MATLAB FOR MECHANICAL ENGINEERING UNDERGRADUATES

An introductory guide to facilitate the use of MATLAB
as part of the Mechanical Engineering Curriculum for McGill University

Department of Mechanical Engineering
McGill University
817 Sherbrooke Street West
Montreal, Quebec, Canada
H3A 0C3

August 23, 2017

Preface

The purpose of this document is to expose Mechanical Engineering undergraduate students at McGill University the basics of MATLAB. When solving exercises in this book, it is suggested that the reader write out the code rather than copying and pasting the code. Learning is not a passive process, but rather an active one.

Please note that this text is a work in progress that is updated, corrected, and extended quite frequently. If you notice a typo (a spelling error, a mathematical error, anything), please notify Prof. James Richard Forbes at

james.richard.forbes@mcgill.ca

Acknowledgements

This document is written by undergraduate students for undergraduate students with guidance from Prof. James Richard Forbes. Its creation would not have been made possible without the support of McGill Associated of Mechanical Engineers (MAME) and the following individuals.

- Raphi Zonis, MAME VP academic who championed this project starting in 2015.
- Michael Geary, who wrote the initial draft in Fall 2015.
- Dylan Caverly, who undertook a major update in Fall 2016.

Contents

List of Figures

Chapter 1

Basics

1.1 Introduction

MATLAB is short for "matrix laboratory" [1]. This chapter will review basic MATLAB commands.

1.2 MATLAB Workspace

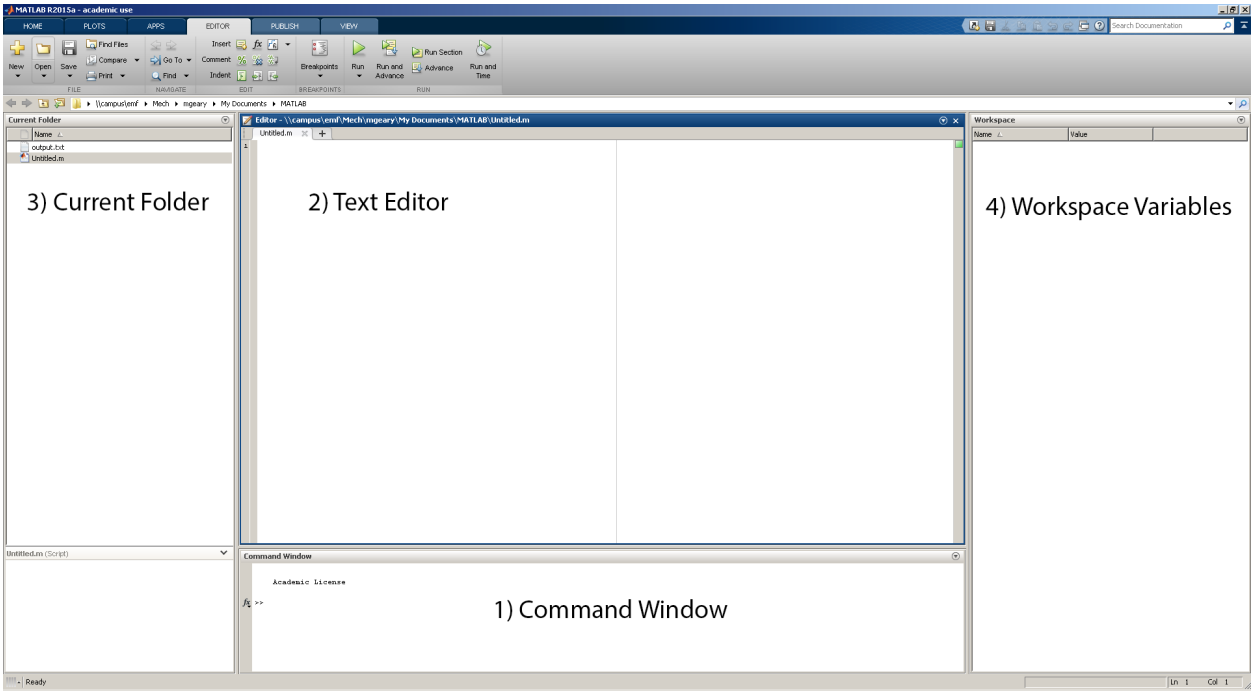


Figure 1.1: MATLAB Workspace

The MATLAB workspace is broken down into the following subsections as labeled in Figure 1.1.

1. Command Window
    - Used to enter commands to be executed, similar in operation to an operating system prompt.

[1] https://en.wikipedia.org/wiki/MATLAB

2. Text Editor
    - Used to write scripts and functions which can be saved for later execution.
3. Current Folder
    - Identifies the current working directory and sub-directories.
4. Workspace Variables
    - Variables created through execution of commands are stored here in the workspace.

1.3 Math Operations

1.3.1 Basic Math Operations

MATLAB may be used to perform basic math operations. These operations can be typed into the command window or executed from a script. For example, the simple operation

```
>> 4*(3^2 + 1)/5 - 1

ans =

     7
```

can be executed from the command window.

Remark 1.1. When the result of an operation is not assigned to a variable, the result will be assigned to the default variable name `ans`. Like all other variables defined in, or resulting from the command prompt, `ans` will be stored in the MATLAB workspace. Anytime a user would like to know what variables are in the workspace, simply type `whos` in the command window.

Since a user may want to access the value of an operation at a later time, the result can be stored as a variable with a specified name. For example,

```
>> y = 4*(3^2 + 1)/5 - 1

y =

     7
```

Remark 1.2. The variable names `i` and `j` should be avoided because `i` and `j` are reserved and designate imaginary numbers (i.e., $\sqrt{-1}$). In addition to `i` and `j`, other built-in MATLAB function names should not be used for variable assignment. For example, a user may be tempted to use `alpha`, `beta`, and `gamma` as variables names, but it turns out these are all built in MATLAB functions. Try typing in `help gamma` into the command prompt in the command window to find out what the function `gamma` goes.

Ending a command or variable assignment with a semicolon, `;`, will suppress the output. The variable will still be created for future use, but its value will not be output in the command window. This can be especially useful for long MATLAB scripts with many variables being stored. For example, consider

```
>> x = 2;
>> y = 3;
>> z = x + y

z =

     5
```

1.3.2 Built-In Math Operations

Many basic mathematical functions are available to be used in calculations. For example, $\sin(x)$ or e^x can be evaluated at $x = 3$ using MATLAB's built-in functions in the following way:

```
>> x = 3;
>> sin(x)

ans =

     0.1411


>> exp(x)

ans =

   20.0855
```

More information about any built-in MATLAB function can be found by entering `help` or `doc` in the command window, followed by the function. For example, `help sin` will produce general information about the sine function, as well as hyperlinks to relevant sections on the MATLAB website.

```
>> help sin
 sin    Sine of argument in radians.
    sin(X) is the sine of the elements of X.

    See also asin, sind.

    Reference page for sin
    Other functions named sin
```

Typing `doc` followed by a function opens up the MATLAB documentation page in a new window. This will provide a more extensive summary of the function, and will often include examples of the selected function.

1.4 Matrices

MATLAB has significant functionality built-in for creation, manipulation, and operation on matrices. By default, single numerical values are treated as 1×1 matrices, thus single value variables are a subset of matrix variables. This means that matrices can be handled in a similar fashion to single value variables. This

section will deal with the creation and modification of matrices, as well as the matrix algebra and operations which can be performed using MATLAB.

1.4.1 Matrix Definition

Row matrices are defined using commas to separate columns and semicolon to separate rows. Let us begin by defining a row matrix x.

```
>> x = [1,2,3]

x =

     1     2     3
```

Row matrices can alternatively be defined using a space between values without need for a comma. The following command produces the same result as above.

```
>> x = [1 2 3]

x =

     1     2     3
```

To define a column matrix, semicolons are used as follows.

```
>> x = [4; 5; 6]

x =

     4
     5
     6
```

Using spaces and semicolons, an arbitrary 3×3 matrix can be defined.

```
  >> A = [1 2 3; 4 5 6; 7 8 9]

A =

     1     2     3
     4     5     6
     7     8     9
```

It is also possible to transpose a matrix in MATLAB using an apostrophe. A row matrix can be transposed to become a column matrix in the following way.

```
>> x = [1 2 3]

x =
```

```
     1     2     3

>> y = x'

y =

     1
     2
     3
```

Similarly, a matrix can be transposed using an apostrophe.

```
>> A = [1 2 3; 4 5 6; 7 8 9]

A =

     1     2     3
     4     5     6
     7     8     9

>> B = A'

B =

     1     4     7
     2     5     8
     3     6     9
```

Complex matrices may also be defined using the same methods as real matrices.

```
>>  A = [1  3+5i; 1+2i 0]

A =

   1.0000 + 0.0000i   3.0000 + 5.0000i
   1.0000 + 2.0000i   0.0000 + 0.0000i
```

Note that using the apostrophe command on a complex matrix will yield the complex-conjugate transpose by default. The apostrophe command is defined generally as the complex-conjugate transpose command, but the complex-conjugate aspect of this command only manifests itself when applied to complex matrices.

```
>>  A = [1 3+5i; 1+2i 0 ];
>>  A = A'

A =

   1.0000 + 0.0000i   1.0000 - 2.0000i
   3.0000 - 5.0000i   0.0000 + 0.0000i
```

If the transpose of a matrix without conjugation is desired, the apostrophe command `A'` is replaced by `A.'`.

```
>>  A = [1 3+5i; 1+2i 0 ];
>>  A = A.'

A =

   1.0000 + 0.0000i   1.0000 + 2.0000i
   3.0000 + 5.0000i   0.0000 + 0.0000i
```

Remark 1.3. Both `i` and `j` are reserved by `MATLAB` to represent imaginary numbers. Do not use `i` and `j` as variable names, nor as counters in loops. See Section 1.7.

1.4.2 Indexing

Question 1.1. How do we access specific values of the matrix **A**?

First let us define a matrix to be used as an example for indexing operations.

```
>> A = [3 2 5; 10 3 4; 4 7 9]

A =

     3     2     5
    10     3     4
     4     7     9
```

Indexing is used to access specific values, or ranges of values within a matrix. To access a value within an matrix, we can specify the row and column of the desired value. For instance, accessing the numeric value in `Row 1` and `Column 3` can be done as follows.

```
>> A(1,3)

ans =

     5
```

Using a colon in place of a row or column coordinate will return all the values in that dimension. For instance, accessing all of `Column 2` is done as follows.

```
>> A(:,2)

ans =

     2
     3
     7
```

We can also query a specified range of values using the colon operator. Requesting the values in `Column 3` from `Row 1` to `Row 2` can be done in the following way

```
>> A(1:2,3)

ans =

     5
     4
```

1.4.3 Matrix Math Operations

MATLAB is well suited to perform matrix-math operations. First, addition of two matrices will be demonstrated.

```
>> A = [1 3 7; 2 5 6; 0 8 1];
>> B = [1 2 1; 2 7 4; 3 1 3];

>> A + B

ans =

     2     5     8
     4    12    10
     3     9     4
```

Matrix multiplication can easily be executed by using the multiplication command as demonstrated in Section 1.3. By default, the result is that of a full matrix multiplication, not simply an element-by-element multiplication. (Element-by-element operations will be covered in Section 1.4.6.) For instance, to multiplying matrices A and B can be realized as follows.

```
>> A*B

ans =

    28    30    34
    30    45    40
    19    57    35
```

When performing operations on a matrix using a scalar value, it is not necessary to use any sort of modified syntax. Here, a matrix is defined and operated on using scalar values.

```
>> A = [1 3; 7 5]

A =

     1     3
     7     5
```

Addition of a scalar value to a matrix can be done in the following way.

```
>> A + 2
```

```
ans =

     3     5
     9     7
```

Multiplication of a scalar value and a matrix is realized in the following manner.

```
>> A*10

ans =

    10    30
    70    50
```

1.4.4 Solving a Linear System of Equations

Using MATLAB's matrix operation capabilities, linear system of equations problem of the form $\mathbf{Ax} = \mathbf{b}$ can be solved quite easily. The matrices **A**, **x**, and **b** can be real or complex. First, **x** will be found using

$$\mathbf{x} = \mathbf{A}^{-1}\mathbf{b}.$$

Once demonstrated, the MATLAB syntax form of the linear system of equations solution will be show. It should be noted that the inverse of **A** can be calculated by using the commands `inv(A)`, `A^-1`, or \, and all yield the same result, but \ is numerically more efficient.

To show how to solve for **x** given $\mathbf{Ax} = \mathbf{b}$, consider the following.

```
>> A = [2 4; 1 3];
>> b = [10; 6];

>> x = inv(A)*b

x =

     3
     1
```

The numerically more efficient way to solve for **x** given $\mathbf{Ax} = \mathbf{b}$ of the form $\mathbf{x} = \mathbf{A}\backslash\mathbf{b}$.

```
>> x = A\b

x =

     3
     1
```

There are many command matrix operations built into MATLAB. The inverse of **A** can be found directly in the following way.

```
>> inv(A)
```

```
ans =

    1.5000   -2.0000
   -0.5000    1.0000
```

The eigenvalues of **A** can be found as follows.

```
>> eig(A)

ans =

    0.4384
    4.5616
```

The rank of **A** can be found directly in a similar fashion, as shown next.

```
>> rank(A)

ans =

     2
```

These are just a few examples of commands which can be applied when working with matrices, MATLAB can perform many more useful operations of this type.

1.4.5 Example: Linear System of Equations (MECH 210: Statics)

Example 1.1. As shown in Figure 1.2, a pickup truck is loaded with a heavy payload for transportation. The following specifications define the truck and payload system. Given these parameters, find the resulting vertical loads on the front and rear axle by solving a system of linear equations. The x location of the truck center of mass and payload are defined as distance from the front axle.

Mass of Truck	m_{tr}	2200	kg
Mass of Payload	m_p	1300	kg
Center of Mass, Truck	x_{tr}	1.2	m
Center of Mass, Payload	x_p	3.8	m
Wheelbase	x_w	4.0	m
Gravitational Acceleration	g	9.81	$\frac{\text{m}}{\text{s}^2}$

Begin with the sum of all forces and the sum of all moments about a specific point equalling zero.

$$\sum_{i=0}^{k} f_i = 0, \tag{1.1}$$

$$\sum_{i=0}^{k} M_i = 0. \tag{1.2}$$

Let f_{front} represent the normal force applied at the front axle and f_{rear} correspond to the normal force applied at the rear axle. Then, using Equation (1.1),

$$\sum_{i=0}^{k} f_i = f_{front} + f_{rear} - gm_{tr} - gm_p = 0. \tag{1.3}$$

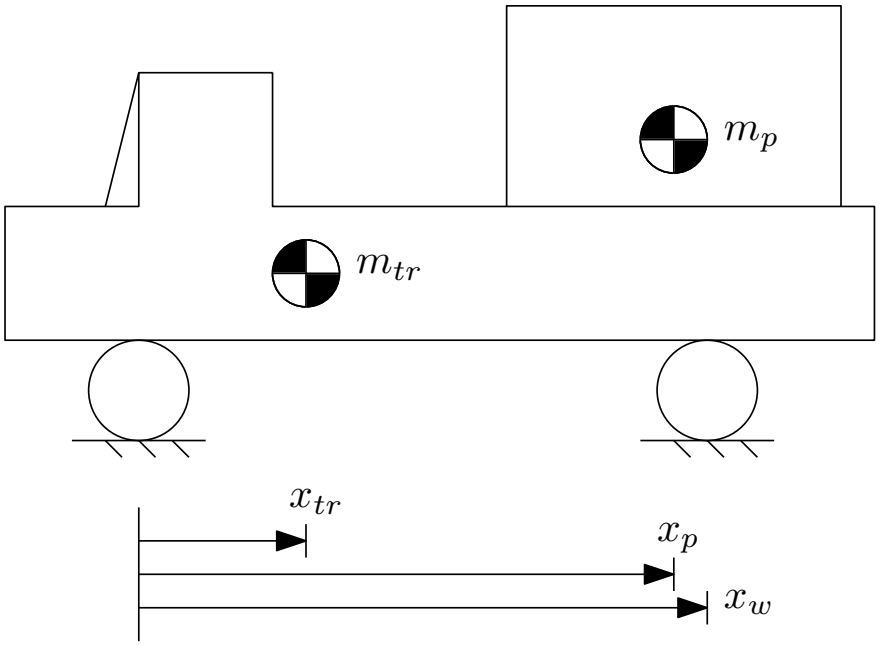


Figure 1.2: Pickup truck with heavy payload

From Equation (1.2), the sum of moments about the front axle yields

$$\sum_{i=0}^{k} M_i = (gm_{tr}x_{tr}) + (gm_p x_p) - (f_{rear}x_w) = 0. \tag{1.4}$$

Equations (1.3) and (1.4) can be written in the form $\mathbf{Ax} = \mathbf{b}$,

$$\mathbf{Ax} = \begin{bmatrix} 1 & 1 \\ 0 & x_w \end{bmatrix} \begin{bmatrix} f_{front} \\ f_{rear} \end{bmatrix} = \mathbf{b} = \begin{bmatrix} gm_{tr} + gm_p \\ gm_{tr}x_{tr} + gm_p x_p \end{bmatrix}. \tag{1.5}$$

In fact, all statics problems can be written in this form! At this point the problem can be set up in MATLAB. First, define the constant variables used.

```
m_tr = 2200;    %(kg) Mass of truck
m_p  = 1300;    %(kg) Mass of payload
x_tr = 1.2;     %(m) Location of truck center of mass
x_p  = 3.8;     %(m) Location of payload center of mass
x_w  = 4.0;     %(m) Wheelbase length
g    = 9.81;    %(m/s^2) Gravitational constant
```

Next, recreate the matrix representation shown in Equation 1.5.

```
A = [1 1; 0 x_w];
b = [(g*m_tr + g*m_p); (g*m_tr*x_tr + g*m_p*x_p)];
```

Finally, solve the system of linear equations using $\mathbf{x} = \mathbf{A}\backslash\mathbf{b}$.

```
>> x  = A\b;
```

```
x =

     15745
     18590
```

Therefore, the front axle load is $15,745$ N and the rear axle load is $18,590$ N.

1.4.6 Element-by-element Operations

Question 1.2. How do you square every element of a matrix?

It may also be desirable to perform element by element operations as opposed to full matrix algebra operations. We can use the standard mathematical operations shown in section 1.3, but can extend these to be performed on matrices of values rather than single values. To initiate an element by element operation, the standard operators ie. `* , / and ^` are replaced by `.* , ./ and .^`.

```
>> A = [1 3 2; 7 5 0; 8 2 4]

A =

     1     3     2
     7     5     0
     8     2     4

>> B = [2 5 6; 3 8 1; 0 9 4]

B =

     2     5     6
     3     8     1
     0     9     4
```

To perform an element-by-element multiplication where the resultant matrix will consist of $\mathbf{C}(\mathbf{x},\mathbf{y}) = \mathbf{A}(\mathbf{x},\mathbf{y})\mathbf{B}(\mathbf{x},\mathbf{y})$, consider the following.

```
>> A.*B

ans =

     2    15    12
    21    40     0
     0    18    16
```

To perform an element-wise division $\mathbf{C}(\mathbf{x},\mathbf{y}) = \mathbf{A}(\mathbf{x},\mathbf{y})/\mathbf{B}(\mathbf{x},\mathbf{y})$, consider the following.

```
>>  A./B

ans =

    0.5000    0.6000    0.3333
    2.3333    0.6250         0
       Inf    0.2222    1.0000
```

The use of `.^` when applied to matrices should be defined for clarification. Simply using command `^` along with the syntax $\mathbf{C} = \mathbf{A}^3$ will yield the result of $\mathbf{C} = \mathbf{AAA}$, a matrix multiplication. To achieve the result where each element of a matrix is raised to a scalar power, as defined by $\mathbf{C}(\mathbf{x}, \mathbf{y}) = \mathbf{A}(\mathbf{x}, \mathbf{y})^3$, we must implement the command `.^`.

```
>>  A^3

ans =

   272   244   116
   476   384   140
   604   436   236

>> A.^3

ans =

     1    27     8
   343   125     0
   512     8    64
```

The command `.^` can also be used to raise each element of a matrix to the power of the corresponding elements in matrix of the same size. $\mathbf{C}(\mathbf{x}, \mathbf{y}) = \mathbf{A}(\mathbf{x}, \mathbf{y})^{\mathbf{B}(\mathbf{x},\mathbf{y})}$

```
>> A.^B

ans =

           1         243          64
         343      390625           0
           1         512         256
```

1.5 Scripts and Functions

1.5.1 Scripts

Question 1.3. How do we create a script to execute a sequence of commands?

Many programs and calculations will require the execution of multiple lines of code, as opposed to single commands. To automate the process of executing commands, we can write these commands into script. We can create a script called newScript by

- typing `edit Untitled` into the command window, where `Untitled` can be replaced by a name of the users choosing, or
- clicking the *New Script* button in the command window.

Commands can be typed into the script and the script can be saved for use at a later time. When a script is run, the code contained in a script is executed from top to bottom, with all variables being stored in the `MATLAB` Workspace. For example, we can write the following into a script then save it to the working directory,

```
x  = 3;
y  = x^3;
z  = sqrt(x);
```

then the script can be executed either by

- typing the name of the script into the command window.
- clicking the *Run* button.
- using the shortcut `F5`.

Once the script has run, the values of `x`, `y` and `z` will be available in the Workspace.

1.5.2 Functions

Another file type, functions, can be used to execute lines of code in the similar way to the execution of scripts. The differentiating feature of functions is that functions are meant to perform calculations based on a set of specific inputs, and return only specific outputs. Functions execute commands, and the values of these commands are stored in the function's own space. Because a function has its on working environment, values in functions will only be returned to the main workspace if they are defined to be part of the output.

As an example of the syntax we will define a function with two input variables and three output variables. The first few lines of a function should be used as comments to make the function's purpose clear to the reader. To add a comment to a script or function, use the % command in front of the lines to be omitted from execution.

```
function [ out1, out2 ] = exampeFunction( in1, in2, in3 )
%exampleFunction Summary of this function goes here
%   Detailed explanation goes here

    % code goes here
end
```

One advantage of using functions is that calculations can be performed, and only the values of interest can be requested, thus keeping the workspace free of intermediate variables which we may not want to store.

```
function [ z ] = f( x, y )

   interVar1   = x^3;
   interVar2   = 47*y - 36;
   z = interVar1 - interVar2;
end
```

To execute the above function, it is called from the command window or a script.

```
>> z = f(10,2)

z =

   942
```

Remark 1.4. Functions are extremely useful when writing a program because they allow for modularity within the program's structure.

Storing calculations in functions allows them to be called in many places throughout a program. Functions also allow for easy modification of calculations later on in development stages of a program since each function acts as a storage space for a specific set of calculations. It should be noted that keeping functions short helps to make a program understandable to users. The purpose of a function should be immediately apparent, and a function should ideally serve an individual purpose.

1.6 Logical Operators

Logical operators can be used in MATLAB to identify when variables meet defined criteria. Logical comparisons will be demonstrated first on a single value, then applied to a matrix of values.

Question 1.4. Is the value contained in our variable **x** less than 5?

We will check out the properties of this variable **x** using comparisons to other values, and will receive logical variables as a result of these comparisons. In the following example, MATLAB will return a logical variable as an answer to our question with a value of either one (`true`) or zero (`false`).

```
>> x = 3
>> y = x < 5

y =

     0
```

When applied to a matrix of values, we will see that the question asked by a logical comparison, Question 1.4, is more generally described as:

Question 1.5. Which elements of this matrix are less than 5?

MATLAB will return a matrix of the same size populated with ones and zeros, where the ones will indicate which elements met the criteria and the zeros indicating those elements which did not.

```
>> x = [7 9 1 0 3 8];
>> y = x < 5

y =

     0     0     1     1     1     0
```

In addition to the "less than" comparison, we can use other comparisons to generate logical variables. These comparisons include `<, >, <=, >=, ==, ~=` (less than, greater than, less than or equal, greater than or equal, equal, not equal) and can also be used in the form `lt, gt, le, ge, eq, ne`.

1.6.1 Logical Indexing

Logical matrices can be applied to numerical matrices as an indexing element. Using a logical matrix as an indexing element extracts from the matrix being queried only the values which have a corresponding `1` in the logical matrix. The whole example is demonstrated as follows.

```
>> x = [7 9 1 0 6 2];
>> logicalLessThanFive = x < 5

logicalLessThanFive =

     0     0     1     1     0     1

>> valuesLessThanFive = x( logicalLessThanFive )

valuesLessThanFive =

     1     0     2
```

The above can be condensed by omitting the intermediate variables which were used for the sake of example.

```
>> x = [7 9 1 0 6 2];
>> x( x < 5 )

ans =

     1     0     2
```

A logical matrix generated based on **x** to index a separate matrix **y** can also be used as long as the matrices in question are of the same size.

```
>> x = [7  9  1  0  6  2];
>> y = [30 54 78 41 29 67];
>> y( x < 5 )

ans =

    78    41    67
```

1.7 Loops

1.7.1 For

A "for loop" is a loop that will execute a body of code a prescribed number of iterations before exiting at the 'end' statement.

In the following example, the body of code will be executed ten times, once for each value of **x** from 1 to 10. The default spacing when defining **x** = 1:10 is `1`.

```
for x = 1:10
    y = 3*x
end
```

A similar loop may be initialized with a non default spacing. A spacing of `2` for a different range of values is shown next.

```
for x = 0:2:20
    y = 3*x
end
```

A `for` loop can also be initialized to work through a predefined matrix of values. Here, a row matrix will be used.

```
for x = [4 2 3 7 5 4 9 6 6 10]
    y = 3*x
end
```

Note that in the above examples, the for loop is operating through a matrix of values, but in each instance is only performing a scalar operation, producing a scalar result which is not stored. In the following example, a for loop is using a predefined matrix, and the resulting values will be stored in a new matrix using the loop variable as an indexing element.

```
x = [4 2 3 7 5 4 9 6 6 10];
for n = 1:length(x)
    y(n) = 3*x(n)
end
```

As can be seen, there are many ways to initialize a for loop. The above examples all use the principles of a for loop with some differences in initialization depending on the desired result.

1.7.2 While

A "while loop" is similar to a for loop except instead of executing the body of code a defined number of times, the while loop will execute the body of code until a given condition is met.

```
x = 10;
while x < 30
   y = 5 + x;
   x = x + 1;
end
```

In the above example, each time the body of code is run, the variable `x` is modified. The value of `x` is checked at the beginning of each execution and the loop is aborted if the condition `x < 30` is not met. Note that the condition which is being evaluated by a while loop is a statement that returns a logical value, similar to those described in Section 1.6. As a demonstration next, it is possible to create an infinite loop. The following loop will run indefinitely.

```
y = 0;
while 1
y = y + 1
end
```

1.7.3 Example: Loop for Newton's Method (MECH 309: Numerical Analysis)

Question 1.6. How can we set up a for loop to find a root using the Newton-Raphson Method?

To demonstrate the use of loops for solving problems, the Newton-Raphson Method will be used to find the root of a polynomial. The Newton-Raphson Method iteratively solves

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)} \tag{1.6}$$

until x_{n+1} differs from x_n by some small value. Consider the polynomial

$$f(x) = x^5 + 2x^3 + 10. \tag{1.7}$$

The derivative with respect to x of the polynomial is

$$f'(x) = 5x^4 + 6x^2 \tag{1.8}$$

To find a root (i.e., one of the roots) of the polynomial given in Equation (1.7), first define a function for the polynomial given in Equation (1.7) as follows.

```
function [ y ] = f( x )
   y = x^5 + 2*x^3 + 10;
end
```

Next, create a function for the derivative with respect to x of the polynomial given in Equation (1.8).

```
function [ y ] = fPrime( x )
   y = 5*x^4 + 6*x^2;
end
```

Given an initial guess for the root (which, in this case, will be -3), a for loop can be used to iteratively solve Equation 1.6 given a fixed number of iterations.

```
x(1) = -3; %Initial guess at root location.
for n = 1:10;
   x(n+1)   =   x(n) - f(x(n))/fPrime(x(n));
end
```

In this case, going from `n = 1` to `n = 10`, yeilds `x` $= -1.4987$, but is this a root of the polynomial in Equation 1.7? Substitution of this value of `x` into Equation (1.7) allows the verification that this `x` value is indeed a root.

```
>> f(x(end))

ans =

     0
```

Thus, a root of the function has been found answering Question 1.6.

Using a for loop is not the only way to find a root of Equation (1.7). In fact, using a for loop is perhaps not the best way to find a root because extra iterations may be taken long after the difference between x_{n+1} and x_n is below some very small tolerance. A while loop with a tolerance, as shown next, may also be used to find find a root of Equation (1.7).

```
n = 1;
x(n) = -3; %Initial guess at root location.
tol = 1e-6; %iteration tolerance
while (n == 1)||( abs(x(n) - x(n-1)) > tol )
    f(x(n))
    x(n+1) = x(n) - f(x(n))/fPrime(x(n))
    n = n+1
end
```

In the above code we have the loop counter `n`, the initial guess of the root `x`, as well as a tolerence, `tol`. Recall that the while loop will execute as long as its argument is `true`. In this example, a short-circuit logical or, denoted `||`, is used. In this case the while loop argument is "`n = 1`" or "the absolute value of `x(n) - x(n-1)` is greater than `tol`". The reason the or statement is needed is that when `n = 1` there is no `x(0)` defined. As such, the "`n = 1`" statement is true when `n = 1`, and so the while loop starts even when `x(0)` is not defined. When `n = 2`, the statement "`n = 1`" is false, but then the statement `( abs(x(n) - x(n-1)) > tol )` is evaluated, and as long as `abs(x(n) - x(n-1))` is strictly greater than `tol`, the while loop continues. Within the loop, the only difference between the for loop and this while loop is that the counter `n` is manually increased.

1.8 Logic (If Else and Case) Statements

1.8.1 If

`if` statements act as checkpoints for a piece of code. The code placed in the body between an `if` statement and the `end` statement will be executed only if the the condition of the `if` statement is met.

```
x = 3;
if x > 0
   'The condition has been met!'
end
```

This code will result in the output

```
ans =

The condition has been met!
```

in the command window. In the above case, the logical returned by the condition $x > 0$ returns true, thus the code is executed. If the condition $x > 0$ were not met, the code would not have executed.

Sometimes it is desirable to use an `if` statement, but also allow for a second block of code to be run if the condition is not met. For this we will use the `else` command.

```
x = 2;
if x > 10
   'The condition has been met!'
else
   'The condition has NOT been met!'
end
```

This code will output

```
ans =

The condition has NOT been met!
```

1.8.2 Switch, Case

`switch` statements are similar to `if` statements. The difference is that `switch` blocks are meant to handle specific scenarios (i.e., $n == 3$), rather than a range of cases (ie. $n > 5$). A `switch` statement will check through a list of cases, and once a `case` is found to be true, will evaluate the code block associated with this `case`. `switch` statements are used as checkpoints in the same way that `if` statements are, but they are meant to handle known or expected scenarios.

```
n = -1;
switch n
 case -1
     disp('case one:   n = -1')
 case 1
     disp('case two:   n = 1 ')
 case 3
     disp('case three: n = 3 ')
end
```

The above piece of code will produce

```
case one:   n = -1
```

An `otherwise` statement can be added to the list of cases to produce an output when none of the cases are met.

```
n = 5;
switch n
 case -1
     disp('case one:   n = -1')
 case 1
     disp('case two:   n = 1 ')
 case 3
     disp('case three: n = 3 ')
 otherwise
     disp('no case condition was met')
end
```

This example of `otherwise` produces

```
no case condition was met
```

Chapter 2

Plotting

2.1 Line Plot

To create a two dimensional line plot, the data that is to be plotted must first be defined. Consider two row matrices of equal size.

```
x = 0:19:100;
y = sin(x);
```

To produce a basic two dimensional line plot the command `plot` is used. The `plot` command accepts as the first argument the `x` dimension of the data, while the second argument is the `y` dimension of the data.

```
plot(x,y)
```

When entered into the command window, the above `plot` command will create Figure 2.1.

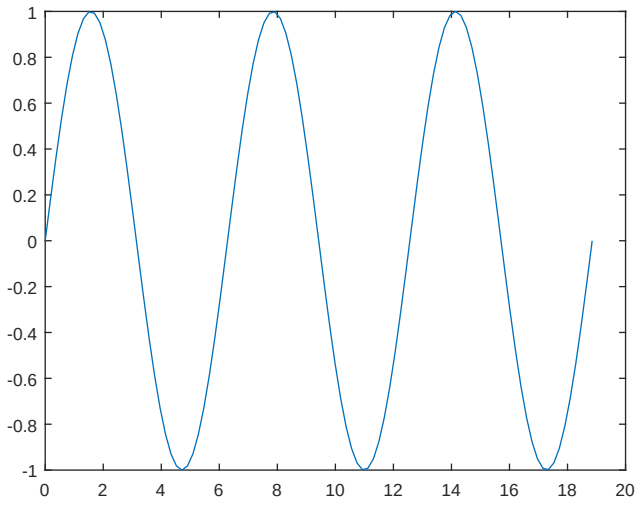


Figure 2.1: Basic two dimensional line plot

It's important to add information to plots when presenting data so that information can be communicated more clearly. Next a few commands will be introduced to control the appearance of the default `MATLAB` plot. The `hold` command will also be used to allow for plotting two lines of the same plot area.

```
x    = 0:6*pi/100:6*pi;
y1   = sin(x);
y2   = cos(x);

plot(x,y1,'b','linewidth',2)
hold on
plot(x,y2,'r','linewidth',2)
grid on
xlabel('Angle (rad)')
ylabel('Output')
ylim([-2 2]);
legend('sin(x)', 'cos(x)')
```

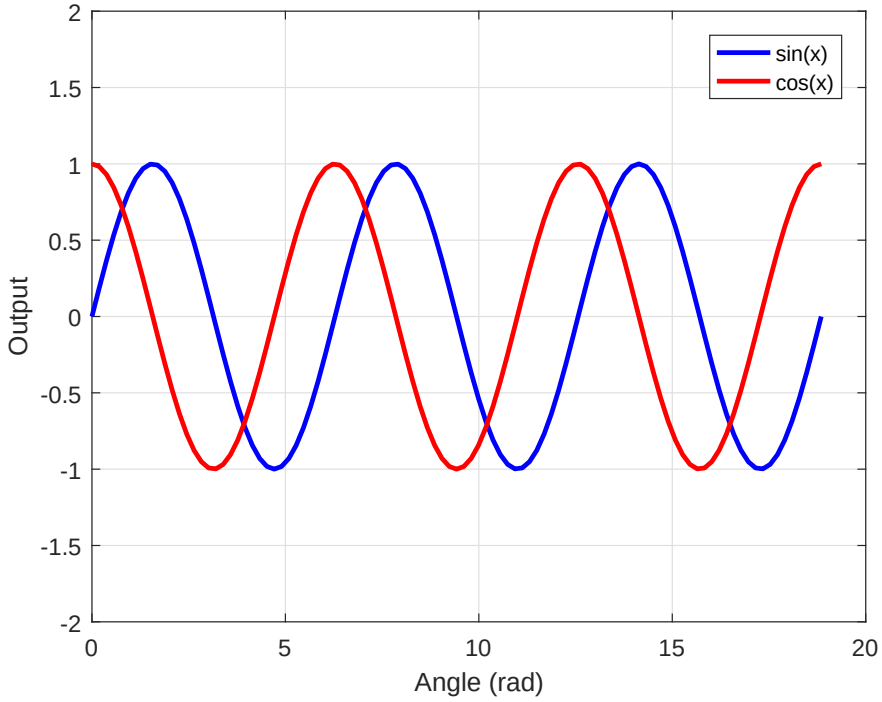


Figure 2.2: Line plot with settings

Chapter 3

Numerical Integration

3.1 Introduction

Numerical integration is used to solve many engineering problems. MATLAB has many built in functions to solve integration problems.

3.2 Basic Integration Example

Here we will evaluate a trivial example of a numerical integration, and in later sections we will add complexity to the problem. Here we will use as an example an object accelerating at some constant acceleration.

Question 3.1. How do we determine the displacement and velocity profile of an accelerating object?

To solve for the displacement and velocity profile of an accelerating object, where the acceleration is given, the equations

$$x_{n+1} = x_n + v_n \Delta t, \tag{3.1}$$

$$v_{n+1} = v_n + a_n \Delta t, \tag{3.2}$$

are numerically integrated. How to select Δt "small enough" is usually problem dependent. To understand where Equation (3.2) comes from, recall the relationship between velocity and acceleration,

$$\frac{\mathrm{d}v(t)}{\mathrm{d}t} = a(t).$$

Making the approximation

$$\frac{\mathrm{d}v(t)}{\mathrm{d}t} \approx \frac{v(t_{n+1}) - v(t_n)}{t_{n+1} - t_n}.$$

leads to

$$\frac{v(t_{n+1}) - v(t_n)}{t_{n+1} - t_n} = a(t_n).$$

Rearranging the above results in Equation (3.2) where $v_{n+1} = v(t_{n+1})$, $v_n = v(t_n)$, $a_n = a(t_n)$, and $\Delta t = t_{n+1} - t_n$. Equation (3.1) can be derived in a similar fashion.

Assuming a constant acceleration of $10\ \mathrm{m/s^2}$, let's set this problem up using a `for` loop in MATLAB.

```
% Define time in (sec)
Dt  = 0.1; % Time step
```

```
t0  = 0; % Initial time
tf  = 10; % Final time
t   = t0:Dt:tf; % Time space

% Initial Conditions
x0  = 0; % (m) Initial Displacement
v0  = 0; % (m/s) Initial Velocity

% Initialize arrays to be filled
v = zeros(size(t)); % Initialize matrix
x = zeros(size(t)); % Initialize matrix
x(1) = x0; % Place initial values in first element
v(1) = v0;

% Acceleration definition
a = 10; % (m/s^2)

% Integration Loop
for n = 1 : length(t)-1

    x(n+1) = x(n) + v(n)*Dt;
    v(n+1) = v(n) + a*Dt;

end
```

3.2.1 Modified Representation

Now, in preparation for using MATLAB's built-in ODE solvers, a slight modification to the representation of the differential equations will be made. MATLAB's solvers are set up to handle problems of the form,

$$\mathbf{y}_{n+1} = \mathbf{y}_n + \mathbf{f}(\mathbf{y}_n, t)\Delta t, \tag{3.3}$$

where $\mathbf{y}$ is a column matrix of the states to be integrated and $\frac{\mathrm{d}\mathbf{y}(t)}{\mathrm{d}t} = \mathbf{f}(t, \mathbf{y}(t))$ is a general function, that could be linear or nonlinear, relating the states $\mathbf{y}$ and time t to the time derivative of $\mathbf{y}$. The above problem will be reformulated with this new representation to get comfortable with the syntax.

```
% Define time in (sec)
Dt  = 0.1; % Time step
t0  = 0; % Initial time
tf  = 10; % Final time
t   = t0:Dt:tf; % Time space

% Initial Conditions
x0  = 0; % (m) Initial Displacement
v0  = 0; % (m/s) Initial Velocity

% Initialize arrays to be filled
x   = zeros(size(t)); % Initialize matrix
```

```
    v   = zeros(size(t)); % Initialize matrix
    x(1)= x0; % Place initial values in first element
    v(1)= v0;

    % Make a generalized state vector y = [ pos; vel ]
    y   = [x; v];

    % Acceleration definition
    a   = 10; % (m/s^2)

    % Integration Loop
    for n = 1:(length(t)-1)

        % State derivative relationship dy/dt = f(y,t)
        f    = [ y(2,n); a ];
        % Integration
        y(:,n+1) = y(:,n) + f*Dt;

    end
```

This example produced the same result, but uses a generalized state variable **y** in place of individual variable definitions for each state. The individual rows of the resultant matrix **y** each represent a state, while the individual columns correspond to a time in the trajectory of the states.

3.3 MATLAB's ODE Solvers

In this section the same ODE will be solved as in the previous section, but the built-in `MATLAB` function `ode45` will be used in place a custom user-made integration loop.

First, the function $\frac{\mathrm{d}\mathbf{y}(t)}{\mathrm{d}t} = \mathbf{f}(t, \mathbf{y}(t))$ must be coded as a function. This function will receive a column matrix of states as its input, and return a column matrix according to $\mathbf{f}(t, \mathbf{y}(t))$ as its output.

```
function [ dydt ] = f( t, y )

    % Define acceleration magnitude
    acc = 10; % (m/s^2)

    % Output derivatives
    xdot = y(2); % dx/dt = v
    vdot = acc; % dv/dt = a

    % Format output
    dydt = [ xdot; vdot ];

end
```

This function `f` should be saved to the working directory so that we may call it from our main script. Now, the problem will be set up as before, but this time the integration will be performed by a call to `ode45`, a built-in numerical integration tool.

```
% Define time in (sec)
Dt  = 0.1; % Time step
t0  = 0; % Initial time
tf  = 10; % Final time
t   = t0:Dt:tf; % Time space

% Initial Conditions
x0  = 0; % (m) Initial Displacement
v0  = 0; % (m/s) Initial Velocity
y0  = [x0; v0];

% Integrate Differential Equation
[T,Y]   = ode45(@f,t,y0);
```

The result `Y` from `ode45` will be similar to the results we received in the previous examples, only this time the result will come in the form of a matrix where the individual columns represent the state variables and the rows represent the distinct instances in time.

The advantage of using `MATLAB's` ODE solvers, and the according state and state derivative representation, is that differential equations of any order or level of complexity can be integrated. As long as we define a function $\mathbf{f}(t, \mathbf{y}(t))$ which returns the derivatives of each of the states in the column matrix $\mathbf{y}$, any ODE can be solved with the same call as before,

```
[T,Y]   = ode45(@f,t,y0);
```

3.3.1 Example: Numerical Integration of Object in Flight (MECH 220: Dynamics)

We will formulate an additional example to expand on the use of `MATLAB`'s ODE solvers.

Example 3.1. An object is launched upwards from the ground with an initial velocity of 30 m/s. Taking into account the effects of air resistance, find the time required for the object to reach its peak altitude.

Mass	m	20	kg
Drag Coefficient	c_D	0.5	$-$
Area	A	1	m^2
Air Density	ρ	1.225	$\frac{\mathrm{kg}}{\mathrm{m}^3}$
Acceleration of Gravity	g	9.81	$\frac{\mathrm{m}}{\mathrm{s}^2}$

For this problem the sign convention will be defined as positive upwards and negative downwards. Air resistance, or drag, represented by f_d, will be modeled as

$$f_d = \frac{1}{2}\rho v^2 c_D A. \tag{3.4}$$

The gravitational force,

$$f_g = gm, \tag{3.5}$$

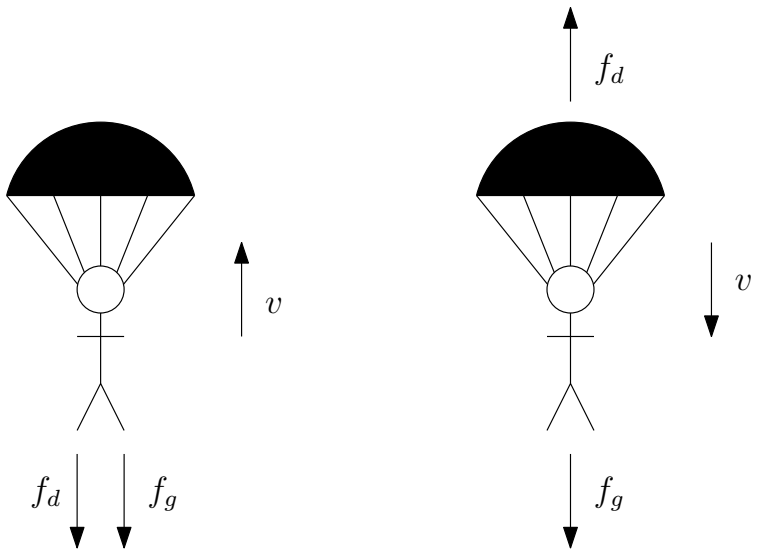


Figure 3.1: Body in flight.

will be applied to the object throughout its flight as well. Applying Newton's second law gives

$$\begin{aligned} \sum_{i=0}^{k} f_i &= ma \\ a &= \sum_{i=0}^{k} f_i/m = (f_d - f_g)/m. \end{aligned}$$

It is important to note that drag will always act opposite the object's velocity. As such, the acceleration equation is modified to be

$$a = (-\text{sign}(v) f_d - f_g)/m$$

where sign(v) returns a value of 1 or -1 depending on if or if not v is positive of negative.

As in the previous examples, the states to be integrated will be position, x and velocity, v,

$$\mathbf{y}(t) = \begin{bmatrix} x(t) \\ v(t) \end{bmatrix}$$

and our solver will have access to the state derivatives through our function `f`.

$$\frac{\mathrm{d}\mathbf{y}(t)}{\mathrm{d}t} = \begin{bmatrix} \dot{x}(t) \\ \dot{v}(t) \end{bmatrix} = \begin{bmatrix} v(t) \\ a(t) \end{bmatrix} = \mathbf{f}(t, \mathbf{y}(t)).$$

```
function [ dydt ] = f( t, y )

    % Define system constants
    g = 9.81; % (m/s^2)
    mass = 20; % (kg)
    CD = 0.5; % (-)
    A = 1; % (m^2)
```

```
    rho = 1.225;% (kg/m^3)

    % Forces
    v = y(2);
    fDrag = CD*A*(1/2)*rho*(y(2)^2); % (N) proportional to v?2
    fGravity = mass*g; % (N)

    % Output derivatives
    xdot = y(2); % dx/dt = v
    vdot = ( -sign(v)*fDrag - fGravity )/mass; % dv/dt = a = sum(F)/m

    % Format output
    dydt = [xdot; vdot];

end
```

Again, using the same format as the earlier examples, define a time space, define the initial conditions, and call `ode45` to perform the integration.

```
% Define time in (sec)
Dt = 0.1;       % Time step
t0 = 0;         % Initial time
tf = 5;         % Final time
t  = t0:dt:tf;  % Time space

% Initial Conditions
x0 = 0;        % (m)   Initial Altitude
v0 = 35;       % (m/s) Initial Upwards Velocity
y0 = [x0; v0];

% Integrate Differential Equation
[T,Y]   = ode45(@f,t,y0);
```

The results of the numerical integration can be plotted to get an idea of the trajectory of the states.

```
figure(1)
plot(T,Y,'linewidth',2)
grid on
xlabel('Time (sec)')
ylabel('Altitude, Velocity')
legend('Altitude (m)', 'Velocity (m/s)')
title('Example: Launched Object')
```

Finally, the time where maximum altitude is reached can be identified using logical indexing as outlined in Section 1.4.2.

```
peakAltitude  = max( Y(:,1) )
peakTime      = T( Y(:,1) == peakAltitude )
```

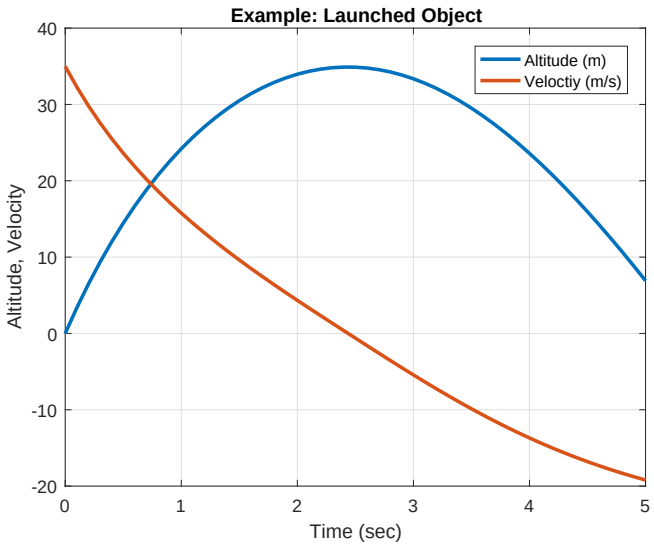


Figure 3.2: Result of numerical integration

```
peakAltitude =

   34.8957


peakTime =

    2.4000
```

The solution to the initially posed problem is found. The peak altitude reached is 34.9 meters, and the time required to reach this altitude is 2.4 seconds.

3.3.2 Example: Numerical Integration of Draining Tank (MECH 231: Fluid Mechanics)

An additional example to expand on the use of MATLAB's ODE solvers will now be given.

Example 3.2. Consider a large tank of fluid being drained, as shown in Fig. 3.3. The height of the fluid can be described by

$$\frac{\mathrm{d}h(t)}{\mathrm{d}t} = -\sqrt{2g}\left(\frac{d}{D}\right)^2\sqrt{h(t)}, \tag{3.6}$$

where d is the diameter of the outlet, D is the diameter of the tank, g is the acceleration due to gravity, and $h(t)$ is the height of the fluid at time t. Given a tank with outer diameter $D = 1$ m, outlet diameter $d = 0.05$ m, acceleration due to gravity $g = 9.81$ m/s^2 and initial fluid height $h(0) = 10$ m, find the amount of time it takes to completely drain the tank, that is, the time it takes to reach $h(t) = 0$.

To begin, write a function the describes the right hand side of Equation (3.6).

```
function [ dot_h ] = tankflow(t, h )
% @tankflow is the ODE defining the height of the fliud in the tank
```

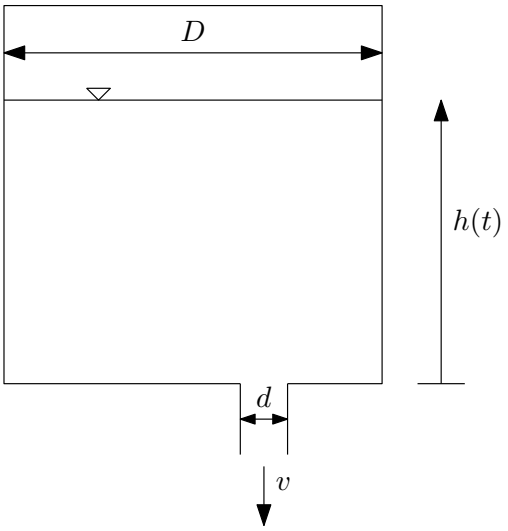


Figure 3.3: Water tank.

```
g = 9.81; % m/s^2
d = 0.05; % m
D = 1;    % m

dot_h = -sqrt(2*g)*(d/D)^2*sqrt(h);
end
```

To numerically integrate the system, use `ode45`. Try a simulation time of 600 seconds.

```
t = 0:0.1:600; % s, time span of simulation
h0 = 10; % m, initial fluid height
[T,h] = ode45(@tankflow,t,h0);
```

By plotting or examining the data, it is apparent that $h = 0$ around $t = 571$ s.

Note, in this case, there is an analytic solution to Equation (3.6), that being

$$h(t) = \left(-\sqrt{0.5g} \left(\frac{d}{D} \right)^2 t + \sqrt{h_o} \right)^2 ,$$

where h_o is the height of the fluid at time $t = 0$. To find the time required to drain the tank, isolate t and set $h(t_d) = 0$, to get

$$t_d = \left(\frac{d}{D} \right)^2 \sqrt{\frac{2h_o}{g}}. \tag{3.7}$$

Using Equation (3.7) solve for t_d and compare this solution with your original answer.

Chapter 4

Curve Fitting

4.1 Polynomial Fitting

Question 4.1. How do we fit a polynomial to a set of data points?

To answer Question 4.1 we can use the built-in `MATLAB` function `polyfit`. This function accepts as arguments the order of the desired polynomial, and the data to be fit using the polynomial. The result of the function is a matrix containing the polynomial coefficients.

```
p = polyfit(x,y,n)
```

The result **p** of the call to `polyfit` will be the coefficients arrangement from highest order term to lowest order term, equation 4.1 demonstrates for the general case n^{th} order polynomial.

$$p = p(1)x^n + p(2)x^{n-1}...p(n)x + p(n+1) \tag{4.1}$$

Example 4.1. In this example we will create a set of data with some added noise, then we will generate a fitted polynomial to represent the data. First we must define our data,

```
xdata = 0:0.05:1;                          % Define independent variable
ydata = exp(5*xdata) + randn(size(xdata));  % Creat data to be fit.
```

Next, we can call `polyval` to fit a polynomial to the data. For this example we will begin with a second order polynomial.

```
n = 4;
p = polyfit(xdata,ydata,n);  % Fit fourth order polynomial
```

Instead of writing our own function to evaluate the polynomial with coefficients defined in `p`, we can use the function `polyval`. This function takes in the matrix of polynomial coefficients (in the same format as the output of `polyfit`), and evaluates the polynomial of order `length(p) - 1` using these coefficients.

```
yFit = polyval(p,x);  % Evaluate polynomial
```

We can plot the result of our fit over the generated data to visual evaluate our results.

```
figure
scatter(xdata,ydata,'bx','linewidth',2)
hold on
plot(xdata,yFit,'r','linewidth',1.5)
grid on
xlabel('x')
ylabel('y')
legend('Data', 'Fourth Order Polynomial Fit','Location','best')
hold off
```

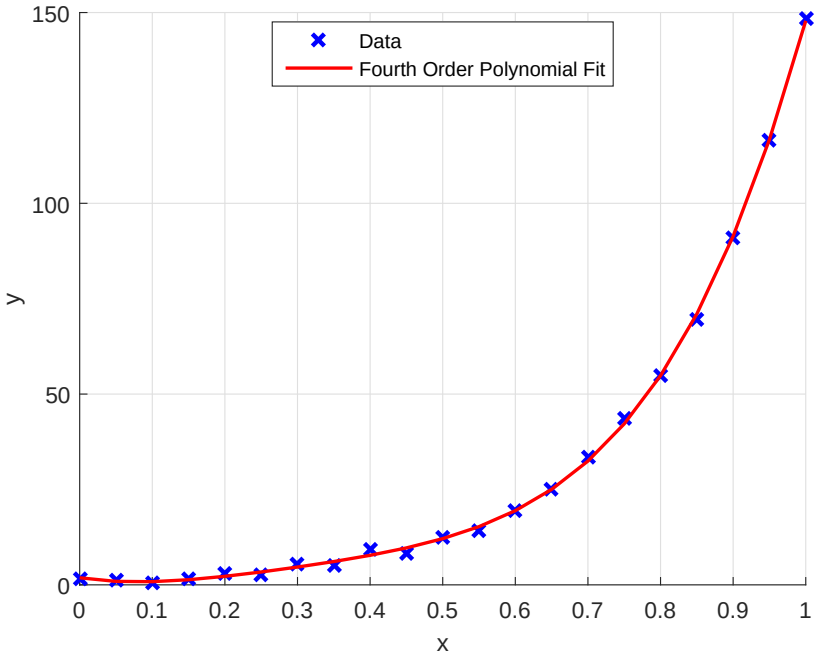


Figure 4.1: Polynomial fit using `polyfit` compared to raw data.

4.2 Nonlinear Function Fitting

4.2.1 Example: Curve Fit of Atmospheric Data (MECH 331: Fluid Mechanics)

Question 4.2. How do we fit an arbitrary nonlinear function to a set of data points?

MATLAB's basic fitting tools also allow us to fit functions of our own definition to a set of data. We must simply provide a reference to the function which defines our fit, and an initial guess `x0` at the values of the coefficients in the form of a column matrix of coefficients.

```
x = lsqcurvefit(fun,x0,xdata,ydata)
```

Example 4.2. We will use some data from the *U.S. Standard Atmosphere Air Properties* and fit an exponential model to represent air density as a function of altitude.

```
alt = [0 1000 2000 3000 4000 5000 6000 7000 8000 9000 10000 ...
       15000 20000 25000 30000 40000 50000]/1000;   %(km)Altitude
rho = [1.225 1.112 1.007 0.909 0.819 0.736 0.660 0.590 0.526 ...
       0.4671 0.414 0.195 0.089 0.040 0.018 0.004 0.001]; %(kg/m^3)rho
```

We can define the function to model the air density as function of altitude, for this example an exponential model will be used.

$$\rho = x_1 \exp\left(-\frac{alt}{x_2}\right) \tag{4.2}$$

We must create a function in MATLAB to represent equation 4.2.

```
function [ rho ] = f( x, alt )
rho = x(1)*exp( -alt/x(2) );
end
```

Now that the function behind our model has been defined, we can make an initial guess at what the coefficients might be, then call `lsqcurvefit` to modify the coefficients until a satisfactory fit is achieved.

```
x0 = [5; 5];                        %Initial guess at coefficients
x  = lsqcurvefit(@f,x0,alt,rho);    %Call to lsqcurvefit
```

For this specific example, the resultant coefficients found were

```
x =

    1.2560
    8.9418
```

Using these model coefficients, we can evaluate our model at the sites of the independent variable `alt`,

```
rhoModeled   = f(x,alt);
```

and plot the results to compare our fit to the raw data.

```
figure
scatter(alt,rho,'bo')
hold on
plot(alt,rhoModeled,'r','linewidth',1.5)
grid on
xlabel('Altitde (km)')
ylabel('Air Density \rho (kg/m2)')
title('1976 Standard Atmosphere, Exponential Model')
legend('Data', 'Exponential fit')
hold off
```

Using `lsqcurvefit` we have answered Question 4.2. We can use this same method to fit nonlinear functions of any nature to a set of x-y data.

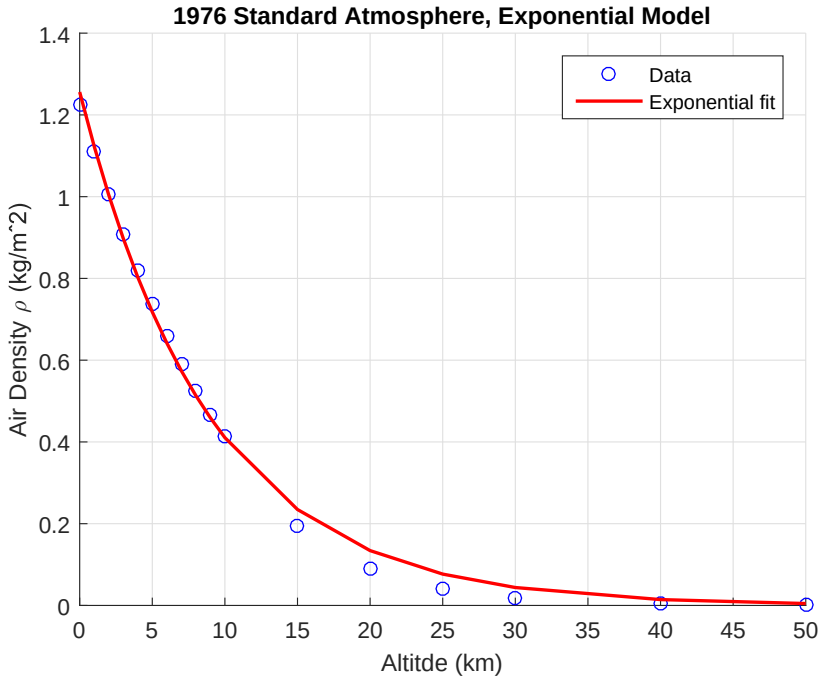


Figure 4.2: Exponential fit of air density relationship to altitude

4.2.2 Example: Curve Fit of Drag forces on a Circular Cylinder (MECH 331: Fluid Mechanics)

Using conservation of momentum and control volume analysis, one is able to calculate an approximation of the drag forces acting on an arbitrary solid body. For the purposes of this example, we will consider a circular cylinder with a diameter of $d = 0.05$ m.

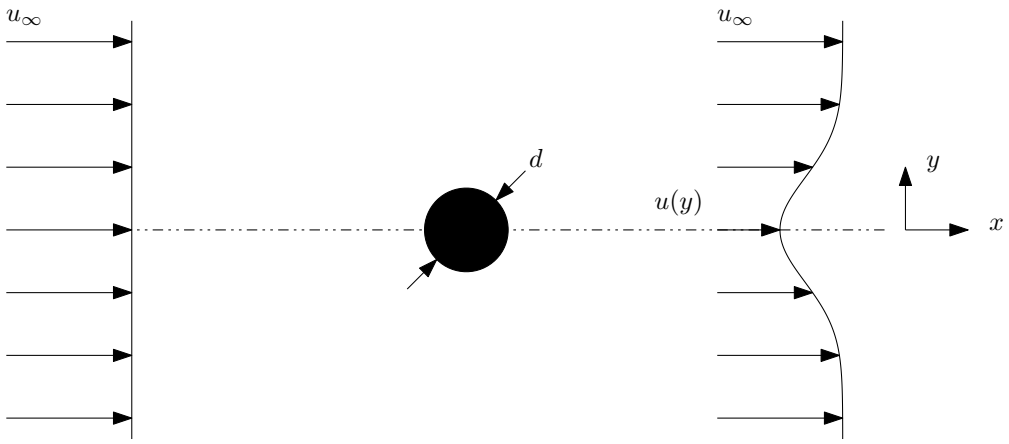


Figure 4.3: Circular cylinder in flow.

The drag force is obtained by

$$D = \int_{-\infty}^{\infty} \rho u(y) \left(u_\infty - u(y)\right) \mathrm{d}y. \tag{4.3}$$

Literature states that the drag coefficient c_d is

$$c_d = \frac{D}{\frac{1}{2}\rho {u_\infty}^2 d}. \tag{4.4}$$

In the case of a long circular cylinder, the drag coefficient is approximately $c_d \approx 1.2$.

An experiment is performed and the velocity profile downstream of a circular cylinder is taken. The freestream velocity is $u_\infty = 10\ \mathrm{m/s}$. Assume the air density is constant at $\rho = 1\mathrm{kg/m}^3$. The experiment is repeated further two times, giving three total data sets, `u1`, `u2` and `u3`.

```
u1 = [9.971620 10.001134  9.970918  9.664542  8.292998  6.993700 ...
      8.130454  9.645742  9.983267  9.902597 10.088830];

u2 = [9.890089  9.980984  9.940466  9.730433  8.126774  6.850396 ...
      8.300156  9.702786  9.944626  9.858009  9.958374];

u3 = [9.985393 10.024815  9.941674  9.628729  8.186961  6.832944 ...
      8.061358  9.517864  9.918589  9.968973 10.001156];
```

Alternatively, this data can be found by loading `cylinder_example.mat`. For each traverse, eleven equally spaced y locations are sampled, ranging from $y = -0.25$ m and $y = 0.25$ m.

1. Use `MATLAB` to obtain a column matrix for the y locations of the different data points.

   Enter `y=linspace(-0.25,0.25,11)` into the command window

2. Calculate the drag coefficient c_d for each data set, and compare to literature.

   Note: since you do not have a function describing the velocity distribution, you cannot use numerical integration techniques presented in Chapter 4.

   One recommended solution is to use `trapz` to use the trapezoid method to approximate the area under the integral in Equation 4.3. For example, for the first data set `u1`, your `MATLAB` code should resemble the following.

   ```
   c_d1 = trapz(y,rho*u1.*(uinf-u1))./(0.5*rho*uinf^2*d)
   ```

   Repeat the process wit data sets `u2` and `u3`. You should return the following drag coefficients.

   $$\begin{aligned} c_{d1} &= 1.1629 \\ c_{d2} &= 1.1980 \\ c_{d3} &= 1.2358 \end{aligned}$$

3. Interpolate additional data points, (101 total points) using `interp1`. Compute the value of c_d again. Is it the same as the drag coefficient in question 2? Why or why not?

First, create a set of interpolated y positions with `yint = linspace(-0.25,0.25,101)`. Type `u1int = interp1(y,u1,yint)` to linearly interpolate the data to 101 spaces, then use `trapz` to calculate the interpolated drag coefficient. You should calculate the following drag coefficients.

$$\begin{aligned} c_{d1} &= 1.1873 \\ c_{d2} &= 1.2257 \\ c_{d3} &= 1.2616 \end{aligned}$$

4. Looking at the data, it appears that the difference $u_\infty - u(y)$ is approximately a normal distribution. Fit a Gaussian to the data, using `MATLAB` code and `lsqcurvefit`, then recalculate the value of c_d.

   Create a function titled `u_y.m`.

```
function [ u ] = u_y( x,ydata )
% This function plots a gaussian distribution subtracted from uinf,
% with amplitude and sigma as variables.
% ydata represents the measurement points y.
a = x(1); % a is the amplitude
b = x(2); % b is sigma

u = 10 - a* exp(-((ydata)/b).^2);

end
```

   To solve for the values of `a` and `b`, use `lsqcurvefit`. The command

```
x=lsqcurvefit(@u_y,x0,y,u1)
```

   where `x0` is an initial guess for each variable. Try an initial guess of `a=3, b=0.1`.

   Once more accurate values of `a` and `b` are found, use `trapz` to find c_d.

```
ylong = linspace(-0.25,0.25,1001);
% You can test this over a larger range. See if you get the same c_d!
c_dgauss1 = trapz(ylong,rho*u_y(x,ylong).*(uinf-rho*...
  u_y(x,ylong))./(0.5*rho*uinf^2*d)
```

   Using the same technique for runs 2 and 3, the following data should be found.

Run	a	b	c_d
1	3.018	0.0686	1.155
2	3.162	0.0657	1.143
3	3.149	0.0701	1.217

Chapter 5

Optimization

5.1 Introduction

Optimization is used to minimize (or maximize) the value of an objected function. These optimization techniques can be applied to a wide variety of problems with the objective of identifying minima and maxima of a function.

5.2 fsolve

Question 5.1. How can we find the solution to a nonlinear function?

`fsolve` is a built-in `MATLAB` function meant to find the zeros of an arbitrary function `f`. The root finding process can be applied to nonlinear expressions, allowing `fsolve` to deal with complicated problems using a simple syntax similar to that shown in Chapter 3, which deals with numerical integration.

`fsolve` will attempt to bring the result of our function `f` equal to zero by varying the input value x.

$$f(x) = 0 \tag{5.1}$$

We will use a trivial example with a known solution to begin with.

$$f_x = x^2 - 2 \tag{5.2}$$

We must set up our objective function `f`, to be called later by the iterative solver.

```
function [ y ] = f( x )
y = x^2 - 4;
end
```

To initiate the solution process, we must simply provide an initial guess at the solution,

```
x0 = 3;
```

and then make a call to `fsolve` while pointing to our objective function using `@f`.

```
xSolution = fsolve(@f,x0)
```

Behind the scenes, `fsolve` will run through a root finding process to arrive at our solution value, which due to the triviality of the objective function, we knew to be $\mathbf{x} = 2$. Thus, using `fsolve` we have answered Question 5.1.

```
xSolution =

    2.0000
```

5.2.1 Example: Iterative Solution To Isentropic Flow (MECH 430: Fluid Mechanics 2)

Example 5.1. Use `fsolve` to find the exit Mach number of air ($\gamma = 1.4$) flow in a converging-diverging nozzle with $\frac{A}{A^*} = 2$. Assume that the flow is sonic at the throat ($M_{throat} = 1$). Note that there can be a subsonic, as well as supersonic exit solution to this problem. *Note, this example does not require extensive knowledge of Fluid Mechanics, the underlying principle being demonstrated is simply solving an equation which cannot be solved directly.*

The governing equation for this problem is

$$\frac{A}{A^*} = \frac{1}{M}\left[\left(\frac{2}{\gamma+1}\right)\left(1+\frac{\gamma-1}{2}M^2\right)\right]^{\frac{(\gamma+1)}{2(\gamma-1)}} \tag{5.3}$$

We know $\frac{A}{A^*}$ from the problem definition, and we know that for air ($\gamma = 1.4$), so we are simply trying to find a value M which satisfies the given conditions. To be clear, we are trying to find M such that the right side of Equation 5.3 equals the left side of Equation 5.3. We could write Equation (5.3) as

$$2 = \frac{1}{M}\left[\left(\frac{2}{1.4+1}\right)\left(1+\frac{1.4-1}{2}M^2\right)\right]^{\frac{(1.4+1)}{2(1.4-1)}},$$

to clearly see that we are attempting to find M such that the right side of Equation 5.3 equals the left side of Equation 5.3, but for generality, we will leave A/A^* and γ as variables. Since `fsolve` attempts to bring a function to zero, we will rearrange Equation 5.3.

$$y = 0 = \frac{1}{M}\left[\left(\frac{2}{\gamma+1}\right)\left(1+\frac{\gamma-1}{2}M^2\right)\right]^{\frac{(\gamma+1)}{2(\gamma-1)}} - \frac{A}{A^*} \tag{5.4}$$

Writing this as a function $y = f(M)$ in `MATLAB` we will arrive at the following,

```
function [ y ] = fNozzle( M )

areaRatio = 2;
gamma     = 1.4;

y = (1/M)*...
    ((2/(gamma+1))*(1+((gamma-1)/2)*(M^2)))^((gamma+1)/(2*(gamma-1)))...
    - areaRatio;
end
```

Since we are trying to find a root of the equation, we must provide an initial guess, then call `fsolve` as before.

```
x0_subsonic = 0.2;
x_subsonic  = fsolve(@fNozzle,x0_subsonic)
```

This yields our first result, the subsonic exit solution of the converging-diverging nozzle.

```
x_subsonic =

    0.3059
```

Now, we will initialize the solver with a supersonic initial guess.

```
x0_supersonic   = 2.0;
x_supersonic    = fsolve(@fNozzle,x0_supersonic)
```

This yields our second result, the supersonic exit solution of the converging-diverging nozzle.

```
x_supersonic =

    2.1972
```

Both these results can be verified as accurate when compared to the values found in a reference table of relationships for isentropic flow.

5.2.2 Options

It can often be useful to get some information about what goes on behind the scenes in `fsolve`, to do this we can modify the default options, and pass our new options as an argument to `fsolve`. As an example, we'll check out what goes on at each iteration of the solution process.

```
options   = optimoptions(@fsolve,'Display','iter');
xSolution = fsolve(@f,x0,options)
```

Now that we've asked `fsolve` to tell us about the iterative process, we should get an output similar to the following,

```
                                        Norm of      First-order
Iteration  Func-count     f(x)          step         optimality
    0          2              25                           30                1
    1          4        0.482253       0.833333           3.01
    2          6     0.000659572       0.160256          0.103
    3          8     1.67777e-09     0.00640002       0.000164
    4         10     1.10976e-20    1.02401e-05       4.21e-10
```

Remark 5.1. Note that the column labeled `f(x)` in the output has `f(x)` = 25 at iteration 0, but we know `f(x0)` $=$ `f(3)` $= 3^2 - 4 = 5$. The output is not wrong, this result is actually the function evaluation used by `fsolve`'s root finding process. To find a root, `fsolve` attempts to minimize $\|$`f(x)`$\|^2$, thus the displayed output is actually reporting $\|$`f(x)`$\|^2$ at each iteration.

5.3 fmincon

Question 5.2. How can we find the minimum value of a function subject to constraints?

MATLAB's function `fmincon` is similar in usage to `fsolve`, but instead of finding the the roots of the objective function, `fmincon` attempts to minimize the objective function, subject to user defined constraints. Before diving into the implementation of `fmincon`, it is important to define and understand the user defined settings and constraints. `fmincon` with all possible constraint types is defined as follows.

```
x = fmincon(f,x0,A,b,Aeq,beq,lb,ub,nonlcon)
```

The first argumnet of `fmincon` is referring to the objective function `f`, while the second argument is the initial guess $\mathbf{x_0}$. The significance of the remaining arguments will be outlined in the following subsections.

Remark 5.2. The input of the objective function, **x** can be a single value, or a column matrix of values for a multi variable objective function.

5.3.1 Linear Inequalty Constraints

This subsection will deal with the input arguments **A** and **b** which are used to define linear inequality constraints. The inequality constraints are evaluated in the form,

$$\mathbf{Ax} \leq \mathbf{b} \tag{5.5}$$

where **A** is a matrix with number of rows equal to the number of inequality constraints to be satisfied, while the number of columns equal to the number of elements in the input **x**. **b** is a column matrix with number of rows equal to the number of linear inequality constraints. This format is useful since it is generalized to be used with any number of constraints and **x** of any size.

Remark 5.3. Note that the standard format for linear equality constraints is $\leq$, thus constraints in the form of $\geq$ must be modified accordingly. The desired result may be achieved by multiplying both sides of a $\geq$ inequality by -1.

Example 5.2. If we would like to ensure that the conditions $x_1 \leq 2$ and $x_2 \geq 3$ for a two value x, we can formulate this in the following format.

$$\mathbf{Ax} = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix} \begin{bmatrix} x1 \\ x2 \end{bmatrix} \leq \mathbf{b} = \begin{bmatrix} 2 \\ -3 \end{bmatrix} \tag{5.6}$$

In MATLAB we simply define the two matrices **A** and **b** as described in Section 1.4.

```
A = [1 0; 0 -1];
b = [2; -3];
```

5.3.2 Linear Equality Constraints

Linear equality constraints are defined in the same manner as linear inequality constraints. The method is described in Subsection 5.3.1, this time with a matrix of coefficients $\mathbf{A_{eq}}$, and a column matrix of values $\mathbf{b_{eq}}$ such that,

$$\mathbf{A}_{eq}\mathbf{x} = \mathbf{b}_{eq} \tag{5.7}$$

5.3.3 Lower and Upper Bounds

The variables arguments `lb` and `ub` are meant to ensure that **x** satisfies the equation,

$$\texttt{lb} \leq \mathbf{x} \leq \texttt{ub}. \tag{5.8}$$

To satisfy this condition, `lb` and `ub` are defined as column matrices with the same number of rows as x, the input to the objective function.

5.3.4 Nonlinear Constraints

Nonlinear constraints may also be defined to constrain an optimization problem. Nonlinear equality, and nonlinear inequality constraints are defined using a separate function which is called during evaluation.

$$\mathbf{c}(\mathbf{x}) \leq 0 \tag{5.9}$$
$$\mathbf{c_{eq}}(\mathbf{x}) = 0 \tag{5.10}$$

To define the nonlinear constraint function, we define a function where the first output is $\mathbf{c}(\mathbf{x})$ and the second output is $\mathbf{c_{eq}}(\mathbf{x})$. Note that if there are multiple nonlinear constraints to be satisfied, these outputs can be in the form of column matrices with number of rows equal to the number of constraints of that type.

```
function [c,ceq] = nonlcon(x)
 c   = ...
 ceq = ...
end
```

5.3.5 Examples

Example 5.3. Here we will minimize a simple mathematical function subject to select constraints to demonstrate the use of `fmincon`. The objective function we will minimize will be,

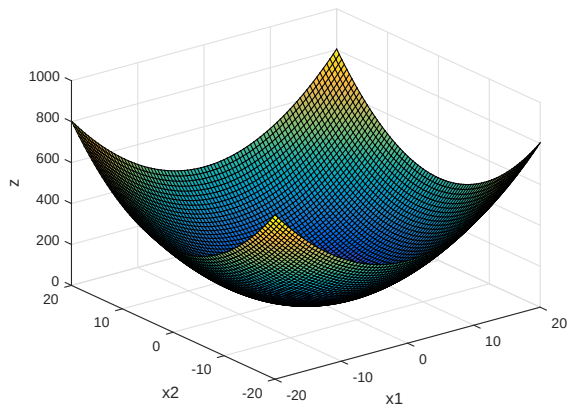
$$z(x_1, x_2) = x_1^2 + x_2^2 + 3. \tag{5.11}$$

We can define this as a `MATLAB` function for use in our optimization process.

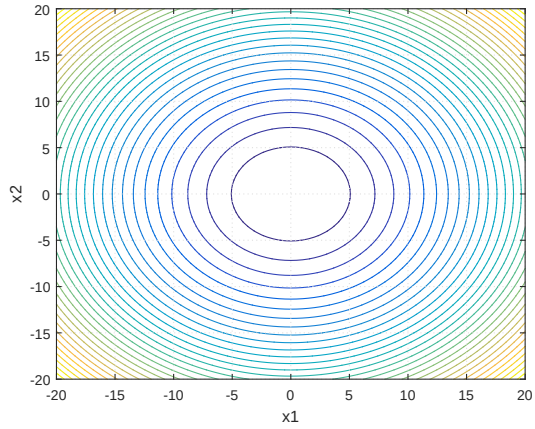
```
function [ z ] = f( x )

  z = x(1).^2 + x(2).^2 + 3;
end
```

This objective function represents a simple bowl function which is shown in Figure 5.1(a)
We will constrain this problem using the following constraints,

$$x_2 \geq \frac{1}{2}x_1 + 10 \tag{5.12}$$
$$x_2 = (x_1 - 5)^2 + 4 \tag{5.13}$$



(a) Surface plot of objective.



(b) Contour plot of objective.

Figure 5.1: Objective function.

Given such constraints, we must convert them to a format in accordance with MATLAB's defined constraint format.

$$\frac{1}{2}x_1 - x_2 \leq 10 \tag{5.14}$$

$$x_2 - (x_1 - 5)^2 - 4 = 0 \tag{5.15}$$

Our first constraint, Equation 5.14 is a linear inequality constraint which can be setup using the following matrices **A** and **b**.

```
A = [0.5 -1];
b = [-10];
```

Since our second constraint, Equation 5.15, is a nonlinear constraint, so we will need to represent it using a function.

```
function [ c, ceq ] = nonlcon( x )

   c   = [];
   ceq = x(2) - ( x(1) - 5 ).^2 - 4;
end
```

Remark 5.4. If a constraint type will not be used, its definition can be replaced by `[]`.

Now that we have defined our constraints, we can make an initial guess $\mathbf{x_0}$, then make a call to `fmincon`, omitting the constraint types which will not be used.

```
x0  = [-5; 20];
x   = fmincon(@f,x0,A,b,[],[],[],[],@nonlcon)
```

Once `fmincon` has done its work, we should be returned with a solution to our problem.

```
x =

    2.3238
    11.1619
```

We can visualize this result, as well as the constraints on a contour plot of our objective function in Figure 5.2. Note that for Equation 5.14 to be satisfied, the solution must lie on or above the red line, and for Equation 5.15, the nonlinear equality constraint, the equation must lie on the blue curve. The solution found by `fmincon` is marked by the $\times$, this is the minimum value of the objective function which can be achieved subject to the constraints.

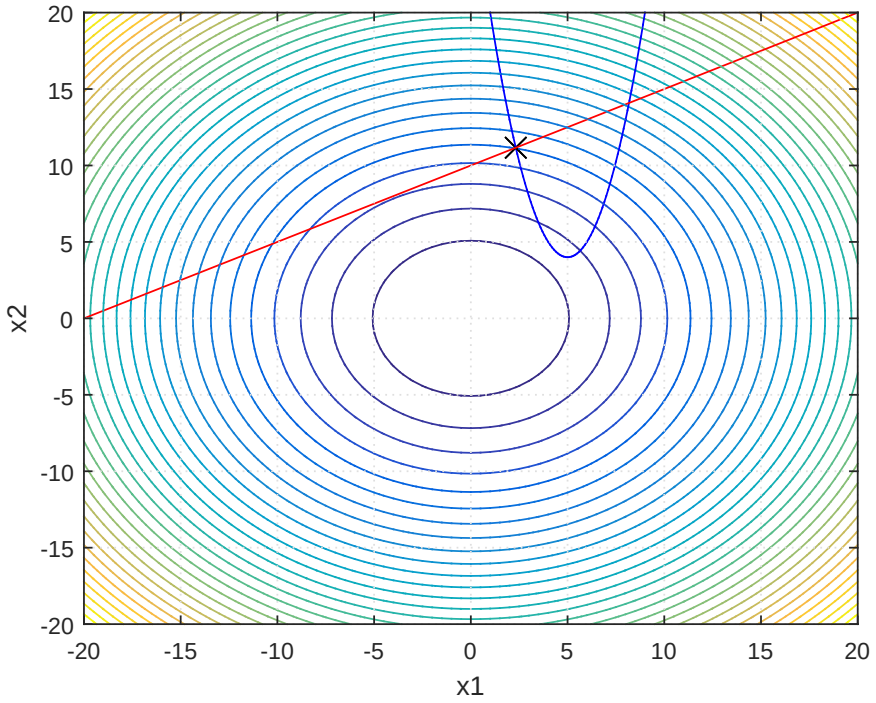


Figure 5.2: Result of `fmincon` shown with constraints

Chapter 6

Simulink

6.1 Introduction

The goal of this chapter is to introduce Simulink, a valuable `MATLAB` toolbox. It is a block diagram environment to help run simulations with a user-friendly graphical user interface. Several packages are available online that can be used to interact with sensors, motors and microcontrollers, such as the Simulink Support Package for Arduino.

This chapter will focus on Simulink's potential application to simulating nonlinear ordinary differential equations. By the end of this chapter, students should be able to solve ODEs using Simulink.

6.2 Getting started

To open Simulink, click on the `HOME` button on the toolstrip, located at the absolute top left of your `MATLAB` window. Next, click **New** $>$ **Simulink** $>$ **Simulink Model** to open Simulink. To open a blank Simulink file, click on the icon labeled **Blank Model**.

6.3 Basic Example - Part I

In this example we will walk through a basic simulation in which a sine wave's amplitude is doubled, and then the result is plotted. Before starting, be sure to save your `.slx` file in your current folder.

Library Browser

Most basic operations and processes necessary in Simulink are represented by blocks found in the Library Browser. The Library Browser has an icon in the Simulink toolbar, or can alternatively be opened by pressing `Ctrl + Shift + L` or by clicking **View** $>$ **Library Browser**.

Question 6.1. What if we have trouble finding a specific block?

The search bar in the Library Browser is a valuable tool to find any block available in the Simulink Library. Remember that some functions can be described in several different ways. For example there is no block to create a cosine wave, but the sine wave can be created with a phase shift of 90°, which is the same function.

Common Blocks

Most Simulink simulations require at least some kind of input (**Sources**) as well as some kind of output (**Sinks**). In this example, the sine wave and the cosine wave are the two sources, and the plotted results will be the output.

First, open the Library Browser. Next, select **Simulink** $>$ **Sources** and right-click on the **Sine Wave** block. Select `Add block to model`. Then, add a **Scope** block to your model, found in **Simulink** $>$ **Sinks**. Finally, add a **Gain** block to your model, found in **Simulink** $>$ **Math Operations**. Alternatively, these blocks can be added by dragging them from the Library Browser to your model.

The model should now resemble Fig. 6.1

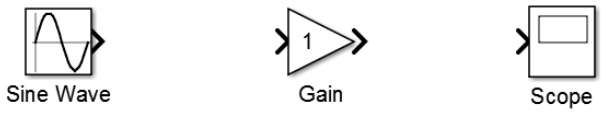


Figure 6.1: Simulink blocks in the model.

Wires

These blocks currently do not interact with each other and must be connected by wires. Click on the $>$ on the right of the **Sine Wave** block, and drag your mouse to the $>$ on the left of the **Gain** block. These two blocks are now connected by a wire. Repeat the process to connect the **Gain** and **Scope** blocks. Your model should now resemble Fig. 6.2.

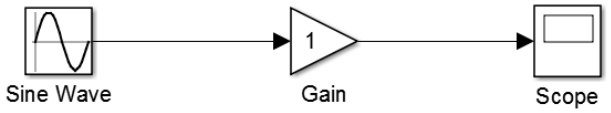


Figure 6.2: Simulink blocks in the model, now connected by wires.

Additional settings

Double-click the **Sine Wave** block to edit the signal's parameters, such as the sine wave's amplitude, frequency and phase. Ensure the `sine type` is set to `time based` and that the `Time (t)` value is set to `Use simulation time`. Next, set the frequency to 3 rad/s and click `Apply`, then `OK`.

Next, double-click the **Gain** block, and set the gain to 2. This outputs twice the input value at each time step. The same result could be obtained by changing the amplitude in the **Sine Wave** block's settings.

In the Simulink toolbar, change the `Simulation stop time` in the Simulink toolbar to $5s$. By default, in new Simulink files, the simulation time is set to $10s$.

Run Simulation

To run the simulation, click the green `Run` button in the Simulink toolbar, or press **Simulation** $>$ **Run**. The simulation may take a few seconds to complete. When the simulation is done, double-click the **Scope** block to see the simulation results in a new window. The results should resemble Fig. 6.3. In this case, the plot colour, line colour and line thickness were altered by double-clicking the **Scope** block, then clicking **View** $>$ **Style**.

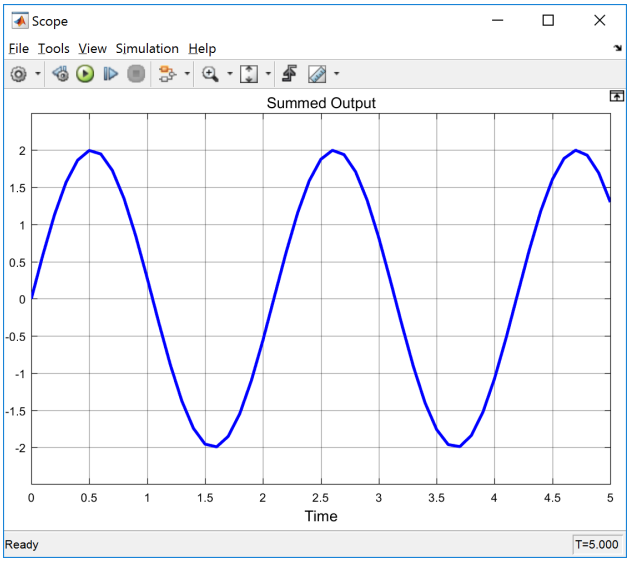


Figure 6.3: Simulink blocks in the model, now connected by wires.

6.4 Basic Example - Part II

In this section, we will add a constant bias to the system in Section 6.3 and cover a few additional commonly used blocks.

More Useful Blocks

Continuing with the `.slx` used in Part I, add the following blocks to your model.

- An **Add** block, found in **Simulink** > **Math Operations**. This outputs the sum of the two inputs.
- A **Constant** block, found in **Simulink** > **Sources**. Once it is in the model, set the constant to 2 in the block's settings.
- A **Mux** block, found in **Simulink** > **Signal Routing**. The **Mux** block combines inputs into vector outputs. This can be extremely useful when outputting data in the form of a column matrix, such as exporting all states of a given system at each time step. In this case, we will use it to output the two inputs on the same plot.
- Add one more **Scope** block to the model.

Wire Splitting

Arrange the blocks as shown in Fig. 6.4. There are multiple ways to connect one source to several sinks.

1. Click on the wire connected to the source. Hold `Ctrl` while clicking and dragging away from the wire. The signal will be split to both sinks.
2. Click on the > on the left side of the desired sink. Click and drag your mouse back to the source wire to connect the system.

Signal Labels

When dealing with more complicated systems, keeping track of each signal can be cumbersome. Signal labels are a useful tool to keep track of each input and output. To add a label to a signal, double-click on a wire and type your desired label. The label can be dragged anywhere adjacent to the wire. Label every signal, as shown in Fig. 6.4.

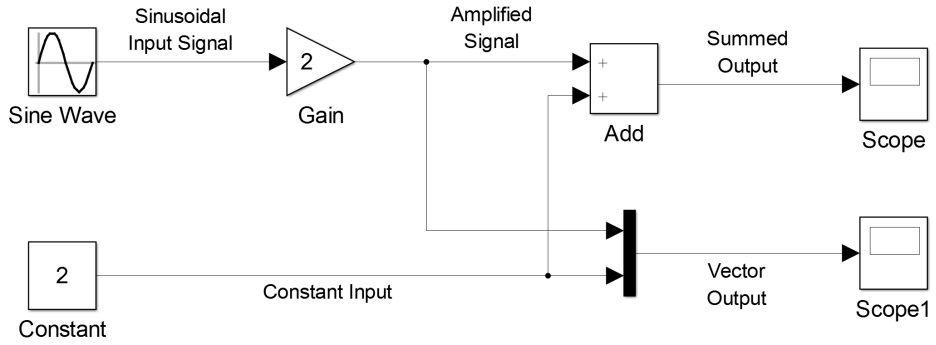
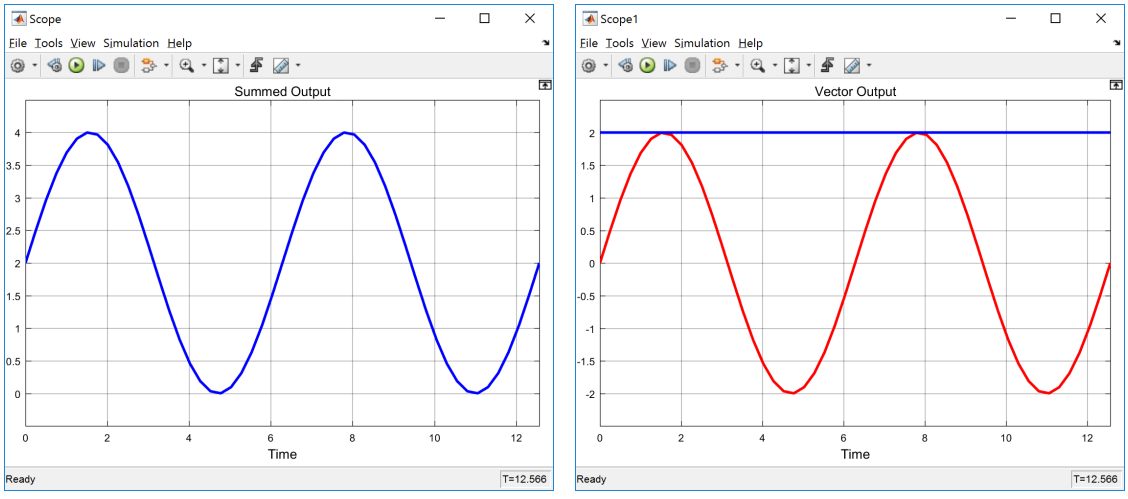


Figure 6.4: Simulink model in Basic Example - Part II.

Remark 6.1. Try to avoid overlapping wires. For simple models it may be possible to track overlapping wires, but once more complicated systems are involved, tracking a pile of tangled wires can be near impossible, even with proper labels.

Set the simulation time to `4*pi` s, change the sine wave's frequency to 1 rad/s and run the simulation. Your results should resemble Fig. 6.5(a) and Fig. 6.5(b).



(a) (b)

Figure 6.5: Plots of Basic Example - Part II, (a) the sum of both inputs, and (b), each individual input to the system.

6.5 Controlling Simulink through Matlab

6.5.1 Example: Simulation of Simple Pendulum (MECH 220: Dynamics)

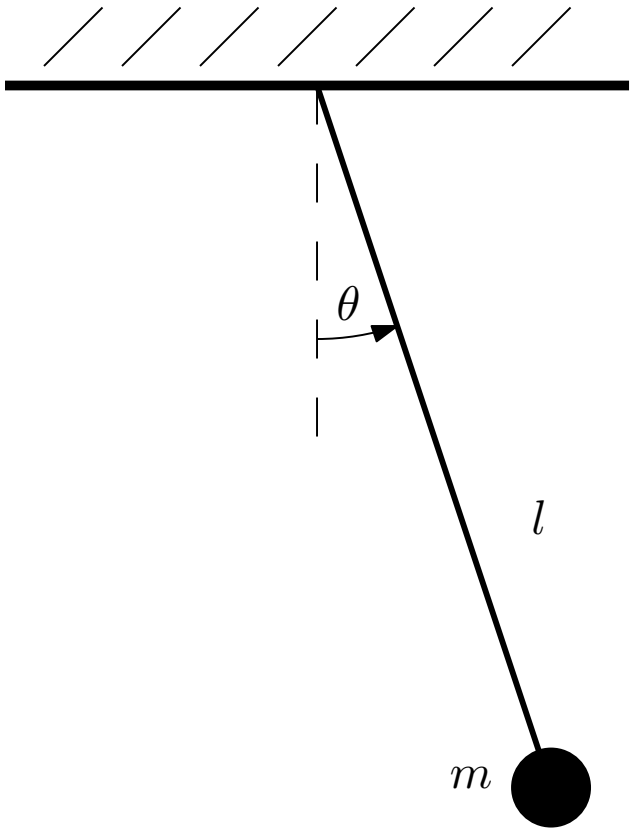


Figure 6.6: Simple pendulum.

Consider an unforced simple pendulum, as shown in Fig. 6.6. The beam has length l and negligible mass, while the tip has mass m and has a negligible diameter. The beam is released from rest from $\theta = 10°$. This system will be simulated by **Simulink** via `MATLAB`. This system can be described by the equation

$$I_m\ddot{\theta} = -m\ g\ l\ \sin(\theta). \tag{6.1}$$

Isolating for $\ddot{\theta}$, Equation 6.2 can be rearranged as

$$\ddot{\theta} = \frac{-m\ g\ l}{I_m}\sin(\theta). \tag{6.2}$$

First, create a new **Simulink** model, as shown in Fig. 6.7. The **Integrator** block is found in **Simulink** > **Continuous**. The **Sine Wave Function** block is found in **Simulink** > **Math Operations**. The **Clock** block is found in **Simulink** > **Sources**, and the **Out1** output block is found in **Simulink** > **Sinks**.

The gain block **Gain** is set to `-m*g*l/I_m`. `MATLAB` and **Simulink** both work in radians, so the block **Gain1** is `180/pi` in order to convert the output θ from radians to degrees.

Next, set the simulation time to `t_end`. Edit the Block Parameters of the second **Integrator** block, between $\dot{\theta}$ and θ, so that the initial condition is `IC`.

Edit the **Model Configuration Parameters** (**Simulation** > **Model Configuration Parameters**). Under

Data Import/Export, ensure that the format is set to **Array**.

Save the file as `pendulum_example.slx`. In the same folder, create a new script named `main.m`. In `main.m`, add the following script.

```
clear all    % This clears the workspace
close all    % This closes all open figures
clc          % This clears the command window

% You could alternatively create these constants in a separate
% script that would then be called in main.m
m = 10;      % mass, kg
l = 2;       % bar length, m
I_m = m*l^2; % moment of inertia of point mass, kg*m^2
g = 9.81;    % gravitational constant, m/s^2
IC = 10*pi/180; % Initial Conditions, converted to radians
t_end = 5;   % simulation run time, s

SimOut = sim('pendulum_example.slx','AbsTol','1e-6','RelTol','1e-6', ...
    'SaveState','on','StateSaveName','xoutNew','SaveOutput','on', ...
    'OutputSaveName','youtNew');
yout = SimOut.get('youtNew');

time  = yout(:,1);
theta = yout(:,2);
```

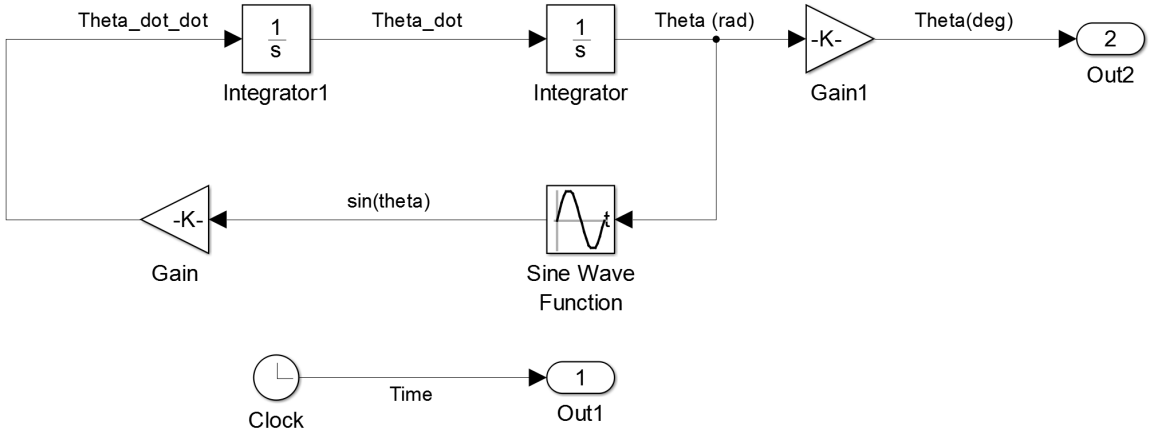


Figure 6.7: Simulink Model of nonlinear system

This script creates all necessary constants before beginning the **Simulink** simulation. The terms `AbsTol` and `RelTol` define the absolute and relative tolerance of the numerical simulation. In this case, `yout` returns a matrix in which each row represents one time step of the simulation, as determined by the **Clock** block, and each column is a different set of data from an **Out** block. In this example, the first column of `yout` is the time, while the second column is the value of θ at the corresponding time. If, for example, additional **Out** blocks were added to the `.slx` file for $\dot{\theta}$ and $\ddot{\theta}$, then `yout` would have more columns.

Run the script. Plot θ vs t. Your results should resemble Fig. 6.8.

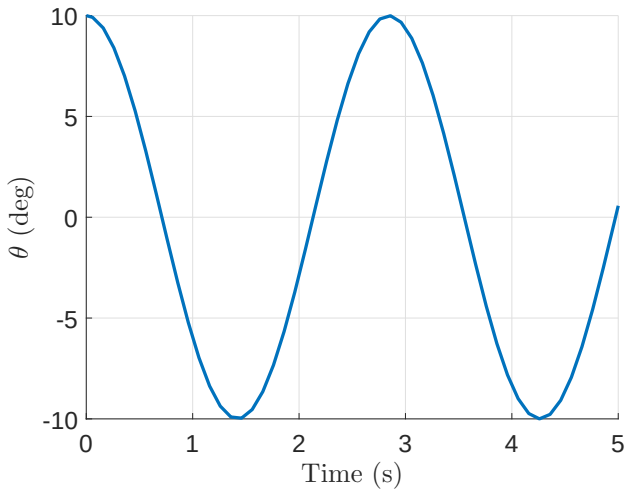


Figure 6.8: Plot of θ vs time for nonlinear pendulum example.

6.5.2 Example: Control of Inverted Pendulum (MECH 412: System Dynamics and Control)

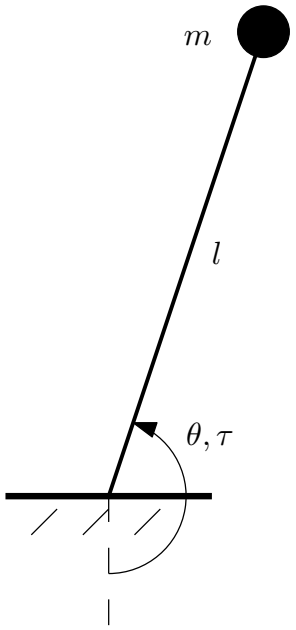


Figure 6.9: Simple inverted pendulum.

Consider an inverted pendulum, as shown in Fig. 6.9. The system has an encoder outputting θ, and a motor applying a torque τ at the base. Assume the link is rigid with negligible mass relative to the tip. This system is governed by the equation of motion,

$$\ddot{\theta} = -\frac{mgl}{I_m}\sin(\theta) + \frac{\tau}{I_m}. \tag{6.3}$$

In this case, we will use proportional-derivative (PD) control to set the link pointing straight up at $\theta = 180^\circ = \pi$ rad. Set up the system as shown in Fig. 6.10.

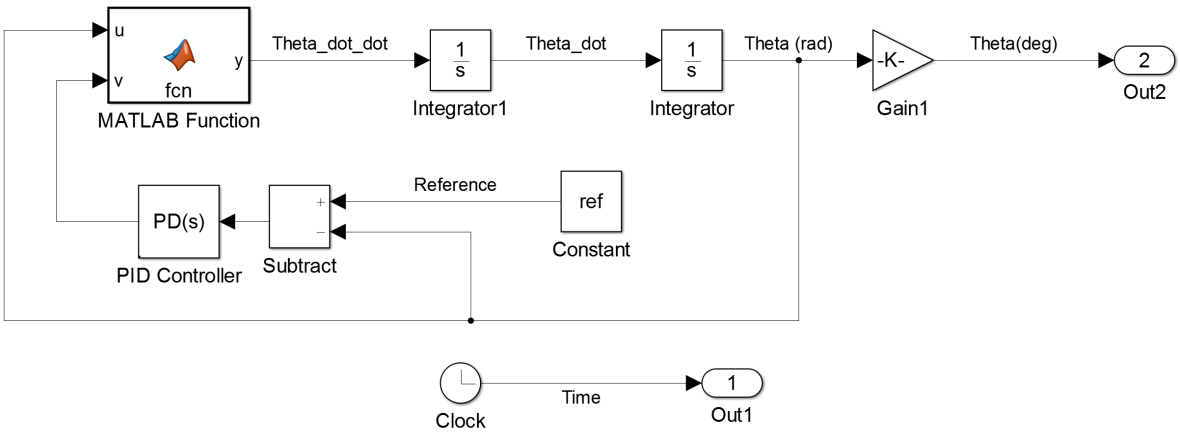


Figure 6.10: Simulink Model of inverted pendulum

Some new blocks in this system are the **MATLAB Function** block, found in **Simulink** > **User-Defined Functions**, and the **PID Controller**, found in **Simulink** > **Continuous**. Double-click the **MATLAB Function** block and enter the following text.

```
function y = fcn(u,v)
% This function receives theta and theta_dot, and
% outputs theta_dot_dot
theta = u;
tau = v;

%constants
m = 10; % mass, kg
l = 2; % bar length, m
I_m = m * l^2; % moment of inertia of point mass, kg * m2
g = 9.81; % gravitational constant, m/s2

theta_dot_dot = -m*g*l*sin(theta)/I_m + tau/I_m;

y = theta_dot_dot;
```

If you need to flip a block horizontally, right-click on the block, then select **Rotate & Flip** > **Flip Block** Set the **Constant** block to `pi` as the ideal beam angle, and keep the same initial conditions from Example 6.5.1. Next, double-click the **PID Control** block, and select PD control. Set `P = 250`, `D = 60`, and `N = 100`, then select `Apply` and `Ok`. Edit the **Model Configuration Parameters** (**Simulation** > **Model Configuration Parameters**). Under **Data Import/Export**, ensure that the format is set to **Array**. Save the Simulink file as `inverted_pendulum.slx`.

Next, create a new script titled `inv_pen.m`. Enter the following information.

```
clear all % This clears the workspace
close all % This closes all open figures
clc % This clears the command window

% You could alternatively create these constants in a separate
```

```
% script that would then be called in main.m

IC = 10 * pi/180; % Initial Conditions, converted to radians
ref = pi; % Desired angle theta, radians
t_end = 25; % simulation run time, s

SimOut = sim('pendulum_example.slx','AbsTol','1e-6','RelTol','1e-6', ...
'SaveState','on','StateSaveName','xoutNew','SaveOutput','on', ...
'OutputSaveName','youtNew');

yout = SimOut.get('youtNew');
time = yout(:,1);
theta = yout(:,2);
```

Click `Run` in the Simulink toolbar. Plot your results, which should resemble Fig. 6.11. Try adjusting the P and D values to shape the response.

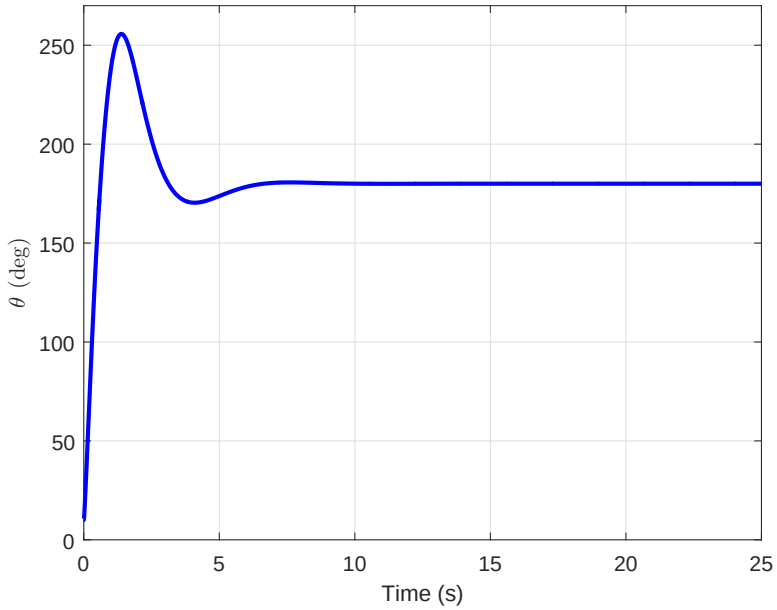


Figure 6.11: Tip angle versus time for inverted pendulum.

6.5.3 Example: Review of Section 3.3.1 Using Simulink (MECH 220: Dynamics)

Consider the example in Section 3.3.1, in which an object's flight is numerically integrated using `ode45`. Remake this example using Simulink. The following blocks may be useful.

- **Sign** (**Simulink** > **Math Operations**)
- **Integrator** (**Simulink** > **Continuous**)
- **Gain** (**Simulink** > **Math Operations**)

Index of examples