



TECHNOLOGICAL UNIVERSITY OF THE PHILIPPINES VISAYAS

Capt. Sabi St., City of Talisay, Negros Occidental

**College of Engineering
Office of the Program Coordinator**

LEARNING MODULE

MATH 322: NUMERICAL METHODS (Computer Methods in ECE)

DEPARTMENT: ELECTRONICS ENGINEERING

COMPILED BY:

ALEXANDER T. DIAMANTE JR.

2020

VISION

The Technological University of the Philippines shall be the premier state university with recognized excellence in engineering and technology at par with leading universities in the ASEAN region.

MISSION

The University shall provide higher and advanced vocational, technical, industrial, technological and professional education and training in industries and technology, and in practical arts leading to certificates, diplomas and degrees.

It shall provide progressive leadership in applied research, developmental studies in technical, industrial, and technological fields and production using indigenous materials; effect technology transfer in the countryside; and assist in the development of small-and-medium scale industries in identified growth center. (Reference: P.D. No. 1518, Section 2)

QUALITY POLICY

The Technological University of the Philippines shall commit to provide quality higher and advanced technological education; conduct relevant research and extension projects; continually improve its value to customers through enhancement of personnel competence and effective quality management system compliant to statutory and regulatory requirements; and adhere to its core values.

CORE VALUES

- T - Transparent and participatory governance
- U - Unity in the pursuit of TUP mission, goals, and objectives
- P - Professionalism in the discharge of quality service
- I - Integrity and commitment to maintain the good name of the University
- A - Accountability for individual and organizational quality performance
- N - Nationalism through tangible contribution to the rapid economic growth of the country
- S - Shared responsibility, hard work, and resourcefulness in compliance to the mandates of the university

TABLE OF CONTENTS

	<i>Page Numbers</i>
TOPIC/S.....	1
EXPECTED COMEPETENCIES	1
CONTENT/TECHNICAL INFORMATION	1
Introduction	1
Mathematical Models.....	2
Numerical Methods Included	7
PROGRESS CHECK.....	9
REFERENCES	10
TOPICS.....	11
EXPECTED COMEPETENCIES	11
CONTENT/TECHNICAL INFORMATION	11
Introduction to MATLAB	11
Built-in Functions.....	15
Graphics	17
PROGRESS CHECK.....	20
REFERENCES	21
TOPICS.....	22
EXPECTED COMEPETENCIES	22
TECHNICAL INFORMATION / CONTENT	22
Script Files.....	22
Function Files.....	25
Input and Output.....	28
Structured Programming	29
Loops.....	30
Anonymous Functions.....	32
Functions in a Function	32
PROGRESS CHECK.....	32
REFERENCES	34
TOPICS.....	35
EXPECTED COMEPETENCIES	35
TECHNICAL INFORMATION / CONTENT	35
Accuracy and Precision.....	35
Roundoff errors	38

Truncation Errors	39
Total Numerical Error	45
PROGRESS CHECK.....	46
REFERENCES	47

COURSE DESCRIPTION

Elementary numerical analysis: roots of equations, systems of linear algebraic equations, curve fitting, integration, and solution of ordinary differential equations. Numerical techniques are presented in the context of engineering applications, and example problems are solved using a variety of computer-based tools (structure programming, spreadsheet, and computational/symbolic processing software packages).

COURSE OUTCOMES

1. Define and identify special types of matrices, perform basic matrix operations, perform Gaussian elimination to solve a linear system, and use LU decomposition to find the inverse of a matrix.
2. Describe situations in which numerical solutions to nonlinear equations are needed, implement the bisection method for solving equations, list advantages and disadvantages of the bisection and interpolation methods, implement Newton-Raphson, and describe the difference.
3. Define and distinguish between ordinary and partial differential equations, implement Euler's methods for solving ordinary differential equations, investigate how step size affects accuracy in Euler's method, and implement and use the Runge-Kutta 4th order method for solving ordinary differential equations.
4. Integrate all the methods into a form of program using MATLAB.

GRADING SYSTEM

Mid-term grade: (weekly assessment from wk 1 to 6) X 0.70 + (MTE x .0.30)

End-term grade: (weekly assessment from wk8 to 13) X 0.70 + (ETE x .0.30)

Grade for the subject: (midterm grade + end term grade) / 2

The passing grade for this course is 5.0

LEARNING COMPETENCE RUBRIC			
CRITERIA / PERFORMANCE	LEVEL 3 8-10	LEVEL 2 5-7	LEVEL 1 0-4
Concept	Appropriate concepts that are fully understood, clearly stated and employed correctly.	Appropriate concepts that are mostly understood but employed with errors.	Little or no understanding of the concepts
Strategic Approach	The approach chosen is appropriate, clearly shown, clearly stated and all elements are valid.	Valid approach with errors that impede understanding.	Little or no understanding of how to approach the problem.
Solutions	The problem is solved accurately in terms of mathematical manipulation and numerical calculation.	The solution contains some minor math or numerical errors	Major error in solving the problem.

CRITERIA	LEVEL 4 9 – 10	LEVEL 3 7 – 8	LEVEL 2 5 – 6	LEVEL 1 0 – 4
Delivery	<ul style="list-style-type: none"> Completed between 90-100% of the requirements. Delivered on time, and in correct format (disk, email, etc.) 	<ul style="list-style-type: none"> Completed between 80-90% of the requirements. Delivered on time, and in correct format (disk, email, etc.) 	<ul style="list-style-type: none"> Completed between 70-80% of the requirements. Delivered on time, and in correct format (disk, email, etc.) 	<ul style="list-style-type: none"> Completed less than 70% of the requirements. Not delivered on time or not in correct format (disk, email, etc.)
Coding Standards	<ul style="list-style-type: none"> Includes name, date, and assignment title. Excellent use of white space. 	<ul style="list-style-type: none"> Includes name, date, and assignment title. Good use of white space. Organized work. 	<ul style="list-style-type: none"> Includes name, date, and assignment title. White space makes program fairly easy to read. 	<ul style="list-style-type: none"> No name, date, or assignment title included Poor use of white space (indentation, blank lines). Disorganized and messy

	<ul style="list-style-type: none"> • Creatively organized work. 		<ul style="list-style-type: none"> • Organized work. 	
Efficiency	<ul style="list-style-type: none"> • Solution is efficient, easy to understand, and maintain. 	<ul style="list-style-type: none"> • Solution is efficient and easy to follow (i.e. no confusing tricks). 	<ul style="list-style-type: none"> • A logical solution that is easy to follow but it is not the most efficient. 	<ul style="list-style-type: none"> • A difficult and inefficient solution.

LEARNING GUIDE

Week No.: 1

TOPIC/S

1. Introduction to Numerical Methods and Computer Methods
2. Mathematical Models
3. Methods used in this Module

EXPECTED COMEPETENCIES

Upon completing this Learning Module, you will be able to:

1. Explain how numerical methods provides a means for solutions to be implemented on a digital computer.
2. Explain how mathematical models can be formulated from scientific principles.
3. Define the different numerical methods to be used in this module.

CONTENT/TECHNICAL INFORMATION

Introduction

Numerical Methods are techniques by which mathematical problems are formulated so that they can be solved with arithmetic and logical operations. Because digital computers excel at performing such operations, numerical methods are sometimes referred to as **Computer Mathematics** (Chapra, 2015).

Here are some reasons why we need to study Numerical Methods:

1. Numerical methods greatly expand the types of problems you can address. They are capable of handling large systems of equations, nonlinearities, and complicated geometries that are not uncommon in engineering and science and that are often impossible to solve analytically with standard calculus. They greatly enhance your problem-solving skills.
2. Numerical methods allow you to use related software with insight. The intelligent use of these programs is greatly enhanced by an understanding of the basic theory underlying the methods.
3. Many problems cannot be approached using this related software alone, if you are conversant with numerical methods, and adept at computer programming, you can design your own program to solve these problems.
4. Numerical methods provide a vehicle for learning to use computers. When you successfully implement numerical methods on a computer, and then apply them to solve otherwise intractable problems, you will be provided with a dramatical demonstration of how computers can serve your professional development.

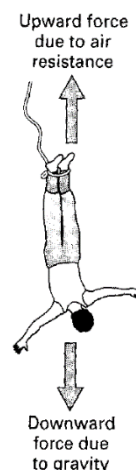
5. Numerical methods provide a vehicle for you to reinforce your understanding of mathematics. Because one function of numerical methods is to reduce higher mathematics to get at the "nuts and bolts" of some otherwise obscure topics. Enhanced understanding and insight can result from this alternative perspective.

Mathematical Models

Suppose that a bungee-jumping company hires you. You are given the task of predicting the velocity of a jumper as a function of time during the free-fall part of the jump. This information will be used as part of a larger analysis to determine the length and required strength of the bungee cord for jumpers of different mass.

You know from your studies of physics that the acceleration should be equal to the ratio of the force to the mass (Newton's second law). Based on this insight and your knowledge of fluid mechanics, you develop the following mathematical model for the rate of change of velocity with respect to time,

$$\frac{dv}{dt} = g - \frac{Cd}{m}v^2$$



Where:

m = jumper's mass (Kg)

v = vertical velocity (m/s)

t = time (s)

Cd = 2nd order drag coefficient (Kg/m)

g = acceleration due to gravity (9.81 m/s²)

Because this is a differential equation, you know that calculus might be used to obtain an analytical or exact solution for v as a function of t . But we will be answering this while illustrating an alternative solution.

A mathematical model can be broadly defined as a formulation or equation that expresses the essential features of a physical system or process in mathematical terms. In a general sense, it can be represented as a functional relationship of the form

$$\text{Dependent Variable} = f \left(\begin{array}{c} \text{independent variables,} \\ \text{parameters,} \\ \text{forcing functions} \end{array} \right)$$

- The **dependent variable** is a characteristic that usually reflects the behavior or state of the system.
- The **independent variables** are usually dimensions, such as time and space, along which the system's behavior is being determined.
- The **parameters** are reflective of the system's properties or composition.
- The **forcing functions** are external influences acting upon the system

The actual mathematical expression can range from a simple algebraic relationship to large complicated sets of differential equations. For example, on the basis of his observations, Newton formulated his second law of motion, which states that the time rate of change of momentum of a body is equal to the resultant force acting on it. Its mathematical model is,

$$F = ma$$

Where:

F = net force acting on the body (N, or Kg m/s²) m = mass of object (Kg)

a = acceleration (m/s²)

The second law can be recast in the format for our first equation by dividing both sides by m to give

$$a = \frac{F}{m}$$

- **a** is the dependent variable reflecting the system's behavior
- **F** is the forcing function
- **m** is a parameter

In this simple case, there is no independent variable because we are not yet predicting how acceleration varies in time or space.

Some characteristics that are typical of mathematical models of the physical world:

- It describes a natural process or system in mathematical terms.
- It represents an idealization and simplification of reality. That is, the model ignores negligible details of the natural process and focuses on its essential manifestations. Thus, the second law does not include both the effects of relativity that are of minimal importance when applied to objects and forces that interact on or about the earth surface at velocities and on scales visible to humans.
- It yields reproducible results and, consequently, can be used for predictive purposes. For example, if the force on an object and its mass are known, we can easily compute for the acceleration.

Now, let us expand on the 2nd law of motion to determine the terminal velocity of a free-falling body near the earth's surface (the bungee jumper). A model can be derived by expressing the acceleration as the time rate of change of the velocity.

$$\frac{dv}{dt} = \frac{F}{m}$$

Thus, the rate of change of the velocity is equal to the net force acting on the body normalized to its mass. If the net force is positive, the object will accelerate, decelerate if negative. If the net force is zero, the velocity will remain at a constant level.

We will now express the net force in terms of measurable variables and parameters. For a body falling within the vicinity of the earth, the net force is composed of two opposing forces: the downward pull of gravity F_D and the upward force of air resistance F_U :

$$F = F_D + F_U$$

If the force in the downward direction is assigned a positive sign, the second law can be used to formulate the force due to gravity as

$$F_D = mg$$

Air resistance can be formulated in a variety of ways. A good first approximation would be to assume that it is proportional to the square of the velocity.

$$F_U = -C_d v^2$$

C_d is a proportionality constant called the drag coefficient. Thus, the greater the fall velocity, the greater the upward force due to air resistance. This parameter accounts for properties of the falling object, such as shape or surface roughness, that affect air resistance.

The net force is the difference between the downward and upward force. Therefore

$$\frac{dv}{dt} = g - \frac{C_d}{m} v^2$$

This now is a model that relates acceleration of a falling object to the forces acting on it. It is a differential equation because it is written in terms of the differential rate of change of the variable that we are interested in predicting. The exact solution for the velocity of the jumper cannot be obtained using simple algebraic manipulation. More advanced techniques such as those of calculus must be applied to obtain an exact or analytical solution.

For example, if the jumper is initially at rest ($v = 0$ at $t = 0$). Calculus can be used to solve:

$$v(t) = \sqrt{\frac{gm}{C_d}} \tanh\left(\sqrt{\frac{gC_d}{m}} t\right)$$

where \tanh is the hyperbolic tangent that can be either computed directly or via more elementary exponential function as in

$$\tanh x = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

Analytical Solution

Problem. A bungee jumper with a mass of 68.1 Kg leaps from a stationary hot air balloon. Compute the velocity for the first 12 s of free fall. Also determine the terminal velocity that will be attained for an infinitely long cord. Use a drag coefficient of 0.25 Kg/m.

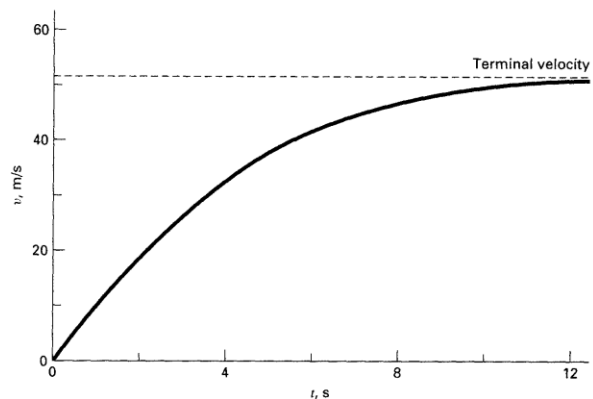
Solution.

$$v(t) = \sqrt{\frac{9.81(68.1)}{0.25}} \tanh\left(\sqrt{\frac{9.81(0.25)}{68.1}} t\right) = 51.6938 \tanh(0.18977t)$$

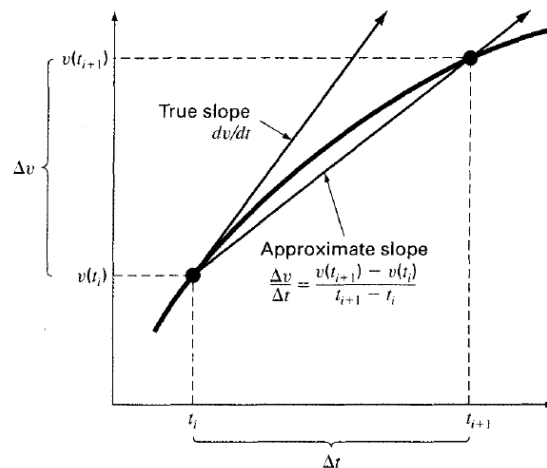
which yields

t, s	V, m/s
0	0
2	18.7292
4	33.1118
6	42.0762
8	46.9575
10	49.4214
12	50.6175
∞	51.6938

According to the model, the jumper accelerates rapidly. A velocity of 49.4214 is attained after 10 s. After a sufficiently long time, a constant velocity (terminal velocity) of 51.6938 is reached. This velocity is constant because eventually, the force of gravity will be in balance with the air resistance.



Numerical methods are those in which the mathematical problem is reformulated so it can be solved by arithmetic operations. This can be illustrated by realizing that the time rate of change of velocity can be approximated by:



$$\frac{dv}{dt} \cong \frac{\Delta v}{\Delta t} = \frac{v(t_{i+1}) - v(t_i)}{t_{i+1} - t_i}$$

This equation is called the **finite-difference approximation** of the derivative at time t_i . Note that the congruence is approximate because they are finite. Remember from calculus that

$$\frac{dv}{dt} = \lim_{\Delta t \rightarrow 0} \frac{\Delta v}{\Delta t}$$

So, substituting it with our analytical solution equation

$$\frac{v(t_{i+1}) - v(t_i)}{t_{i+1} - t_i} = g - \frac{C_d}{m} v(t_i)^2$$

And then can re-arranged to yield

$$v(t_{i+1}) = v(t_i) + \left[g - \frac{C_d}{m} v(t_i)^2 \right] (t_{i+1} - t_i)$$

This can also be re-written as

$$v_{i+1} = v_i + \frac{dv_i}{dt} \Delta t$$

We can now see that the differential equation has been transformed into an equation that can be used to determine the velocity algebraically using the slope and previous values of **v** and **t**. If you are given an initial value for velocity at some time, you can easily compute velocity at a later time. This new value of velocity can in turn be employed to extend the computation to any time along the way.

$$\text{New value} = \text{old value} + \text{slope} * \text{step size}$$

This approach is formally called the **Euler's method**. But more of this will be discussed when we get to the differential equations part of the module.

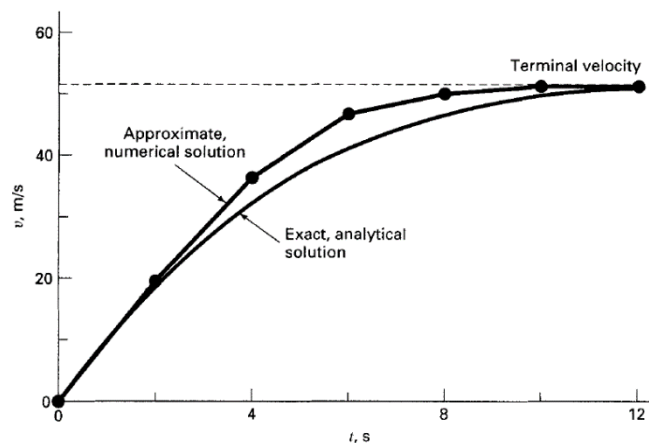
Numerical Solution

At the start of the computation $t_0 = 0$, the velocity of the jumper is zero. Using the information and the parameter values, we can compute velocity at $t_1 = 2$ s:

$$v = 0 + \left[9.81 - \frac{0.25}{68.1} (0)^2 \right] * 2 = 19.62 \text{ m/s}$$

For the next interval (from $t = 2$ to 4 s), the computation is repeated, with the result

$$v = 0 + \left[9.81 - \frac{0.25}{68.1} (19.62)^2 \right] * 2 = 36.4137 \text{ m/s}$$



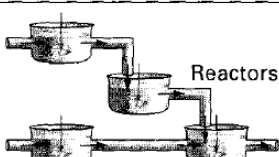

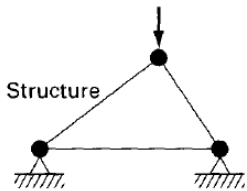
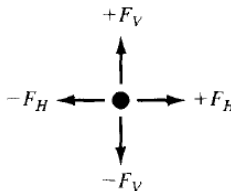

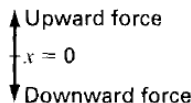
To have a better view of the results, refer to the table below:

t, s	V, m/s
0	0
2	19.6200
4	36.4137
6	46.2983
8	50.1802
10	51.3123
12	51.6008
∞	51.6938

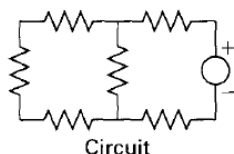
As you can see, the numerical solution can be used to solve complex problems using elementary means. But since we are using straight-line segments to approximate a continuously curving function, there will be discrepancies. One way to minimize this is to use smaller step sizes. Like instead of 2 s, we could use 1 s or 0.5 s, and so on. Thus, you can accurately model the velocity of the jumper without having to solve the differential equation exactly.

Numerical Methods Included

Euler's method is typical of many classes of numerical methods. In essence, most consist of recasting mathematical operations into the simple kind of algebraic and logical operations compatible with digital computers. Images taken from Chapra, S. (2014). Numerical Methods for Engineers (7th ed.). McGraw-Hill Education.

Field	Device	Organizing Principle	Mathematical Expression
Chemical engineering		Conservation of mass	Mass balance:  Over a unit of time period $\Delta \text{mass} = \text{inputs} - \text{outputs}$
Civil engineering		Conservation of momentum	Force balance:  At each node $\Sigma \text{horizontal forces } (F_H) = 0$ $\Sigma \text{vertical forces } (F_V) = 0$
Mechanical engineering		Conservation of momentum	Force balance:  $m \frac{d^2x}{dt^2} = \text{downward force} - \text{upward force}$

Electrical
engineering

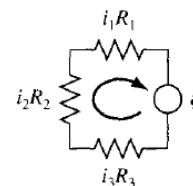


Conservation
of charge

Current balance: $+i_1 \rightarrow -i_3$
For each node
 $\Sigma \text{ current } (i) = 0$

Conservation
of energy

Voltage balance:



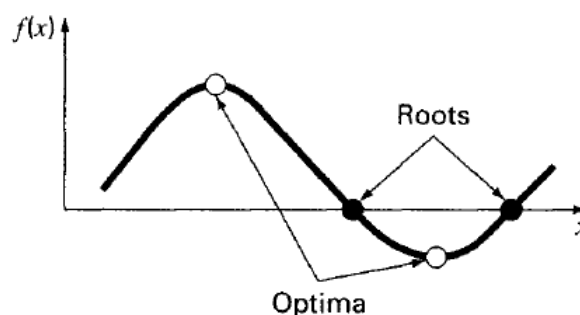
Around each loop
 $\Sigma \text{ emf's} - \Sigma \text{ voltage drops for resistors}$
 $= 0$
 $\Sigma \xi - \Sigma iR = 0$

Here are some numerical methods to be discussed in the module:

Roots and Optimization

Roots: Solve for x so that $f(x) = 0$

Optimization: Solve for x so that $f'(x) = 0$

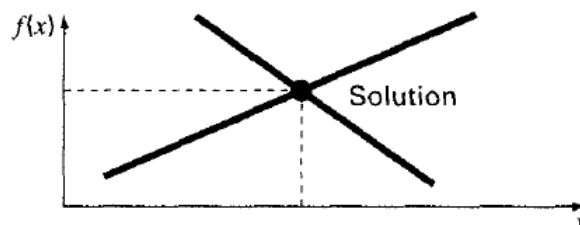


Linear Algebraic Equations

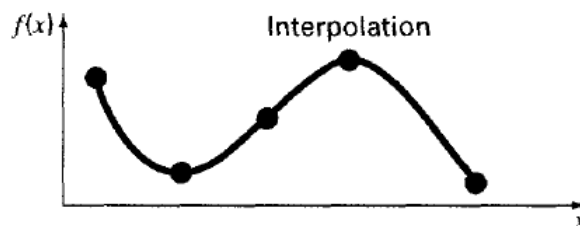
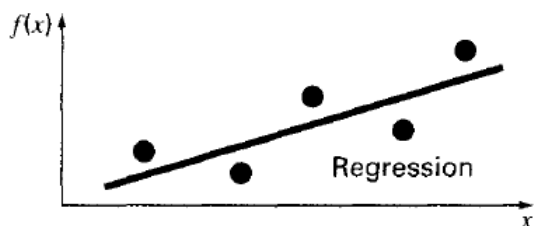
Given the a 's and the b 's, solve for the x 's

$$a_{11}x_1 + a_{12}x_2 = b_1$$

$$a_{21}x_1 + a_{22}x_2 = b_2$$



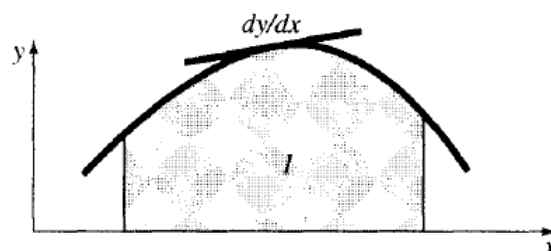
Curve Fitting



Integration and Differentiation

Integration: Find the area under the curve

Differentiation: Find the slope of the curve



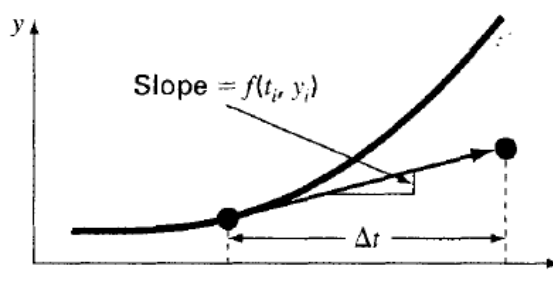
Differential Equations

Given

$$\frac{dy}{dt} \approx \frac{\Delta y}{\Delta t} = f(t, y)$$

solve for y as a function of t

$$y_{i+1} = y_i + f(t_i, y_i)\Delta t$$



PROGRESS CHECK

1. Describe the advantages and/or disadvantages (if any) of employing numerical methods in solving complex mathematical problems.

2. For a free-falling bungee jumper with linear drag, assume a first jumper is 70 Kg and has a drag coefficient of 12 Kg/s. If a second jumper has a drag coefficient of 15 Kg/s and a mass of 75 Kg, how long will it take her to reach the same velocity jumper 1 reached in 10 s? Use Euler's method with step size of 1 s.

3. A storage tank contains a liquid at depth y where $y = 0$ when the tank is half full. Liquid is withdrawn at a constant flow rate Q to meet demands. The contents are resupplied at a sinusoidal rate $3Q \sin^2(t)$, This can be written as

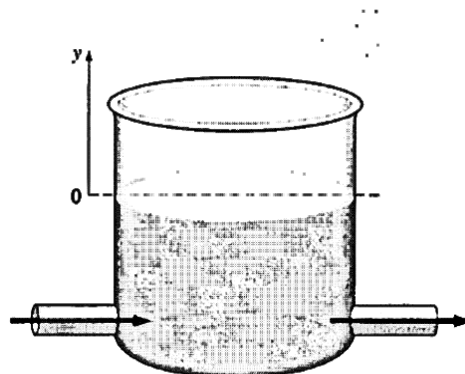
$$\frac{d(Ay)}{dt} = 3Q \sin^2(t) - Q$$

or since the surface area A is constant

$$\left(\frac{\text{change in}}{\text{volume}} \right) = (\text{inflow}) - (\text{outflow})$$

$$\frac{dy}{dt} = 3 \frac{Q}{A} \sin^2(t) - \frac{Q}{A}$$

Use Euler's method to solve for depth y from $t = 0$ to 10 d with step size of 1 d. The parameter values are $A = 1200 \text{ m}^2$ and $Q = 500 \text{ m}^3/\text{d}$. Assume the initial condition is $y = 0$.



REFERENCES

Textbook/s :

Chapra, S. C. (2015). Applied Numerical Methods with MATLAB for Engineers and Scientists. New York: McGraw-Hill.

LEARNING GUIDE

Week No.: 2

TOPICS

1. Introduction to MATLAB
2. Built-In Functions
3. Graphics

EXPECTED COMEPETENCIES

Upon completing this Learning Module, you will be able to:

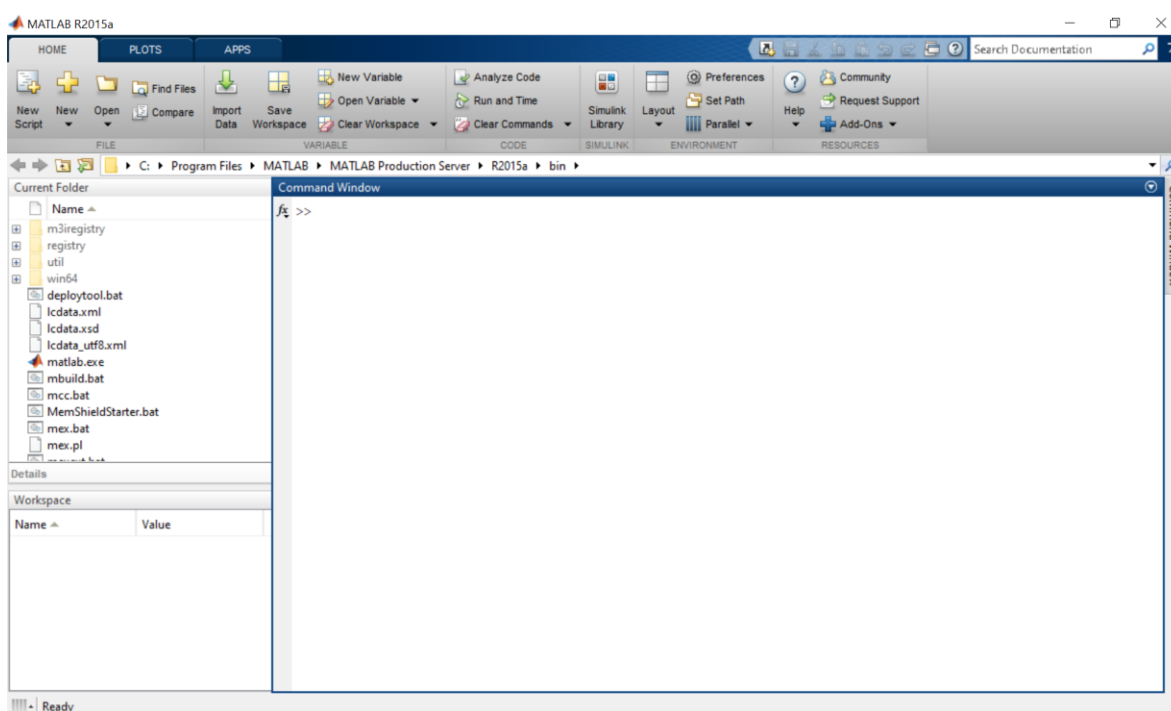
1. Familiarize with the MATLAB environment.
2. Apply built-in functions to perform operations and solve problems.
3. Utilize MATLAB tools to create graphs and other representations.

CONTENT/TECHNICAL INFORMATION

Introduction to MATLAB

MATLAB is a computer program that provides the user with a convenient environment for performing many types of calculations. It is an excellent tool to implement numerical methods.

The most common way to operate MATLAB is by entering commands one at a time in the command window. As an introduction, we will be using MATLAB as an advanced calculator to show us the common operations like performing calculations and creating plots.



Shown in the image is the typical environment of MATLAB. A big part of it is the Command Window. This is where we enter the commands and data. You can also see a ribbon menu on top just like in MS Office applications.

The calculator mode of MATLAB operates in a sequential fashion as you type in commands line by line. For each command you get a result. Notice that MATLAB has automatically assigned the answer to a variable, *ans*. Thus, you could now use *ans* in a subsequent calculation.

MATLAB assigns the result to *ans* whenever you do not explicitly assign the calculation to a variable of your own choosing.

Assignment

Assignment refers to assigning values to variable names. This results in the storage of the values in the memory location corresponding to the variable name.

The assignment of values to scalar variables is like other computer languages. The assignment echo prints to confirm what you have done. Echo printing is a characteristic of MATLAB. It can be suppressed by terminating the command line with a semicolon (;) character.

You can type several commands on the same line by separating them with commas or semicolons. If you separate them with commas, they will be displayed, and if you use semicolon, they will not.

MATLAB treats names in a case-sensitive manner. The upper and lower case of a letter is not the same.

We can assign complex values to variables, since MATLAB handles complex arithmetic automatically. The unit imaginary number $\sqrt{-1}$ is preassigned to the variable *i*. A complex value can also be assigned.

There are also several predefined variables, like pi (π). Notice that MATLAB displays four decimal places at first. If you desire additional precision, you may use the code `format long`. When pi is again entered, the results now is displayed to 15 significant figures. To return to the four decimal version, you may use the code `format short`.

Here is a summary of the format commands you will employ routinely in engineering and scientific calculations

```
Command Window
>> 45*68
ans =
    3060
>> ans - 60
ans =
    3000
>> ans * ans
ans =
   9000000
fx >> |
```

```
Command Window
>> a = 7
a =
    7
>> A = 14;
>> a
a =
    7
>> A
A =
   14
fx >> |
```

```
Command Window
>> i
ans =
    0.0000 + 1.0000i
>> 4 + i
ans =
    4.0000 + 1.0000i
>> pi
ans =
    3.1416
>> format long
>> pi
ans =
    3.141592653589793
```

type	Result	Example
short	Scaled fixed-point format with 5 digits	3.1416
long	Scaled fixed-point format with 15 digits for double and 7 digits for single	3.14159265358979
short e	Floating-point format with 5 digits	3.1416e+000
long e	Floating-point format with 15 digits for double and 7 digits for single	3.141592653589793e+000
short g	Best of fixed- or floating-point format with 5 digits	3.1416
long g	Best of fixed- or floating-point format with 15 digits for double and 7 digits for single	3.14159265358979
short eng	Engineering format with at least 5 digits and a power that is a multiple of 3	3.1416e+000
long eng	Engineering format with at least 16 digits and a power that is a multiple of 3	3.141592653589793e+000
bank	Fixed dollars and cents	3.14

Arrays, Vectors and Matrices

An array is a collection of values that are represented by a single variable name. One-dimensional arrays are called vectors and two-dimensional arrays are called matrices. Brackets are used to enter arrays in the command mode.

In practice, row vectors are rarely used to solve mathematical problems. When we speak of vectors, we usually refer to column vectors, which are more commonly used. A column vector can be entered in several ways. You can use the semicolon (;) to separate between rows of a matrix. You may also use the enter key (carriage return).

At any point in a session, a list of all current variables can be obtained by entering the `who` or `whos` command.

```

Command Window
>> who

Your variables are:

A      B      C      D      a      ans

>> whos

      Name      Size      Bytes  Class  Attributes
      ----      -
A      1x5      40    double
B      5x1      40    double
C      3x3      72    double
D      3x3      72    double
a      1x1       8     double
ans    1x1       8     double

```

```

Command Window
>> A = [1 2 3 4 5]

A =

     1     2     3     4     5

>> B = [1;2;3;4;5]

B =

     1
     2
     3
     4
     5

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

C =

     1     2     3
     4     5     6
     7     8     9

>> D = [1 2 3
4 5 6
7 8 9]

D =

     1     2     3
     4     5     6
     7     8     9

```

The subscript notation can be used to access an individual element of an array. For an array, $A(m, n)$ selects the element in the m th row and the n th column.

There are several built-in functions that can be used to create matrices. For example, the `zeros` and `ones` functions create vector or matrices filled with ones and zeros, respectively. Both have two arguments, the first for the number of rows and the second for the number of columns.

The Colon Operator

The colon operator is a powerful tool for creating and manipulating arrays. If a colon is used to separate two numbers, MATLAB generates the numbers between them using an increment of one. If colons are used to separate three numbers, MATLAB generates the numbers between the first and third numbers using an increment equal to the second number.

```

Command Window

>> G = 1:7

G =

     1     2     3     4     5     6     7

>> H = -1:0.5:1

H =

-1.0000   -0.5000         0    0.5000    1.0000

>> C(2,:)

ans =

     4     5     6

>> C(7:9)

ans =

     3     6     9

fx >> |

```

```

Command Window

>> C

C =

     1     2     3
     4     5     6
     7     8     9

>> C(2,2)

ans =

     5

>> E = zeros(3,2)

E =

     0     0
     0     0
     0     0

>> F = ones(2,2)

F =

     1     1
     1     1

fx >> |

```

Aside from creating series of numbers, the colon can also be used as a wild card to select the individual rows and columns of a matrix. When a colon is used in place of a specific subscript, the colon represents the entire row or column.

We can also use the colon notation to selectively extract a series of elements from within an array.

Mathematical Operations

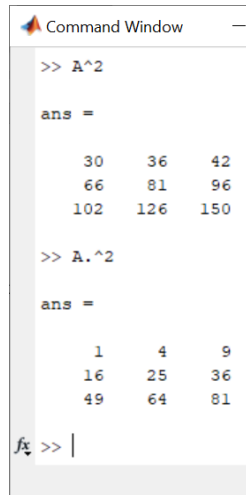
Operations with scalar quantities are handles in a straightforward manner, like other computer languages. The common operators, in order of priority are

\wedge	Exponentiation
$-$	Negation
$*$ /	Multiplication and Division
\backslash	Left Division
$+$ -	Addition and Subtraction

The real power of MATLAB is in its ability to carry out vector-matrix calculations. It will apply the simple arithmetic operators in vector-matrix fashion if possible. At times, you will want to carry out calculations item by item in a matrix or vector. MATLAB provides for that too.

A^2 will result in a matrix multiplication of A with itself. What if you want to just square each element of A ?

That can be done by putting a dot (.) before the operator (in this case \wedge) you want to carry out an element-by-element operation.

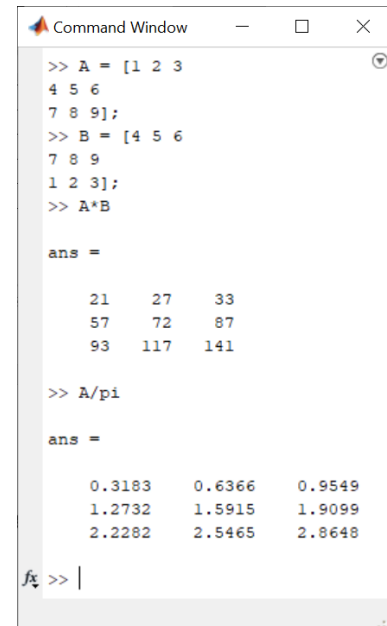


```

>> A^2
ans =
    30    36    42
    66    81    96
   102   126   150

>> A.^2
ans =
     1     4     9
    16    25    36
    49    64    81
fx >> |

```

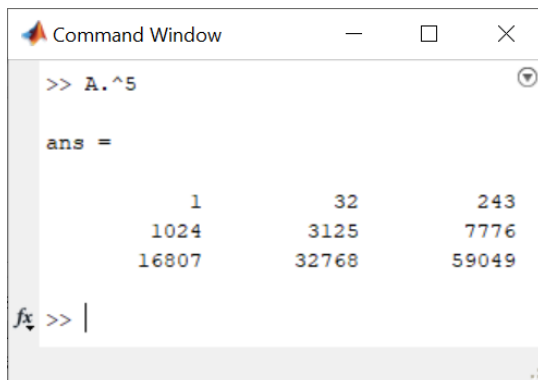


```

>> A = [1 2 3
        4 5 6
        7 8 9];
>> B = [4 5 6
        7 8 9
        1 2 3];
>> A*B
ans =
    21    27    33
    57    72    87
    93   117   141

>> A/pi
ans =
    0.3183    0.6366    0.9549
    1.2732    1.5915    1.9099
    2.2282    2.5465    2.8648
fx >> |

```



```

>> A.^5
ans =
     1    32   243
   1024   3125   7776
  16807   32768   59049
fx >> |

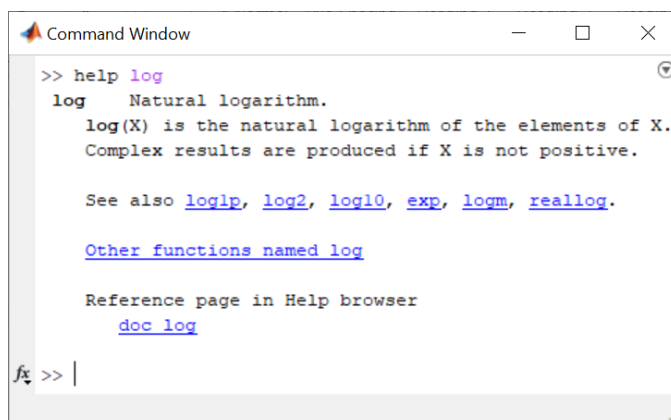
```

MATLAB contains a helpful shortcut for performing calculations that you've already done. By pressing the "up-arrow key", you should get back the most recent lines you typed in, and you will also be able to edit this line. In the example on the left, the $A.^2$ was edited to make it $A.^5$.

You can also type "A" and press the up-arrow once and it will automatically bring up the last command beginning with the letter "A".

Built-in Functions

MATLAB and its Toolboxes (add-ons) have a rich collection of built-in functions. You can use online help to find out more about them. Say you want to know about the `log` function, you just need to type in `help log`.



```

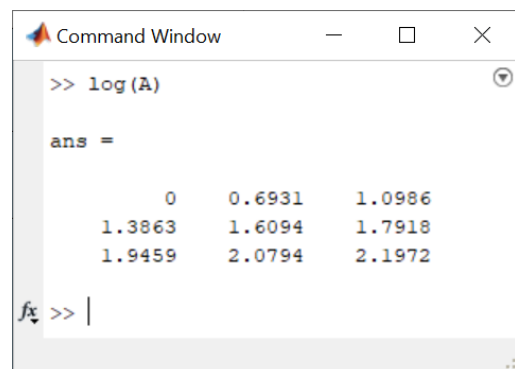
>> help log
log    Natural logarithm.
log(X) is the natural logarithm of the elements of X.
Complex results are produced if X is not positive.

See also loglp, log2, log10, exp, logm, reallog.

Other functions named log

Reference page in Help browser
doc log
fx >> |

```



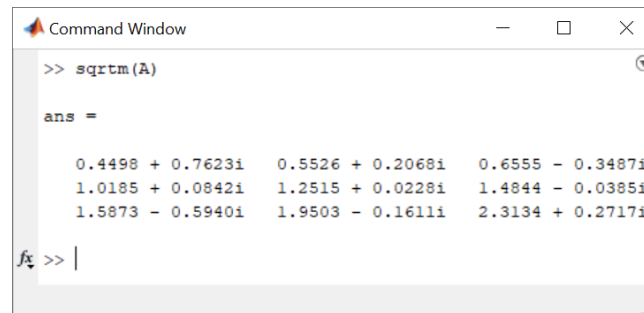
```

>> log(A)
ans =
     0    0.6931    1.0986
   1.3863    1.6094    1.7918
   1.9459    2.0794    2.1972
fx >> |

```

One of the important properties of MATLAB's built-in function is that they will operate directly on vector and matrix quantities. You saw that the natural logarithm function is applied in array style, element-by-element, to the matrix A.

Most functions such as **sqrt**, **abs**, **sin**, **acos**, **tanh**, and **exp**, operate in array fashion. Certain functions, such as exponential and square root, have matrix definitions also. MATLAB will evaluate the matrix version when the letter *m* is appended to the function name.



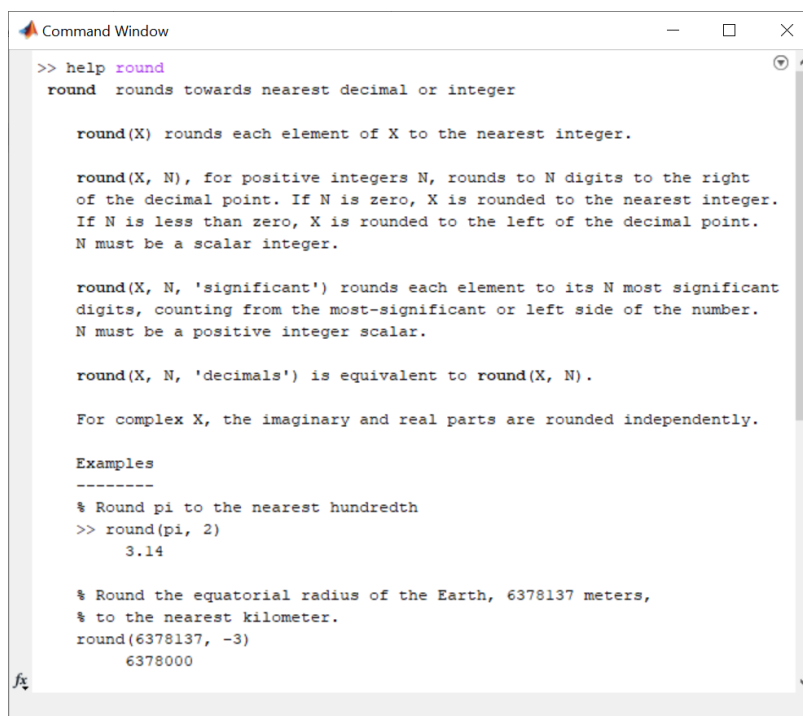
```

>> sqrtm(A)

ans =

    0.4498 + 0.7623i    0.5526 + 0.2068i    0.6555 - 0.3487i
    1.0185 + 0.0842i    1.2515 + 0.0228i    1.4844 - 0.0385i
    1.5873 - 0.5940i    1.9503 - 0.1611i    2.3134 + 0.2717i
  
```

There are several functions for rounding.



```

>> help round
round rounds towards nearest decimal or integer

round(X) rounds each element of X to the nearest integer.

round(X, N), for positive integers N, rounds to N digits to the right
of the decimal point. If N is zero, X is rounded to the nearest integer.
If N is less than zero, X is rounded to the left of the decimal point.
N must be a scalar integer.

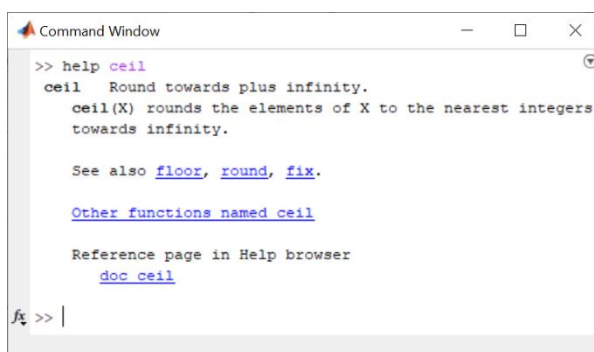
round(X, N, 'significant') rounds each element to its N most significant
digits, counting from the most-significant or left side of the number.
N must be a positive integer scalar.

round(X, N, 'decimals') is equivalent to round(X, N).

For complex X, the imaginary and real parts are rounded independently.

Examples
-----
% Round pi to the nearest hundredth
>> round(pi, 2)
    3.14

% Round the equatorial radius of the Earth, 6378137 meters,
% to the nearest kilometer.
round(6378137, -3)
    6378000
  
```



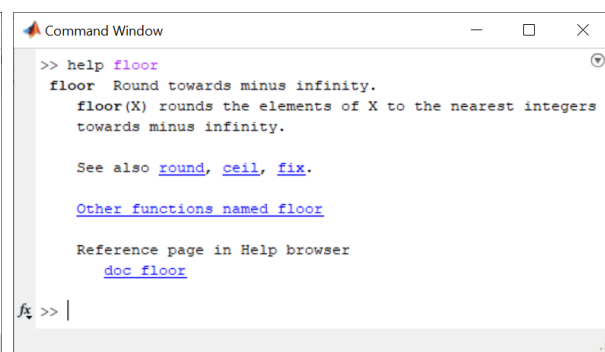
```

>> help ceil
ceil Round towards plus infinity.
ceil(X) rounds the elements of X to the nearest integers
towards infinity.

See also floor, round, fix.

Other functions named ceil

Reference page in Help browser
doc ceil
  
```



```

>> help floor
floor Round towards minus infinity.
floor(X) rounds the elements of X to the nearest integers
towards minus infinity.

See also round, ceil, fix.

Other functions named floor

Reference page in Help browser
doc floor
  
```

A common use of functions is to evaluate a formula for a series of arguments. Recall that the velocity of a free-falling bungee jumper can be computed with

$$v(t) = \sqrt{\frac{gm}{C_d}} \tanh\left(\sqrt{\frac{gC_d}{m}} t\right)$$

where v is velocity (m/s), g is the acceleration due to gravity (9.81 m/s²), m is mass (Kg). C_d is the drag coefficient (Kg/m), and t is time (s).

Let's create a column vector t that contains values from 0 to 20 in steps of 2. We can check the number of items in the t array with the `length` function.

MATLAB allows you to evaluate a formula such as $v = f(t)$, where the formula is computed for each value of the t array, and the results is assigned in the v array.

```
Command Window
>> t = [0:2:20]'

t =

     0
     2
     4
     6
     8
    10
    12
    14
    16
    18
    20

>> length(t)

ans =

    11

fx >> |
```

```
Command Window
>> g = 9.81; m = 68.1; cd = 0.25;
>> v = sqrt(g*m/cd)*tanh(sqrt(g*cd/m)*t)

v =

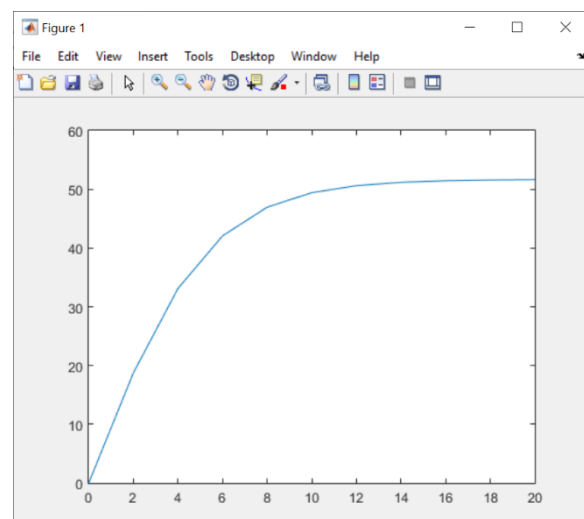
     0
    18.7292
    33.1118
    42.0762
    46.9575
    49.4214
    50.6175
    51.1871
    51.4560
    51.5823
    51.6416

fx >> |
```

Graphics

MATLAB allows graphs to be created quickly and conveniently. For example, to create a graph of the t and v arrays from the data above, we can use the `plot` function.

```
Command Window
>> plot(t, v)
fx >> |
```

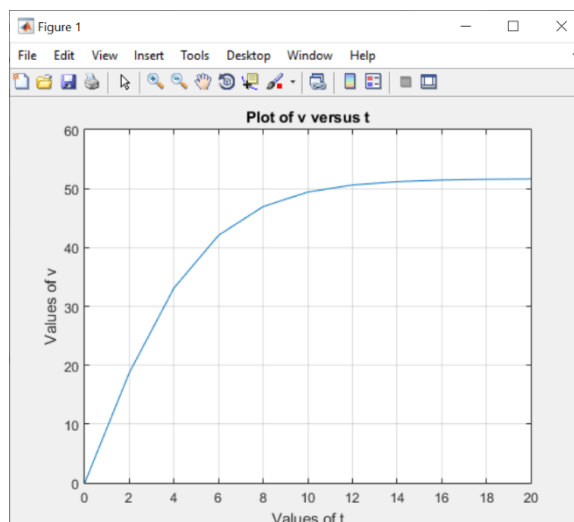


The graph appears in the graphics window and can be printed or transferred via the clipboard to other programs. You can customize the graph a bit with commands such as the following:

```

Command Window
>> title('Plot of v versus t')
>> xlabel('Values of t')
>> ylabel('Values of v')
>> grid
fx >> |

```



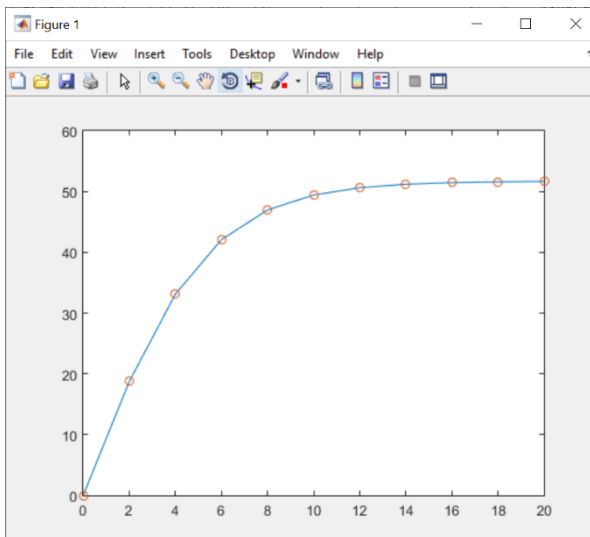
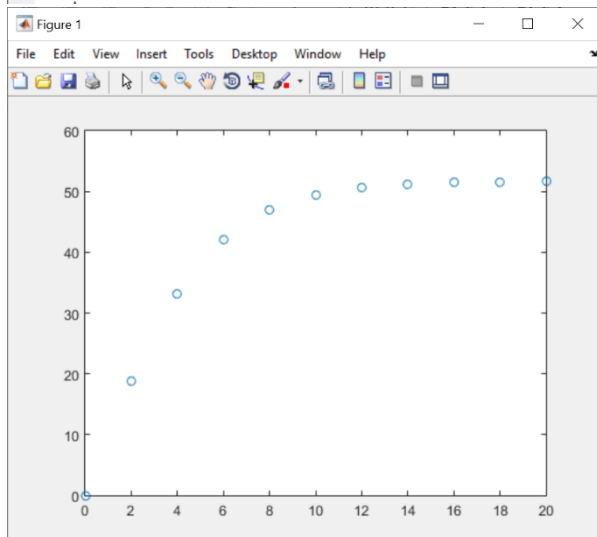
The plot command displays a solid line by default. If you want to plot each point with a symbol, you can include a specifier enclosed in single quotes in the plot function. You can refer to this table:

Colors		Symbols		Line Types	
Blue	b	Point	.	Solid	-
Green	g	Circle	o	Dotted	:
Red	r	X-mark	x	Dashdot	-.
Cyan	c	Plus	+	Dashed	--
Magenta	m	Star	*		
Yellow	y	Square	S		
Black	k	Diamond	D		
		Triangle (down)	v		
		Triangle (up)	^		
		Triangle (left)	<		
		Triangle (right)	>		
		Pentagram	p		
		Hexagram	h		

```

Command Window
>> plot(t, v, 'o')
>> plot(t, v, t, v, 'o')

```



The function `subplot` allows you to split the graph window into subwindows or panes. It has the syntax `subplot(m, n, p)`. This command breaks the graph into an m -by- n of small axes, and selects the p -th axes for the current plot.

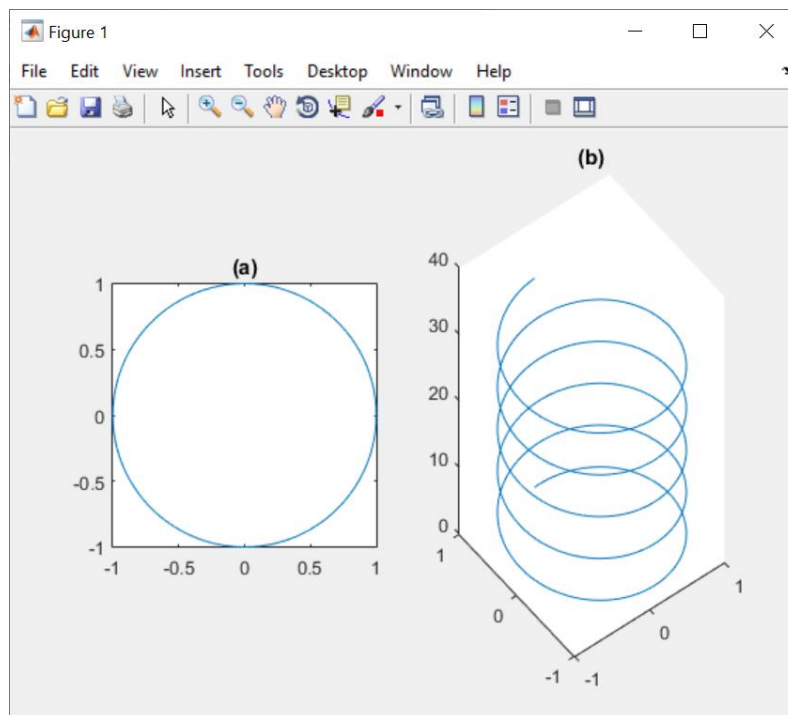
We can demonstrate subplot by examining MATLAB's capability to generate three-dimensional plots. The simplest manifestation of this capability is the `plot3` command which has the syntax `plot3(x, y, z)`. Where x , y , and z are three vectors of the same length. The results is a line in three-dimensional space through the points whose coordinates are the elements x , y , and z .

Plotting a helix provides a nice example to illustrate its utility. First, let's graph a circle with two-dimensional plot function using the parametric representation: $x = \sin(t)$ and $y = \cos(t)$. We employ the `subplot` command so we can subsequently add the three-dimensional plot.

```

Command Window
>> t = 0:pi/50:10*pi;
>> subplot(1,2,1);plot(sin(t),cos(t))
>> axis square
>> title(' (a) ')
>> subplot(1,2,2); plot3(sin(t),cos(t),t)
>> title(' (b) ')
fx >> |

```



If ever you encounter errors in using built-in functions, or get confused in what is happening in MATLAB, you can always ask for “help” by typing it on the command window.

PROGRESS CHECK

For the following numbers, you may either submit screenshots of the command window, or a .txt file of the codes you entered in the command window, or write the codes on the spaces provided.

1. Let the variable A be a row matrix (2, 4, 0, -1, 3), and B be a column matrix whose five elements are 2, 5, 8, 3, -5, in that order. Calculate the quantity $A * (B+1)$.

2. Set up the vector $v_1 = (0,1, 2,...,50)$ and calculate the length, $|v_1|$, as given by the formula

$$|v_1| = \sqrt{\bar{v}_1 * \bar{v}_1}$$

Where \bar{v}_1 is the mean of vector v_1

3. Create a vector of the even whole numbers between 31 and 75.

4. Let $x = [2 \ 5 \ 1 \ 6]$.

a) Add 16 to each element

b) Add 3 to just the odd-index elements

c) Compute the square root of each element

d) Compute the square of each element

5. Let $x = [3 \ 2 \ 6 \ 8]'$ and $y = [4 \ 1 \ 3 \ 5]'$ (x and y should be column vectors).

a) Add the sum of the elements in x to y

b) Raise each element of x to the power specified by the corresponding element in y .

c) Divide each element of y by the corresponding element in x

d) Multiply each element in x by the corresponding element in y , calling the result " z ".

e) Add up the elements in z and assign the result to a variable called " w ".

f) Compute $x'*y - w$ and interpret the result

6. Construct the helix made as an example using the function `plot3(x, y, z)`.

7. Construct another helix. This time use different trigonometric functions and values.

REFERENCES

Textbook/s :

Chapra, S. C. (2015). Applied Numerical Methods with MATLAB for Engineers and Scientists. New York: McGraw-Hill.

LEARNING GUIDE

Week No.: 3

TOPICS

1. M-Files
2. Input and Output
3. Structured Programming
4. Passing Function to M-Files

EXPECTED COMEPETENCIES

Upon completing this Learning Module, you will be able to:

1. Differentiate between the various types of files MATLAB creates
2. Perform input and output operations.
3. Create the various M-files to solve problems

TECHNICAL INFORMATION / CONTENT

M-FILES

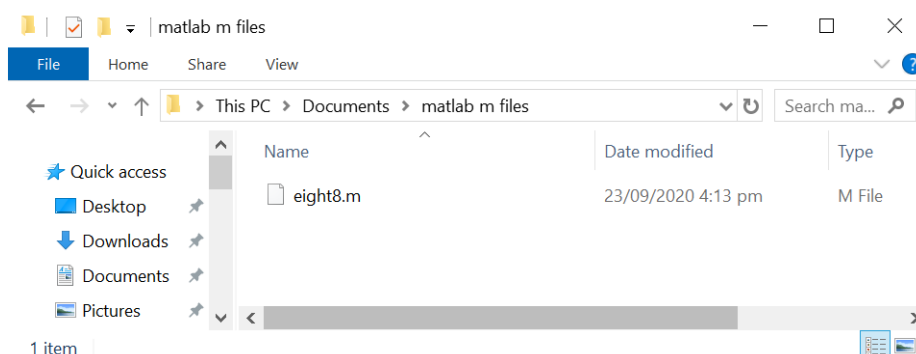
M-files provide an alternative way of performing operations that greatly expand MATLAB's problem-solving capabilities. An M-File contains a series of statements that can be run all at once. These files are called "m-files" because they are stored with .m extension. These can either be script files or function files (Chapra, 2015).

Script Files

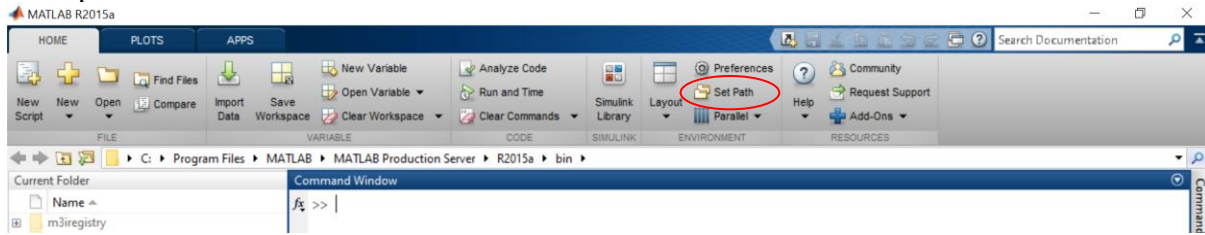
A script file is a series of MATLAB commands that are saved on a file. They are useful for retaining a series of commands that you want to execute on more than one occasion. The script can be executed by typing the file name in the command window (Chapra, 2015).

Before creating M-files, we need to first make a folder where we will save those M-files in. Please see this guide to properly "include a folder" for MATLAB.

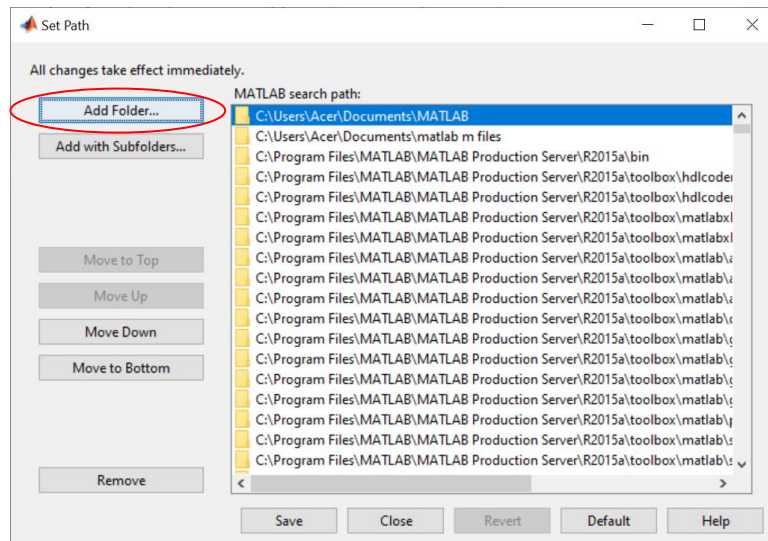
1. Create your folder.



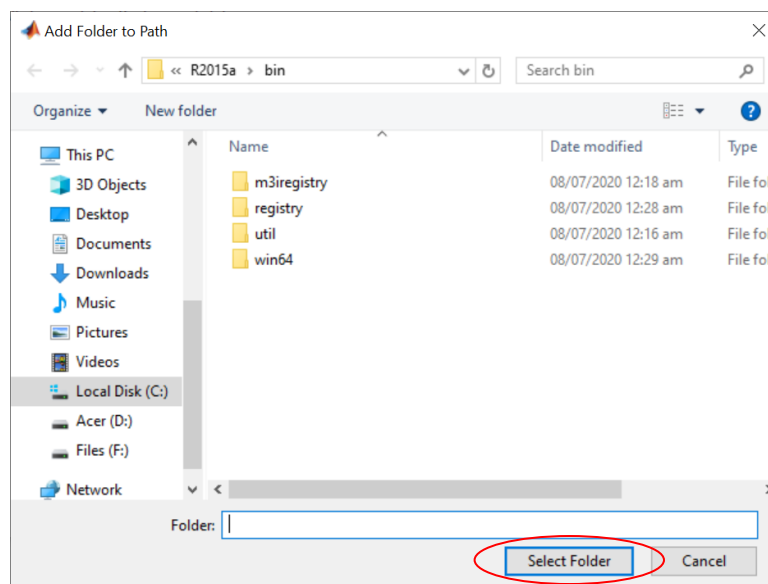
2. Open MATLAB and click on “Set Path”.



This will open the following:

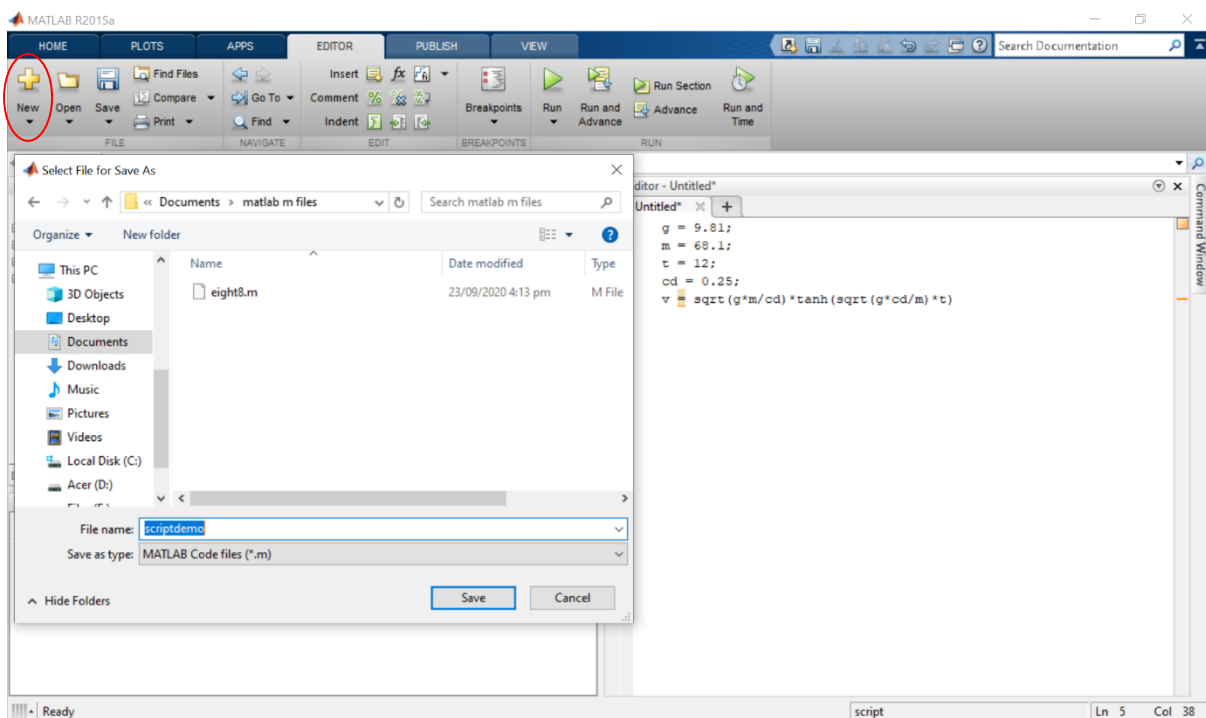


3. Notice that in my case, the folder has already been added (the second folder on the list). To add a folder, click on folder. Locate the folder you just created (“matlab m files” for example). And click on “Select Folder” when you are done.

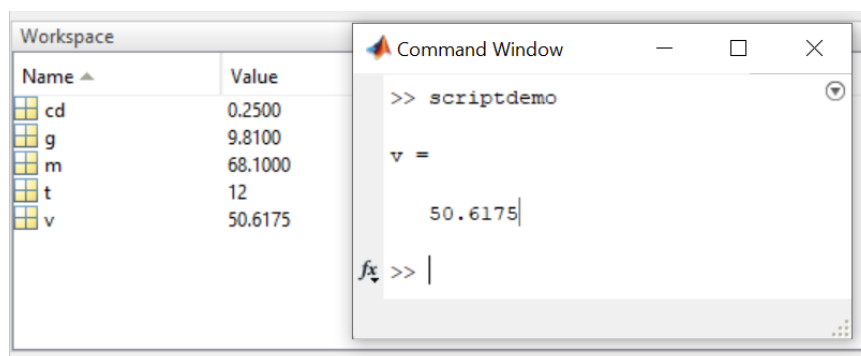


4. If that folder is going to have subfolders, then you should opt for “Add with subfolders” instead. When you are all done, your folder should appear on the list of folders like on the second image above.

Let's now make our first script file. Save as “scriptdemo.m”.



Using the command window, type in “scriptdemo”. The image below shows what is executed as well as what is happening in the workspace (variables are being added).



The script executes commands and/or operations just as if you had typed each of its lines in the command window. Notice on the left of the image that the workspace now is being populated by the variables used in the script. You can now use these variables in the command window as though they have been declared by typing them in.

If you are familiar with the “command prompt” of Microsoft Windows, they also have these equivalent files to script files, they are called “batch files” with file extension of .bat. Batch files contain a collection of codes that you would regularly type in the command prompt. A word of caution, batch files may contain code that may be harmful to your computer’s “health” if changed or deleted.

Function Files

Function files are M-files that start with the word function. In contrast to script files, they can accept input arguments and return outputs. Hence, they are analogous to user-defined functions in programming languages such as Visual Basic or C.

Syntax for the function file can be represented generally as

```
function outvar = funcname(arglist)
%helpcomments
statements
outvar = value;
```

outvar = the name of the output variable

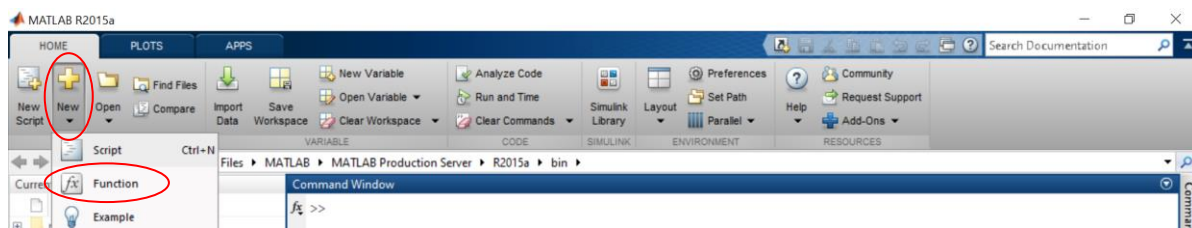
funcname = the name of the function

%helpcomments = like any other language has “comments” to provide the user with information regarding the function

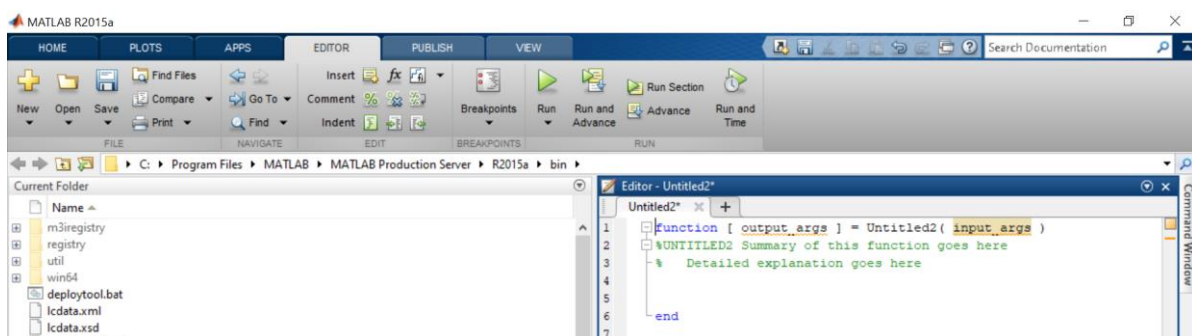
statements = MATLAB statements that compute the value that is assigned to outvar

The M-file should be saved as funcname.m. The function can then be run by typing funcname in the command window.

To start off, click on “New” and then “Function”.

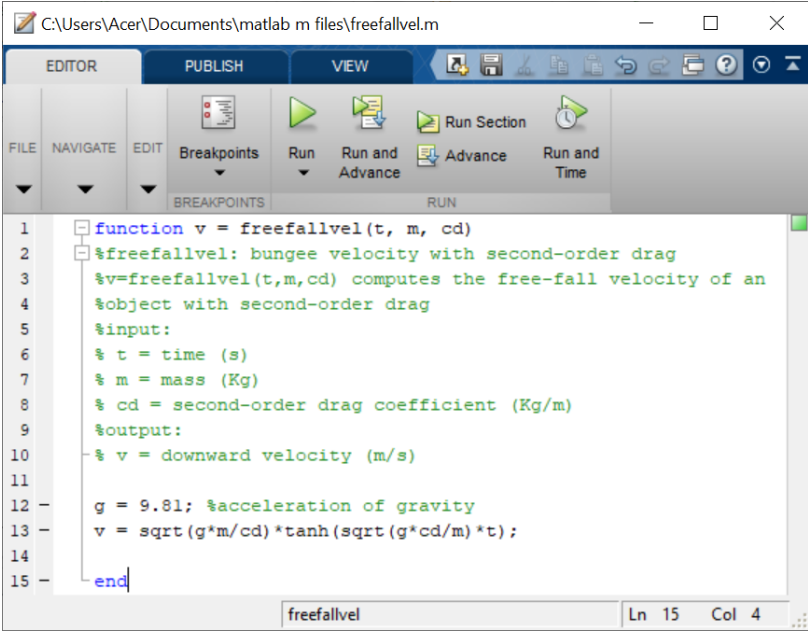


This will open the editor with the template for creating functions.



Notice that it has already made the structure for making a function so that we only need to edit it and add the statements of operations we need for MATLAB to do for us.

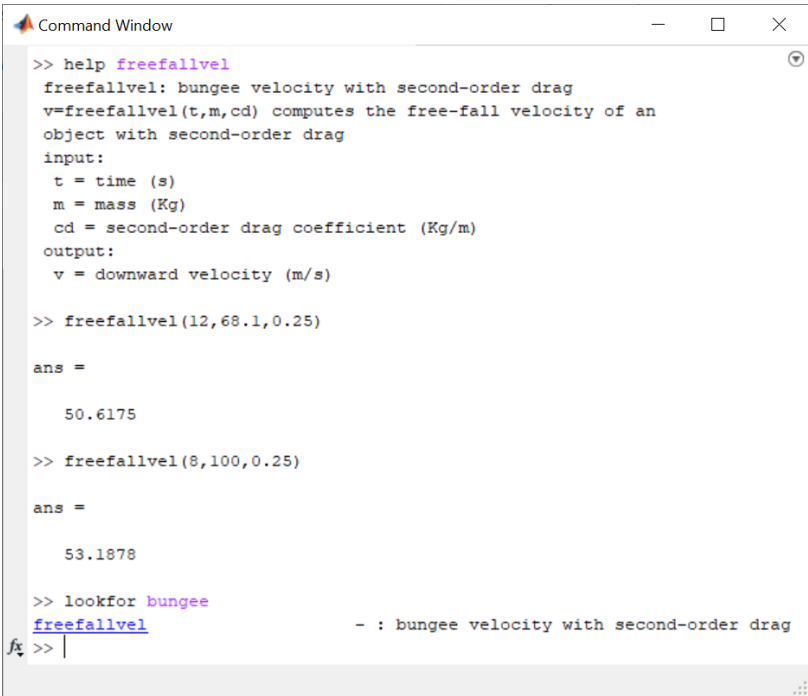
Type in the following to the editor.



```

1 function v = freefallvel(t, m, cd)
2 %freefallvel: bungee velocity with second-order drag
3 %v=freefallvel(t,m,cd) computes the free-fall velocity of an
4 %object with second-order drag
5 %input:
6 % t = time (s)
7 % m = mass (Kg)
8 % cd = second-order drag coefficient (Kg/m)
9 %output:
10 % v = downward velocity (m/s)
11
12 g = 9.81; %acceleration of gravity
13 v = sqrt(g*m/cd)*tanh(sqrt(g*cd/m)*t);
14
15 end
  
```

We'll start off with using the help function to tell us about our newly created function. Then, plug in the values we used in our script file example.



```

>> help freefallvel
freefallvel: bungee velocity with second-order drag
v=freefallvel(t,m,cd) computes the free-fall velocity of an
object with second-order drag
input:
  t = time (s)
  m = mass (Kg)
  cd = second-order drag coefficient (Kg/m)
output:
  v = downward velocity (m/s)

>> freefallvel(12,68.1,0.25)

ans =

    50.6175

>> freefallvel(8,100,0.25)

ans =

    53.1878

>> lookfor bungee
freefallvel - : bungee velocity with second-order drag
fx >>
  
```

Take note that variables within a function are said to be “local” and are erased after the function is executed. In contrast, the variables in a script retain their existence after the script is executed.

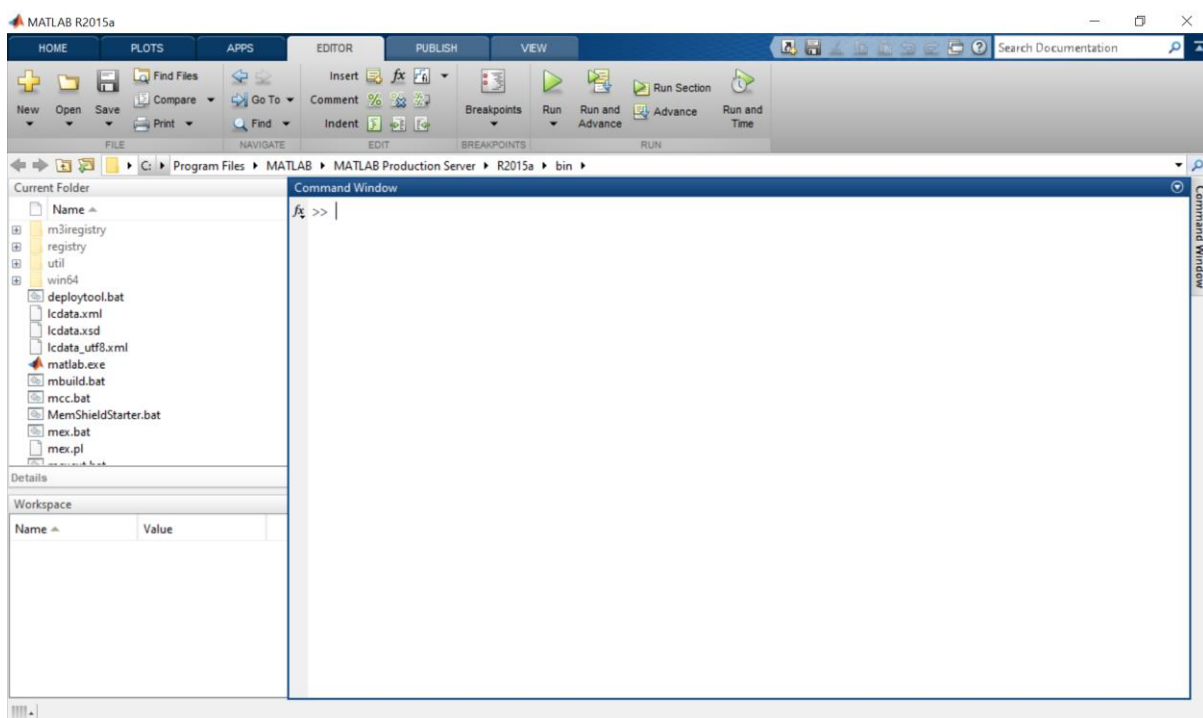
Function M-files can return more than one result. In such cases, the variables containing the results are comma-delimited and enclosed in brackets. For example, the following function,

`lu.m`, computes LU factorization of given matrix. But more of it will be discussed in the linear methods.

Before we proceed, you may be wondering “How do I erase stuff in MATLAB?” Well, you can do two things: erase your declared variables and/or erase what was outputted in the command window. Both are independent of each other. Let’s start with **clearing variables**. You can type in `clear` + name of the variable like `clear cd` and press enter. You can also **clear all** of them by typing `clear all` and press enter.



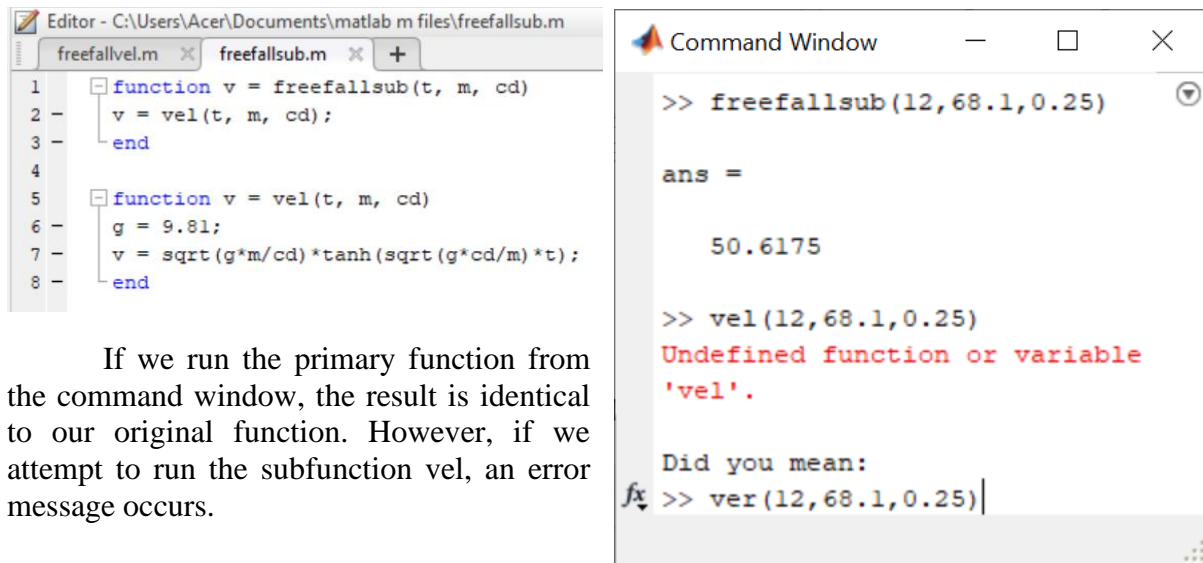
Notice that the command window is still cluttered with the results of previous computations. And so, to clear the command window, type in `clc` and press enter.



You won’t see the `clc` that I entered because it was also cleared by the function.

Functions can call other functions. Although such functions can exist as separate M-files, they may also be contained in a single M-file. For example, we could “double the functions inside of our `freedfallvel.m` but save it as a single M-file. In such cases, the first function is called the main or primary function. It is the only function that is accessible to the command window and other functions and scripts. All other functions (in this case, `vel`) are referred to as subfunctions.

A subfunction is only accessible to the main function and other subfunctions within the M-file in which it resides.



If we run the primary function from the command window, the result is identical to our original function. However, if we attempt to run the subfunction `vel`, an error message occurs.

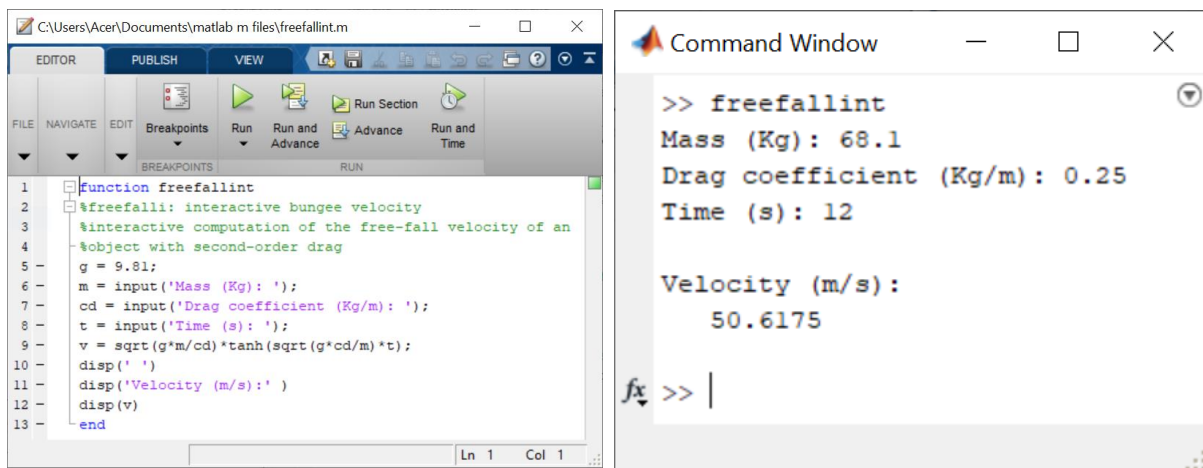
Input and Output

Information is passed into the function via the argument list and is output via the function's name. Two other functions provide ways to enter and display information directly using the command window.

The `input` function allows you to prompt the user for values directly from the command window. The function displays the `promptstring`, waits for keyboard input, and then returns the value from the keyboard. When the first line is executed, the user is prompted with a message. If the user enters a value, it would be assigned to the assigned variable.

The `input` function can also return user input as a string. To do this an 's' is appended to the function's argument list.

The `disp` function provides a way to display value. The things you can display can be a constant or a variable, or a string message enclosed in hyphens.



Structured Programming

The `if` structure allows you to execute a set of statements if a logical condition is true. For cases where only one statement is executed, it is often convenient to implement the `if` structure as a single line. For cases where more than one statement is implemented, the multiline `if` structure is usually preferable because it is easier to read.

The screenshot shows the MATLAB Editor with a file named `grader.m`. The function `grader(grade)` is defined with the following code:

```

1 function grader(grade)
2 % grader(grade):
3 % determines whether grade is passing
4 %input:
5 % grade = numerical value of grade (0 to 10)
6 %output:
7 % displayed message\
8 if grade >= 5.00 disp('Passed')
9 else disp('There is always a next time')
10 end

```

The Command Window shows the execution of the function:

```

>> grader(5)
Passed
>> grader(9)
Passed
>> grader(4.9999999)
There is always a next time
>>

```

The `if ... elseif` is an “upgrade” of the previous structure and is also more often used. This structure is used for multiple different conditions.

The `switch` structure is similar to the `if...else if` structure. But rather than testing individual conditions, the branching is based on the value of a single test expression. Depending on its value, different blocks of code are implemented. In addition, an optional block is implemented if the expression takes one none of the prescribed values.

The screenshot shows the MATLAB Editor with a file named `grader20.m`. The function `grader20(grade)` is defined with the following code:

```

1 function grader20(grade)
2 disp(grade)
3 if grade >= 9.00 && grade <= 10.00
4     remarks = 'Excellent';
5     disp(remarks)
6 elseif grade >= 8.00 && grade <= 8.99
7     remarks = 'Outstanding';
8     disp(remarks)
9 elseif grade >= 7.00 && grade <= 7.99
10    remarks = 'Very Good';
11    disp(remarks)
12 elseif grade >= 6.00 && grade <= 6.99
13    remarks = 'Good';
14    disp(remarks)
15 elseif grade >= 5.00 && grade <= 5.99
16    remarks = 'Passed';
17    disp(remarks)
18 else
19    remarks = 'Fail';
20    disp(remarks)
21 end
22 switch remarks
23 case 'Excellent'
24     disp('S Tier')
25 case 'Outstanding'
26     disp('A Tier')
27 case 'Very Good'
28     disp('B Tier')
29 case 'Good'
30     disp('C Tier')
31 case 'Passed'
32     disp('D Tier')
33 case 'Fail'
34     disp('F Tier')
35 end

```

The Command Window shows the execution of the function:

```

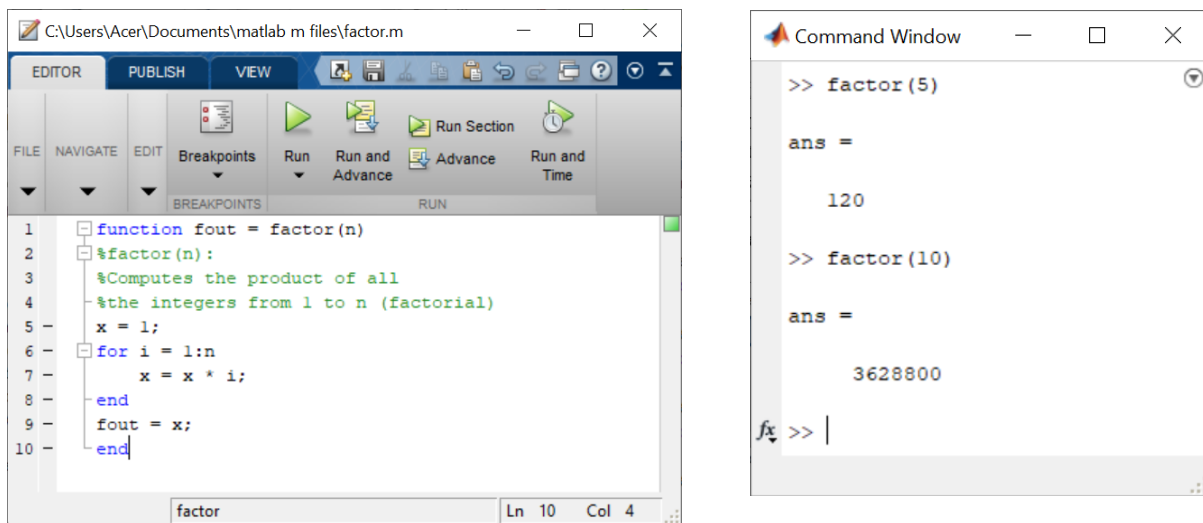
>> grader20(9.5)
9.5000
Excellent
S Tier
>> grader20(4.5)
4.5000
Fail
F Tier
>> grader20(4.9999)
4.9999
Fail
F Tier
>> grader20(1)
1
Fail
F Tier
>> grader20(7.85)
7.8500
Very Good
B Tier
>> grader20(8.01)
8.0100
Outstanding
A Tier
>> grader20(9.25)
9.2500
Excellent
S Tier
>>

```

Loops

As the name suggests, loops perform operations repeatedly. There are two types of loops, depending on how the repetitions are terminated. A for loop ends after a specified number of repetitions. A while loop ends on the basis of a logical condition.

The for loop has this `index` which is a variable that is set at an initial value, `start`. The program then compares the `index` with a desired final value, `finish`. If the `index` is less than or equal to the `finish`, the program executes the statements. When it reaches the end line that marks the end of the loop, the `index` variable is increased by the `step` and the program loops back up to the `for` statement. The process continues until the `index` becomes greater than the `finish` value. At this point, the loop terminates as the program skips down to the line immediately following the end statement.



The while loop repeats as long as a logical condition is true. The statements between the while and the end are repeated as long as the condition is true.

The `while... break` structure is a modified version of while where the `break` terminates execution of the loop. Thus, a single line `if` is used to exit the loop if the condition is true. The `break` can be placed in the middle of the loop.

The `pause` command causes a procedure to stop and wait until any key is hit. This is useful in times when you might want a program to temporarily halt. A nice example involves creating a sequence of plots that a user might want to leisurely peruse before moving on to the next.

We need to understand also that structures can be nested within each other. Nesting refers to placing structures within other structures. Indention helps make the underlying logical structure clear. Also, notice how modular the structures are.

The image displays the MATLAB Editor and Command Window. The Editor shows the `cdownup` function, which uses nested `if` and `while` loops to count down or up from `s` to `e`. The Command Window shows the execution of `cdownup(1,5)`, which outputs the sequence 1, 2, 3, 4, 5. Another Command Window shows the execution of `cdownup(5,1)`, which outputs the sequence 5, 4, 3, 2, 1.

```

function cdownup(s, e)
    %s = start    e = end
    if s < e
        while s <= e
            disp(s)
            s = s + 1;
            pause
        end
    else
        while s >= e
            disp(s)
            s = s - 1;
            pause
        end
    end
end

```

```

>> cdownup(1,5)
1
2
3
4
5

>> cdownup(5,1)
5
4
3
2
1

```

Here is another example of “nesting and indentation”:

The image displays the MATLAB Editor and Command Window. The Editor shows the `quadroots` function, which uses nested `if` and `else` statements to calculate the roots of a quadratic equation. The Command Window shows the execution of `quadroots(1,1,1)`, which outputs the real part of the first root (`r1`) as -0.5000, the imaginary part of the first root (`i1`) as 0.8660, the real part of the second root (`r2`) as -0.5000, and the imaginary part of the second root (`i2`) as -0.8660.

```

function quadroots(a, b, c)
    %quadroots: roots of quadratic equation
    % quadroots(a,b,c): real and complex roots of quadratic equation
    %input:
    % a = second-order coefficient
    % b = first-order coefficient
    % c = zero-order coefficient
    %output:
    % r1 = real part of first root
    % i1 = imaginary part of first root
    % r2 = real part of second root
    % i2 = imaginary part of second root
    if a == 0
        %special cases
        if b ~= 0
            %single root
            r1 = -c / b;
        else
            %trivial solution
            error('Trivial solution. Try again')
        end
    else
        %quadratic formula
        d = b ^ 2 - 4 * a * c; %discriminant
        if d >= 0
            %real roots
            r1 = (-b + sqrt(d)) / (2 * a);
            r2 = (-b - sqrt(d)) / (2 * a);
        else
            %complex roots
            r1 = -b / (2 * a);
            i1 = sqrt(abs(d)) / (2 * a);
            r2 = r1;
            i2 = -i1;
        end
    end
end

```

```

>> quadroots(1,1,1)

r1 =
    -0.5000

i1 =
    0.8660

r2 =
    -0.5000

i2 =
   -0.8660

```

Anonymous Functions

Anonymous functions allow you to create a simple function without creating an M-file. They can be defined within the command window.

```
Command Window
>> f1 = @(x, y) x^2 + y^2

f1 =

    @(x,y) x^2+y^2

>> f1(3,4)

ans =

    25
```

```
>> f2 = @(x) a*x^b

f2 =

    @(x) a*x^b

>> f2(3)

ans =

    36

fx >> |
```

Functions in a Function

There are functions that operate other functions which are passed to it as inputs arguments. The function that is passed to the function is referred to as the passed function. A simple example is the built-in function `fplot`, which plots the graphs of functions.

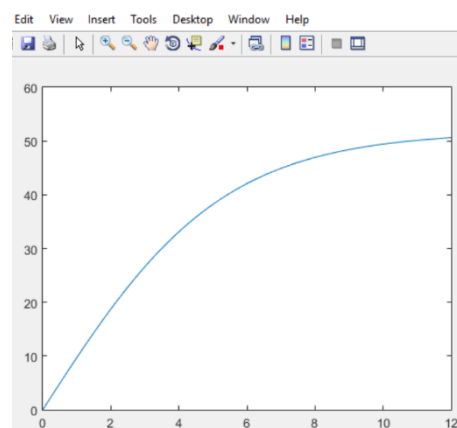
```
Command Window
>> vel = @(t) sqrt(9.81*68.1/0.25)*tanh(sqrt(9.81*0.25/68.1)*t)

vel =

    @(t) sqrt(9.81*68.1/0.25)*tanh(sqrt(9.81*0.25/68.1)*t)

>> fplot(vel,[0 12])

fx >> |
```



PROGRESS CHECK

1. An amount of money P is invested in an account where interest is compounded at the end of the period. The future worth F yielded at an interest rate i after n periods may be determined from the following formula:

$$F = P(1 + i)^n$$

Write an M-file that will calculate the future worth of an investment for each year from 1 to n . The input to the function should include the initial investment P , the interest rate i (as a decimal), and the number of years n for which the future is to be calculated. The output should consist of a table with headings, and columns for n and F . Run the program for $P = \$100,000$, $i = 0.06$, and $n = 7$ years.

Code:	Output:

2. The volume V of liquid in a hollow horizontal cylinder of radius r and length L is related to the depth of the liquid h by

$$V = \left[r^2 \cos^{-1} \left(\frac{r-h}{r} \right) - (r-h) \sqrt{2rh - h^2} \right] L$$

Develop an M-file to create a plot of volume versus depth. Test the program for $r = 2$ m and $L = 5$ m.

Code:	Output:

3. Develop an M-file function to determine the elapsed days in a year. The first line of the function should be set up as where `mo` = month (1 – 12), `da` = the day (1 – 31), and `leap` = (0 for non-leap year and 1 for leap year). Test it for January 1, 1999, February 29, 2000, March 1, 2001, June 21, 2002, and December 31, 2004. (A nice way to do this combines the `for` and the `switch` structures)

Code:	Output:

REFERENCES

Textbook/s :

Chapra, S. C. (2015). Applied Numerical Methods with MATLAB for Engineers and Scientists. New York: McGraw-Hill.

LEARNING GUIDE

Week No.: 4

TOPICS

1. Accuracy and Precision
2. Roundoff Errors
3. Truncation Errors

EXPECTED COMEPETENCIES

Upon completing this Learning Module, you will be able to:

1. Differentiate between accuracy and precision.
2. Quantify errors and apply them in deciding to terminate iterative calculations.
3. Familiarize how roundoff errors occur because digital computers have limited display capabilities.
4. Recognizing that truncation errors occur when exact mathematical formulations are represented by approximations

TECHNICAL INFORMATION / CONTENT

Accuracy and Precision

The errors with both calculations and measurements can be characterized with regard to their accuracy and precision. **Accuracy** refers to how closely a computed or measured value agrees with the true value. **Precision** refers to how closely individual computed or measured values agree with each other (Chapra, 2015).

These concepts can be illustrated graphically using an analogy from target practice. The bullet holes on each target can be thought of as predictions of a numerical technique, whereas the bullseye represents the true value.

Inaccuracy (also called **bias**) is defined as systemic deviation from the truth. **Imprecision** (also called **uncertainty**) refers to the magnitude of the scatter.

Refer to the image on the right for a more detailed visualization of the above terms.

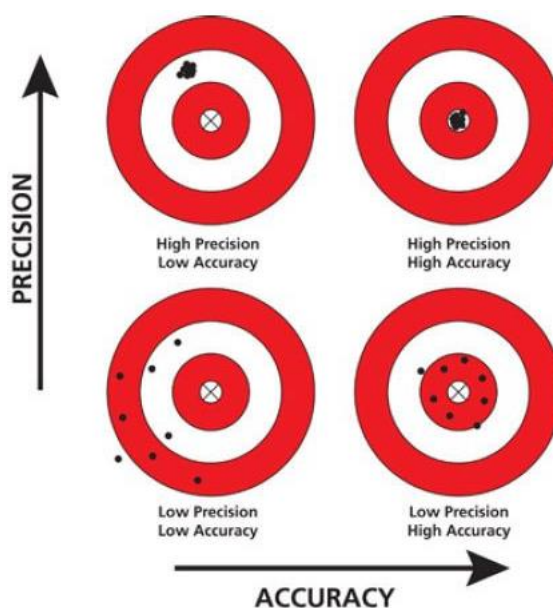


Image taken from <https://www.edvotek.com/>

Numerical methods should be sufficiently accurate or unbiased to meet the requirements of a particular problem. They also should be precise enough for adequate design. In this book, we will use the collective term error to represent both the inaccuracy and imprecision of our predictions.

Errors

Numerical errors arise from the use of approximations to represent exact mathematical operations and quantities. For such errors, the relationship between the exact, or true, result, and the approximation can be formulated as:

$$\text{True value} = \text{approximation} + \text{error}$$

$$E_t = \text{true value} - \text{approximation}$$

E_t is used to designate the exact value of the error. The subscript **t** is included to designate that this is the “true” error. An “approximate” estimate of the error must be employed. Note that the true error is commonly expressed as an absolute value and referred to as the absolute error.

An error of a centimeter is much more significant if we are measuring a rivet than a bridge. One way to account for the magnitudes of the quantities being evaluated is to normalize the error to the true value:

$$\text{True fractional relative error} = \frac{\text{true value} - \text{approximation}}{\text{true value}}$$

The relative error can also be multiplied by 100% to express it as

$$\varepsilon_t = \frac{\text{true value} - \text{approximation}}{\text{true value}} * 100\%$$

ε_t designates the true percent relative error.

For example, suppose that you have a task of measuring the lengths of a bridge and a rivet and come up with 9999 and 9 cm, respectively. If the true values are 10,000 and 10 cm, respectively, the error in both cases is 1cm. However, their percent relative errors can be computed using the above formula, which will yield 0.01% and 10%, respectively. Thus, although both measurements have an absolute error of 1 cm, the relative error for the rivet is much greater. Notice that the subscript **t** is used to signify that the error is based on the true value. For the example above, we were provided with the true value. However, in actual situations such information is rarely available.

For numerical methods, the true value will only be known when we deal with functions that can be solved analytically. Such will typically be the case in this module. However, in real-world applications, we will obviously not know the true answer. For these situations, an alternative is to normalize the error using the best available estimate of the true value, the approximation itself:

$$\varepsilon_a = \frac{\text{approximate error}}{\text{approximation}} * 100\%$$

Where the subscript a signifies that the error is normalized to an approximate value. One of the challenges of numerical methods is to determine error estimates in the absence of knowledge regarding the true value. Certain numerical methods use iteration to compute answers. In such cases, a present approximation is made on the basis of a previous approximation. This process is performed repeatedly, or iteratively, to successively compute better and better approximations. For such cases, the error is often estimated as the difference between the previous and present approximations. Thus, percent relative error is :

$$\varepsilon_a = \frac{\text{present approximation} - \text{previous approximation}}{\text{present approximation}} * 100\%$$

For example, functions can often be represented by infinite series. This exponential function can be computed using:

$$e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \cdots + \frac{x^n}{n!}$$

As more terms are added in sequence, the approximation becomes a better and better estimate of the true value of e^x . The above is called a “Maclaurin series expansion”. Starting with the simplest version, $e^x = 1$, add terms one at a time in order to estimate $e^{0.5}$. After each new term is added, compute the true and approximate percent relative errors. Note that the true value is $e^{0.5} = 1.648721$. Add terms until the absolute value of the approximate error estimate ε_a falls below a prespecified error criterion ε_s conforming to three significant figures:

$$\varepsilon_s = (0.5 * 10^{2-3})\% = 0.05\%$$

The first estimate is simply equal to:

$$e^x = 1 + x$$

$$e^{0.5} = 1 + 0.5 = 1.5$$

$$\varepsilon_t = \left| \frac{1.648721 - 1.5}{1.648721} \right| * 100\% = 9.02\%$$

$$\varepsilon_a = \left| \frac{1.5 - 1}{1.5} \right| * 100\% = 33.3\%$$

Because ε_a is not less than the required value of ε_s , we would continue the computation by adding another term, $\frac{x^2}{2!}$, and repeat the error calculations. The process is continued until $\varepsilon_a < \varepsilon_s$.

Terms	Results	$\varepsilon_t, \%$	$\varepsilon_a, \%$
1	1	39.3	
2	1.5	9.02	33.3
3	1.625	1.44	7.69
4	1.645833333	0.175	1.27
5	1.648437500	0.0172	0.158
6	1.64	0.00142	0.0158

After six terms are included, the approximate error falls below $\varepsilon_s = 0.05\%$, and the computation is terminated. However, rather than three significant, the result is accurate to five. But this is not always the case.

Roundoff errors

These arise because digital computers cannot represent some quantities exactly. In certain cases, they can usually lead to a calculation going unstable and yielding obviously erroneous results. Such calculations are said to be ill-conditioned. Worse still, they can lead to subtler discrepancies that are difficult to detect.

1. Digital computers have size and precision limits on their ability to represent numbers.
2. Certain numerical manipulations are highly sensitive to roundoff errors. This can result from both mathematical considerations as well as from the way in which computers perform operations.

Computer Number Representation

Numerical roundoff errors are directly related to the manner in which numbers are stored in a computer. The fundamental unit whereby information is represented is called a **word**. This is composed of a string of binary digits or **bits**. Numbers are typically stored in one or more words.

A number system is a convention for representing quantities. Because we have 10 fingers and 10 toes, the number system we are most familiar with is the decimal, or base-10 numbers system.

For larger quantities, combinations of these basic digits are used, making use of position or place value for the magnitude. For example, the **positional notation** for 8642.9 would look like:

$$(8 \times 10^3) + (6 \times 10^2) + (4 \times 10^1) + (2 \times 10^0) + (9 \times 10^{-1}) = 8642.9$$

Now, because the decimal system is so familiar, it is not commonly realized that there are alternatives. Like in computers, they only know two states – either 0 or 1. This relates to the fact that the primary logic units of digital computers are on/off electronic components. Hence, numbers on the computer are represented with a binary, or base-2 system. Quantities here can also be represented using positional notation. For example, 101.1_2 is equivalent to:

$$(1 \times 2^2) + (0 \times 2^1) + (1 \times 2^0) + (1 \times 2^{-1}) = 4 + 0 + 1 + 0.5 = 5.5 \text{ in decimal}$$

As we now have a better picture of how number systems work, we can now analyze how integers are represented on a computer. The most straight-forward approach, called signed magnitude method, employs the first bit of a word to indicate the sign, with 0 for positive and a 1 for negative. The remaining bits are used to store the number. For example, the integer value for -173 on a 16-bit computer using the signed magnitude.

1	0	0	0	0	0	0	0	1	0	1	0	1	1	0	1
Sign	Magnitude														

If such a scheme is employed, there clearly is a limited range of integers that can be represented. Again, assuming a 16-bit word size, if one bit is used for the sign, the 15 remaining bits can represent binary integers from 0 to 111111111111111. Thus, a 16-bit computer word can store decimal integers ranging from $-32,767$ to $32,767$.

Truncation Errors

These result from using an approximation in place of an exact mathematical procedure. In the first week, we approximated the derivative of velocity of a bungee jumper by a finite-difference equation of the form

$$\frac{dv}{dt} \cong \frac{\Delta v}{\Delta t} = \frac{v(t_{i+1}) - v(t_i)}{t_{i+1} - t_i}$$

A truncation error was introduced into the numerical solution because the difference equation only approximates the true value of the derivative. To gain insight into the properties of such errors, we now turn to a mathematical formulation that is widely used in numerical methods to express functions in an approximate fashion – the Taylor series.

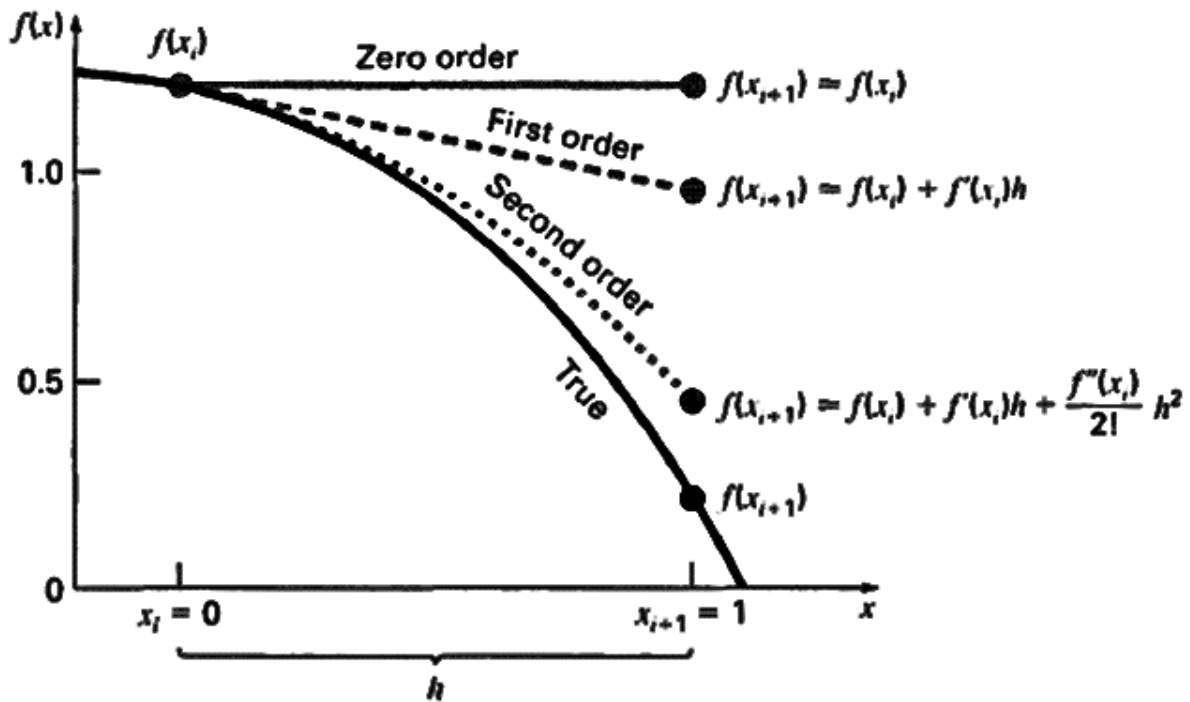
The Taylor Series

Taylor's theorem and its associated formula, the Taylor series, is of great value in the study of numerical methods. In essence, the Taylor Theorem states that any smooth function can be approximated as a polynomial. The Taylor Series then provides a means to express this idea mathematically in a form that be used to come up with practical results.

A useful way to gain insight into the Taylor series is to build it up term by term. A good problem context for this exercise is to predict a function value at one point in terms of the function value and its derivatives at another point.

$$f(x_{i+1}) \cong f(x_i)$$

The zero-order approximation indicates that the value of f at the new point is the same value at the old point. This result makes intuitive sense because if x_i and x_{i+1} are close to each other, it is likely that new value is probably similar to the old value.



$$f(x_{i+1}) \cong f(x_i) + f'(x_i)h$$

The first-order approximation because the additional first-order term consists of a slope $f'(x_i)$ multiplied by h , the distance between x_i and x_{i+1} . Thus, the expression is now in the form of a straight line that is capable of predicting an increase or decrease of the function between x_i and x_{i+1} .

To get a better prediction, we need to add more terms to our equation. We are now going to add a second-order term to the equation and so on.

$$f(x_{i+1}) = f(x_i) + f'(x_i)h + \frac{f''(x_i)}{2!}h^2 + \dots + \frac{f^{(n)}(x_i)}{n!}h^n + R_n$$

In general, we can usually assume that the truncation error is decreased by the addition of terms to the Taylor series. In many cases, if h is sufficiently small. The first- and other lower-order terms usually account for a disproportionately high percent of the error. Thus, only a few terms are required to obtain an adequate approximation.

Taylor Series Expansion Approximation Example

Problem

Use Taylor series expansions with $n = 0$ to 6 approximate $f(x) = \cos x$ at $x_{i+1} = \pi/3$ on the basis of the value of $f(x)$ and its derivatives at $x_i = \pi/4$. This means that $h = \pi/3 - \pi/4 = \pi/12$.

Solution

Our knowledge of the true function means that we can determine the correct value $f(\pi/3) = 0.5$. The zero-order approximation is

$$f\left(\frac{\pi}{3}\right) \cong \cos\left(\frac{\pi}{4}\right) = 0.707106781$$

which represents a percent relative error of

$$\varepsilon_t = \left| \frac{0.5 - 0.707106781}{0.5} \right| 100\% = 41.4\%$$

For the first-order approximation, we add the first derivative term where $f'(x) = -\sin x$:

$$f\left(\frac{\pi}{3}\right) \cong \cos\left(\frac{\pi}{4}\right) - \sin\left(\frac{\pi}{4}\right)\left(\frac{\pi}{12}\right) = 0.521986659$$

which has $|\varepsilon_t| = 4.40\%$. For the second-order approximation, we add the second derivative term where $f''(x) = -\cos x$:

$$f\left(\frac{\pi}{3}\right) \cong \cos\left(\frac{\pi}{4}\right) - \sin\left(\frac{\pi}{4}\right)\left(\frac{\pi}{12}\right) - \frac{\cos\left(\frac{\pi}{4}\right)}{2}\left(\frac{\pi}{12}\right)^2 = 0.497754491$$

with $|\varepsilon_t| = 0.449\%$. Thus, the inclusion of additional terms results in an improved estimate. The process can be continued, and the results listed:

Order n	$f^{(n)}(x)$	$f(\pi/3)$	$ \varepsilon_t $
0	$\cos x$	0.707106781	41.4
1	$-\sin x$	0.521986659	4.40
2	$-\cos x$	0.497754491	0.449
3	$\sin x$	0.499869147	2.62×10^{-2}
4	$\cos x$	0.500007551	1.51×10^{-3}
5	$-\sin x$	0.500000304	6.08×10^{-5}
6	$-\cos x$	0.499999998	2.44×10^{-6}

Taylor Series to Estimate Truncation Errors

$$v(t_{i+1}) = v(t_i) + v'(t_i)(t_{i+1} - t_i) + \frac{v''(t_i)}{2!}(t_{i+1} - t_i)^2 + \cdots + R_1$$

From the above formula, let us truncate the series after the first derivative term:

$$v(t_{i+1}) = v(t_i) + v'(t_i)(t_{i+1} - t_i) + \cdots + R_1$$

This now becomes

$$v(t_i) = \frac{v(t_{i+1}) - v(t_i)}{t_{i+1} - t_i} - \frac{R_1}{t_{i+1} - t_i}$$

First-order Approximation

Truncation Error

Numerical Differentiation

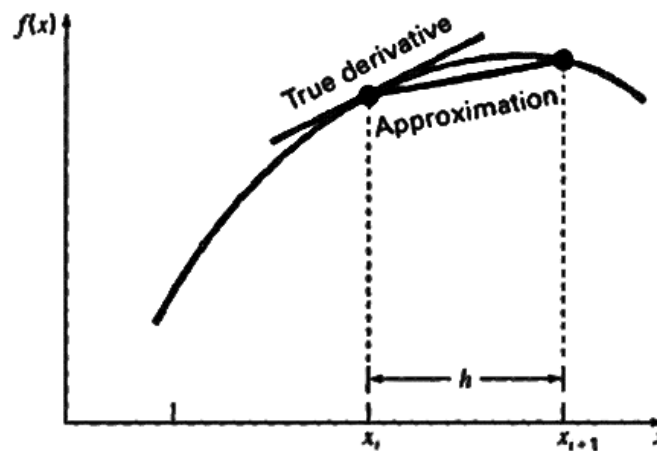
The above equation is an example of a *finite difference*. It can generally be represented as:

$$f'(x_i) = \frac{f(x_{i+1}) - f(x_i)}{x_{i+1} - x_i} + O(x_{i+1} - x_i)$$

or

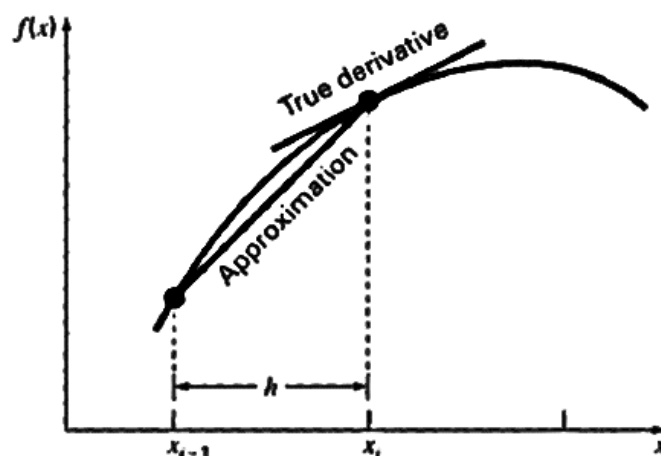
$$f'(x_i) = \frac{f(x_{i+1}) - f(x_i)}{h} + O(h)$$

where h is called the step-size- that is, the length of the interval over which the approximation is made. It is termed a Forward Difference because it utilizes data at i and $i + 1$ to estimate the derivative.



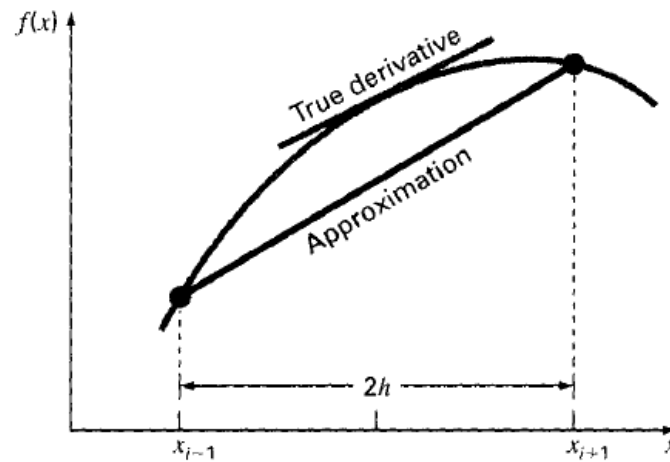
- Backward Difference Approximation of the First Derivative

$$f'(x_i) = \frac{f(x_i) - f(x_{i-1})}{h}$$



- Centered Difference Approximation of the First Derivative

$$f'(x_i) = \frac{f(x_{i+1}) - f(x_{i-1}))}{2h} - O(h^2)$$



Finite-Difference Approximations of Derivatives

Problem

Use forward and backward difference approximations of $O(h)$ and a centered difference approximation of $O(h^2)$ to estimate the first derivative of

$$f'(x) = -0.1x^4 - 0.15x^3 - 0.5x^2 - 0.25x + 1.2$$

at $x = 0.5$ using a step size $h = 0.5$. Repeat the computation using $h = 0.25$. Note that the derivative can be calculated directly as

$$f'(x) = -0.4x^3 - 0.45x^2 - 0.5x - 0.25$$

and can be used to compute the true values as $f'(0.5) = -0.9125$.

Solution

For $h = 0.5$, the function can be employed to determine

$$x_{i-1} = 0 \qquad f(x_{i-1}) = 1.2$$

$$x_i = 0.5 \qquad f(x_i) = 0.925$$

$$x_{i+1} = 1.0 \qquad f(x_{i+1}) = 0.2$$

These values can be used to compute the forward difference

$$f'(0.5) \cong \frac{0.2 - 0.925}{0.5} = -1.45 \qquad |\varepsilon_t| = 58.9\%$$

the backward difference

$$f'(0.5) \cong \frac{0.925 - 1.2}{0.5} = -0.55 \qquad |\varepsilon_t| = 39.7\%$$

and the centered difference

$$f'(0.5) \cong \frac{0.2 - 1.2}{1.0} = -1.0 \quad |\varepsilon_t| = 9.6\%$$

For $h = 0.25$,

$$x_{i-1} = 0.25 \quad f(x_{i-1}) = 1.10351563$$

$$x_i = 0.5 \quad f(x_i) = 0.925$$

$$x_{i+1} = 0.75 \quad f(x_{i+1}) = 0.63632813$$

the forward difference

$$f'(0.5) \cong \frac{0.63632813 - 0.925}{0.25} = -1.155 \quad |\varepsilon_t| = 26.5\%$$

the backward difference

$$f'(0.5) \cong \frac{0.925 - 1.10351563}{0.25} = -0.714 \quad |\varepsilon_t| = 21.7\%$$

and the centered difference

$$f'(0.5) \cong \frac{0.63632813 - 1.10351563}{0.5} = -0.934 \quad |\varepsilon_t| = 2.4\%$$

For both step sizes, the centered difference approximation is more accurate than forward or backward differences. Also, as predicted by the Taylor Series analysis, halving the step size approximately halves the error of the backward and forward differences and quarters the error of the centered difference.

Finite-Difference Approximations of Higher Derivatives

$$f(x_{i+2}) = f(x_i) + f'(x_i)(2h) + \frac{f''(x_i)}{2!}(2h)^2 + \dots$$

can be multiplied by 2 and subtracted to yield

$$f(x_{i+2}) - 2f(x_{i+1}) = -f'(x_i) + f''(x_i)h^2 + \dots$$

- Second Forward Finite Difference

$$f''(x_i) = \frac{f(x_{i+2}) - 2f(x_{i+1}) + f(x_i)}{h^2} = O(h)$$

- Second Backward Finite Difference

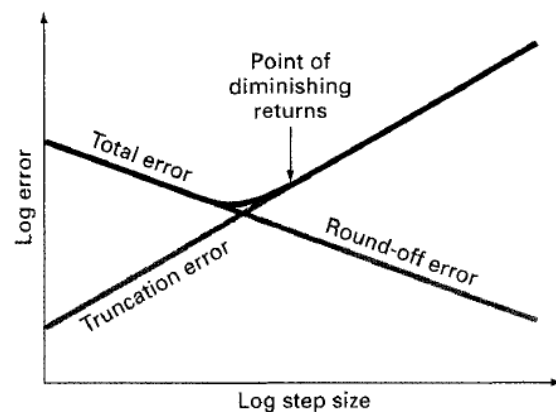
$$f''(x_i) = \frac{f(x_i) - 2f(x_{i-1}) + f(x_{i-2}))}{h^2} = O(h)$$

- Second Centered Finite Difference

$$f''(x_i) = \frac{f(x_{i+1}) - 2f(x_i) + f(x_{i-1}))}{h^2} = O(h)$$

Total Numerical Error

The total numerical error is the summation of the truncation and roundoff errors. In general, the only way to minimize roundoff errors is to increase the number of significant figures of the computer. Do note that roundoff error may increase due to subtractive cancellation or due to an increase in the number of computations in an analysis. However, truncation error can be reduced by decreasing the step size. Decrease in step size can lead to subtractive cancellation or to an increase in computations, the truncation errors are decreased as the roundoff errors are increased (Chapra, 2015).



Blunders

Blunders can occur at any stage of the mathematical modeling process and can contribute to all other components of error. They can be avoided only by sound knowledge of fundamental principles and by the care with which you approach and design your solution to a problem.

Blunders are usually disregarded in discussions of numerical methods. This is because, human as we are, no matter how careful we may be, some mistakes are to a certain extent unavoidable.

Model Errors

Model errors relate to bias that can be ascribed to incomplete mathematical models. An example of a negligible model error is the fact that Newton's second law does not account for relativistic effects. Because errors are minimal on the time and space scales associated with the bungee jumper. With this type of error, if you are working with a poorly conceived model, no numerical method will provide adequate results.

Data Uncertainty

Errors sometimes enter into an analysis because of uncertainty in the physical data on which a model is based. For instance, suppose we wanted to test the bungee jumper model by having an individual make repeated jumps and then measuring his or her velocity after a specified time interval. Uncertainty would undoubtedly be associated with these measurements, as the parachutist would fall faster during some jumps than during others. These errors can exhibit both inaccuracy and imprecision. If our instruments consistently underestimate or overestimate the velocity, we are dealing with an inaccurate, or biased, device. If the measurements are randomly high and low, we are dealing with a question of precision.

Measurement errors can be quantified by summarizing the data with one or more well-chosen statistics that convey as much information as possible regarding specific characteristics of the data.

PROGRESS CHECK

1. Use zero- through third-order Taylor series expansions to predict $f(3)$ for

$$f(x) = 25x^3 - 6x^2 + 7x - 88$$

2. Use zero- through fourth-order Taylor series expansions to predict $f(2)$ for $f(x) = \ln x$ using a base point at $x = 1$. Compute the true percent relative error ε_t for each approximation. Discuss briefly the meaning of the results.

3. The Maclaurin series expansion for $\cos x$ is

$$\cos x = 1 - \frac{x^2}{2} + \frac{x^4}{4!} - \frac{x^6}{6!} + \frac{x^8}{8!}$$

Starting with the simplest version, $\cos x = 1$, add terms one at a time to estimate $\cos(\pi/3)$. After each term is added, compute the true and approximate percent relative errors. Add terms until the absolute value of the approximate error estimate falls below an error criterion conforming to two significant figures.

4. Consider the function the function $f(x) = x^3 - 2x + 4$ on the interval $(-2, 2)$ with $h = 0.25$. Use the forward, backward, and centered finite difference approximations for the first derivatives so as to graphically illustrate which approximation is most accurate. Graph all three first-derivative finite difference approximations long with the theoretical.

REFERENCES

Textbook/s :

Chapra, S. C. (2015). Applied Numerical Methods with MATLAB for Engineers and Scientists. New York: McGraw-Hill.