYmax[0] = magnetometer[1];
493     } else if (magYmax[1] == 0.0 || magnetometer[1] > magYmax[1]) {
494         magYmax[1] = magnetometer[1];
495     } else if (magYmax[2] == 0.0 || magnetometer[1] > magYmax[2]) {
496         magYmax[2] = magnetometer[1];
497     } else { /*do nothing*/
498     }
499     //compare with magYmin
500     if (magYmin[0] == 0.0 || magnetometer[1] < magYmin[0]) {
501         magYmin[0] = magnetometer[1];
502     } else if (magYmin[1] == 0.0 || magnetometer[1] < magYmin[1]) {
503         magYmin[1] = magnetometer[1];
504     } else if (magYmin[2] == 0.0 || magnetometer[1] < magYmin[2]) {
505         magYmin[2] = magnetometer[1];
506     } else { /*do nothing*/
507     }
508 }
509 //z-axis
510 if (magnetometer[2] != 0.0) {
511     //compare with magZmax
512     if (magZmax[0] == 0.0 || magnetometer[2] > magZmax[0]) {
513         magZmax[0] = magnetometer[2];
514     } else if (magZmax[1] == 0.0 || magnetometer[2] > magZmax[1]) {
515         magZmax[1] = magnetometer[2];
516     } else if (magZmax[2] == 0.0 || magnetometer[2] > magZmax[2]) {
517         magZmax[2] = magnetometer[2];
518     } else { /*do nothing*/
519     }
520     //compare with magZmin
521     if (magZmin[0] == 0.0 || magnetometer[2] < magZmin[0]) {
522         magZmin[0] = magnetometer[2];
523     } else if (magZmin[1] == 0.0 || magnetometer[2] < magZmin[1]) {
524         magZmin[1] = magnetometer[2];
525     } else if (magZmin[2] == 0.0 || magnetometer[2] < magZmin[2]) {
526         magZmin[2] = magnetometer[2];
527     } else { /*do nothing*/
528     }
529 }
530 } else if (signalsAreValid && FlightState == takeOff) {
531     //get commands
532     delta_T = 1.0;                                //Full throttle
533     double takeOffPitch = 17.0 * 3.14159 / 180.0; //Calibration take-off ⇒
534     pitch
535     delta_e = max(-1.0, min(kp_pitch * (takeOffPitch - pitch), 1.0));
536     delta_a = max(-1.0, min(-kp_roll * roll, 1.0)); //Just maintain level ⇒
537     flight
538     delta_r = 0.0;
```

```
537     //Write propellor commands
538     int propsCmd = mapPropsCmd(delta_T);
539     writePropCmd(propsCmd);
540
541 } else if (signalsAreValid && FlightState == FlightMode) {
542     double dXI[3];
543     //get the displacement from the airplane to the target
544     target_lat = targetLatVec[waypointIndex] + takeOffLat;
545     target_lon = targetLonVec[waypointIndex] + takeOffLon;
546     target_alt = targetAltVec[waypointIndex] + takeOffAlt;
547
548     function_GPSwaypointAlgorithm(dXI, GPSlat, GPSlon, altitude,
549                                   target_lat, target_lon, target_alt);
550
551     //get the distance to the waypoint
552     double distance = sqrt(dXI[0] * dXI[0] + dXI[1] * dXI[1]);
553
554     //Get the feedback control commands
555     function_FeedbackControlOfDelta_tear(
556         &delta_T,           // [0,1] throttle command
557         &delta_e,          // [-1,0] elevator command
558         &delta_a,          // [-1,1] aileron command
559         &delta_r,          // [-1,1] rudder command
560         dXI,               // (m) x,y, and z displacements from the target      ↵
561         waypoint
562         gyrometer[0],    // (rad/s) gyro measurement of x-axis angular velocity
563         roll,             // (rad) phi_x actual roll euler angle
564         pitch,            // (rad) theta_y actual pitch euler angle
565         yaw,              // (rad) psi_z actual yaw euler angle
566         delta_t_d,        // (0-1) desired throttle at cruise speed
567         kp_throttle,      // (-) k_{p,t} proportional throttle gain
568         kp_roll,          // (-) k_{p,phi} proportional roll gain
569         kd_roll,          // (-) k_{d,phi} derivative roll gain
570         kp_pitch,         // (-) k_{p,theta} proportional pitch gain
571         kp_yaw,           // (-) k_{p,psi} proportional yaw gain
572         kp_rudder,        // (-) k_{psi} proportional rudder yaw gain
573         k_theta_psi,       // (-) k_{theta,psi} proportional yaw-pitch
574         max_roll,          // [limit](rad) max roll angle limit
575         max_pitch,         // [limit](rad) max pitch angle limit
576     );
577
578     //Write propellor commands
579     int propsCmd = mapPropsCmd(delta_T);
580     writePropCmd(propsCmd);
581
582     //Get the waypoint index
583     if (distance < 40.0) {
584         waypointIndex += 1;
585     }
```

```
585 } else if (signalsAreValid && FlightState == landing) {
586     double dXI[3];
587     //Return to the take-off point
588     function_GPSwaypointAlgorithm(dXI, GPSlat, GPSlon, altitude,
589                                     takeOffLat, takeOffLon, takeOffAlt);
590
591     //get the distance to the waypoint
592     double distance = sqrt(dXI[0] * dXI[0] + dXI[1] * dXI[1]);
593
594     //Get the feedback control commands
595     function_FeedbackControlOfDelta_tear(
596         &delta_T,          // [0,1] throttle command
597         &delta_e,          // [-1,0] elevator command
598         &delta_a,          // [-1,1] aileron command
599         &delta_r,          // [-1,1] rudder command
600         dXI,              // (m) x,y, and z displacements from the target      ↵
601         waypoint
602         gyrometer[0],    // (rad/s) gyro measurement of x-axis angular velocity
603         roll,            // (rad) phi_x actual roll euler angle
604         pitch,           // (rad) theta_y actual pitch euler angle
605         yaw,             // (rad) psi_z actual yaw euler angle
606         delta_t_d,       // (0-1) desired throttle at cruise speed
607         kp_throttle,     // (-) k_{p,t} proportional throttle gain
608         kp_roll,         // (-) k_{p,phi} proportional roll gain
609         kd_roll,         // (-) k_{d,phi} derivative roll gain
610         kp_pitch,        // (-) k_{p,theta} proportional pitch gain
611         kp_yaw,          // (-) k_{p,psi} proportional yaw gain
612         kp_rudder,       // (-) k_{psi} proportional rudder yaw gain
613         k_theta_psi,     // (-) k_{theta,psi} proportional yaw-pitch
614         max_roll,        // [limit](rad) max roll angle limit
615         max_pitch        // [limit](rad) max pitch angle limit
616     );
617
618     //Turn off propellers
619     writePropCmd(0);
620
621 } else {
622     //Get the feedback control commands
623     delta_T = 0.0;
624     double LandingPitch = 8.0 * 3.14159 / 180.0; //Calibration take-off      ↵
625     pitch
626     delta_e = max(-1.0, min(kp_pitch * (LandingPitch - pitch), 1.0));
627     delta_a = max(-1.0, min(kp_roll * (-0.09 - roll), 1.0)); //Just      ↵
628     maintain level flight
629     delta_r = 0.0;
630
631     //Turn off propellers
632     writePropCmd(0);
633 }
```

```
631
632     //Write elevon commands (CHANGE THESE PER CALIBRATION DESIRED)
633     //Adjust commands for elevons
634     double LeftServoCMD = delta_e + delta_a;
635     double RightServoCMD = -delta_e + delta_a;
636     LeftElevon.write(mapServoCmd(LeftServoCMD, -10.0, 65.0));    // delta,    ↵
637     RightElevon.write(mapServoCmd(RightServoCMD, 10.0, 65.0));   // delta,    ↵
638     trim_degrees, travel_degrees
639
640     //Get the data string that will be written to the SD card
641     String dataString = "";
642     dataString += t; //time
643     dataString += ",";
644     dataString += gravity[0]; //gFx
645     dataString += ",";
646     dataString += gravity[1]; //gFy
647     dataString += ",";
648     dataString += gravity[2]; //gFz
649     dataString += ",";
650     dataString += gyrometer[0]; //wx
651     dataString += ",";
652     dataString += gyrometer[1]; //wy
653     dataString += ",";
654     dataString += gyrometer[2]; //wz
655     dataString += ",";
656     dataString += height_P; //altitude in meters
657     dataString += ",";
658     dataString += magnetometer[0]; //Bx
659     dataString += ",";
660     dataString += magnetometer[1]; //By
661     dataString += ",";
662     dataString += magnetometer[2]; //Bz
663     dataString += ",";
664     dataString += yaw; //Azimuth
665     dataString += ",";
666     dataString += pitch; //Pitch
667     dataString += ",";
668     dataString += roll; //Roll
669     dataString += ",";
670     if (useGPS) {
671         char charArray[12];
672         dtostrf(GPSlat, 1, 6, charArray); //Latitude
673         dataString += charArray;
674         dataString += ",";
675         dtostrf(GPSlon, 1, 6, charArray); //Longitude
676         dataString += charArray;
677         dataString += ",";
678         dataString += GPSAlt; //GPS altitude (m)
```

```
678 } else {
679     dataString += "0"; //Latitude
680     dataString += ",";
681     dataString += "0"; //Longitude
682     dataString += ",";
683     dataString += "0"; //GPS altitude (m)
684 }
685 dataString += ",";
686 dataString += delta_T;
687 dataString += ",";
688 dataString += delta_e;
689 dataString += ",";
690 dataString += delta_a;
691 dataString += ",";
692 dataString += GPSspeed;
693
694 //write the data string to the SD card
695 writeToSDcard(dataString);
696 }
697
698 void writeToSDcard(String dataString) {
699     //create the SD card file and open it for writing
700     File dataFile = SD.open(filename, FILE_WRITE);
701
702     //If it opened correctly, write the data string to it
703     if (dataFile) {
704         dataFile.println(dataString);
705         dataFile.close();
706     } else {
707         if (USBconnected) {
708             Serial.println("Failed to open the file");
709         }
710     }
711 }
712
713 static void readGPS(unsigned long ms) {
714     unsigned long start = millis();
715     do {
716         while (ss.available())
717             gps.encode(ss.read());
718     } while (millis() - start < ms);
719 }
720
721 static int mapServoCmd(double delta, double trimDeg, double travel) {
722     //FOR CALIBRATION, CHANGE THIS EQUATION
723     double ServoCMD = max(90.0 - travel + trimDeg, min(90.0 + trimDeg +
724         travel * delta, 90.0 + travel + trimDeg));
725     ServoCMD = max(0.0, min(ServoCMD, 180.0));
726     return ((int)ServoCMD); ↵
```

```
726 }
727
728 static int mapPropsCmd(double delta) {
729     double pwm = 180.0 * delta;
730     return max(0, min((int)pwm, 180));
731 }
732
733 void writePropCmd(int cmd) {
734     int PWM = map(cmd, 0, 180, ESC_MIN, ESC_MAX);
735     props.write(PWM);
736 }
737
738 static double median(double x[3]) {
739     if ((x[1] <= x[0] && x[0] <= x[2]) || (x[2] <= x[0] && x[0] <= x[1])) {
740         //x[0] is between x[1] and x[2]
741         return (x[0]);
742     } else if ((x[0] <= x[1] && x[1] <= x[2]) || (x[2] <= x[1] && x[1] <= x[0])) {
743         //x[1] is between x[0] and x[2]
744         return (x[1]);
745     } else {
746         //x[2] is between x[0] and x[1]
747         return (x[2]);
748     }
749 }
750 }
```

