Murad Elarbi

Programming Numerical Methods in MATLAB

Sample chapters

MechTutor.com

For the full version of the e-book,
https://www.mechtutor.com/downloads/programming-numerical-methods-in-matlab/

Programming Numerical Methods in MATLAB

For any questions or comments, please contact us via info@mechtutor.com

About the author

Murad Elarbi is a mechanical engineer and lecturer at the University of Benghazi. He taught courses of mechanical engineering like Engineering Drawing, Dynamics, Strength of Materials, Theory of Machines and Machine Design Projects. Because of his interest in computer programming and numerical methods, he programmed many engineering solutions at different levels and instructed numerous computer programming training courses in BASIC, FORTRAN, C++ and MATLAB. He also has recently instructed an online course about Numerical Methods in Python.

Save A Tree!

Dear reader, please keep on using this e-publication in the electronic form (pdf file) since every paper-print may lead to lose more of our Earth's beautiful trees.

Preface

Numerical methods have great and increasing importance in the scientific and engineering computations. This is because most of the mathematical formulas developed from the real life cases of study cannot be solved by the analytical methods due to many factors such as nature, geometry, composition and internal and external affecting forces.

The nature of a problem could lead to a total change in the computational discipline. For example, the problem of a laminar flow of a fluid inside a pipe can be solve with direct analytical equations because of the predictable nature of the fluid particles at low speeds. But when the flow exceeds a specific speed, and with the influence of the physical properties of the fluid and the geometrical limits of the pipe, the motion of the particles becomes chaotic and flow is then known as turbulent. In this case, another set of equations shall govern the problem which needs to be solved numerically.

Since the mathematical formulas are derived depending on assumptions based on geometries governed by simple equations, such as straight lines, triangles, circles and ellipses, the intricate geometric elements of a structure or boundaries of a thermodynamic or a hydraulic system need to be modeled with equations that have parallel level of complication and require the application of numerical methods in solution.

The composition of the matter is highly important in creating an accurate mathematical model. For example, the materials that have homogeneous mechanical properties in the all directions are known as isotropic materials. This property makes them extremely easy to be modeled since one equation is valid to solve a vast range of problems. Meanwhile, some other materials have totally different mechanical properties, or even nature, at each direction, like fibrous and composite materials. In such a case, numerical analysis becomes more likely to be the only solution technique.

In science and engineering, all physical, chemical and biological systems are affected in some way by variety of internal and external factors such as gravitational forces, fluid pressure, transferred or generated heat and electrical charge and environmental changes. Each of those effects has to be represented within the mathematical model of the studied phenomenon or solved problem. The more factors influence the case, the more complicated model it requires and, hence, more need to numerical analysis techniques rises.

All those and more other reasons made the courses of numerical analysis methods essential parts of the curricula of the most science and engineering schools in both undergraduate and graduate levels.

Although most basic numerical methods were developed before the invention of the digital computers, they are frequently described as “computer methods.” This is due to the repetitive nature of the numerical methods which makes them extremely tedious and time-

consuming for hand calculations while a simple computer program can perform millions of mathematical operations in few seconds.

MATLAB is one of the most well-known computing environments in science and engineering. Beside its simple and intuitive user interface, it contains a huge number of mathematical and numerical functions in numerous fields of applications. It has a powerful interpreter to executed commands directly and its programming language is remarkably simple. That's why it is one of the easiest computer languages to learn and apply.

In this book, I have introduced the programming steps of the most basic numerical methods in a simplified way by using MATLAB functions and statements, and I believe this will help the students who study the numerical methods and need to learn how they are coded. I have kept the codes of this book as simple and basic as possible to avoid the distraction of the reader from the main concept by secondary or redundant statements.

Finally, I hope Programming Numerical Methods in MATLAB will be a helpful resource for the all readers who are interested in the topic.

Murad Elarbi

Denver, Colorado
January 5, 2018

Contents

Chapter 1

Introduction

Numerical methods

The numerical methods, or numerical analysis, are simplified procedures to get *approximate numerical solutions* to equations and problems in algebra, calculus and other fields of mathematics. These procedures should be organized in definite and general steps that are applicable to the problem which they are formulated to solve. These steps are known as algorithms and can be programmed by using a computer language.

Programming languages used in numerical methods

Numerical calculation of science and engineering problems was one of the first applications of computers in the 1950's. The early high-level computer language used for the purpose was FORTRAN. Development of other powerful languages like Pascal, C++, Java and Python in addition to advanced computation environments such as MATLAB and Mathematica increased the efficiency of codes and techniques of numerical methods in parallel with the advantages of each language.

The goal of the book

Programming Numerical Methods in MATLAB aims at teaching how to program the numerical methods with a step-by-step approach in transforming their algorithms to the most basic lines of code that can run on the computer efficiently and output the solution at the required degree of accuracy. Thus, the reader can comprehend the fundamental numerical procedures that are required to develop more advanced codes for study and research.

Why MATLAB programming language

In this book, MATLAB is used in programming the numerical methods because it has many advantages:

1- It has very simple syntax. Statements and functions can be written in very simple formats in comparison with other languages.

2- All functions are built-in by default, so no external libraries or header files are needed to be included at the beginning of each program.

3- Vectors and matrices can be created and manipulated by means of its pre-defined operators and functions.

4- Plotting function are included by default, so neither external plotting modules are needed to be imported nor data have to be exported to another plotting application.
5- It includes the whole known numerical methods as built-in functions.
6- It is available in most colleges since it is installed in the computer labs of most science and engineering departments in the world.
7- It is taught in many colleges of science and engineering and applied as computational tool in solving the problems of many courses.

What this book focuses at

This book focuses mainly on the programming steps of the basic numerical methods that are studied in a first course on numerical method. Thus, it is designed to be an additional practical resource for the students who study numerical analysis.
The most of the codes in this book are written in the basic MATLAB programming statements and functions which does not require a thorough experience in MATLAB to understand. Furthermore, the students who study courses of science or engineering that require numerical programming in other languages could find this book useful in coding for their assignments and project.

What is needed to read this book

In order to get the best outcome of the topics and the codes *of Programming Numerical Methods in MATLAB*, it is recommended to have
1- a good background in algebra and calculus, in addition to the basic knowledge about computers since they are the pre-requisite of any Numerical Analysis course,
2- some basic knowledge about MATLAB in order to be able to run the codes of each section,
3- MATLAB installed on the computer that will be used in working the examples and the exercises. Otherwise, Octave can be installed and used without any noticeable difference in programming tools required in this book. It can be downloaded for free from https://www.gnu.org/software/octave/

Source code files

A companion zip folder that includes the MATLAB m-files of the programs of this book should be downloaded with it at purchase from https://www.MechTutor.com.

Chapter 2
Roots of High-Degree Equations

High-degree equations can be polynomials or equations containing radicals and/or transcendental (trigonometric and logarithmic) functions. The simplest example of high-degree equations is the quadratic equation:

$$ax^2 + bx + c = 0$$

Its roots can easily be obtained by the equation:

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

But in case of equations of higher degrees (power) or when terms of transcendental functions exist, numerical methods become the only way to obtain their roots.

Simple Iterations Method

The simple iterations is the simplest method used in finding roots of high-degree equation. In this method, the equations is rearranged to have the variable on the left side or the equation. An initial value of the variable, known as initial guess, substituted in the right side and a new value for the variable calculated. In the second iteration (repeat) the new value substituted on the right side. This cycle will continue until the old value and resulting new value of the variable become theoretically equal. Thus, this value will be one of the roots of the equations. Other root can be found with the same manner just by changing the initial guess.

Now, let's write solution steps as an algorithm:

1- Rearrange the equation so that the variable is put on the left side
2- Assume (guess) an initial value of the variable to start the first iteration
3- Substitute the value of the variable in the right side of the equation and calculate an *new* value for the variable
4- If the new value of the variable is not equal to the previous value, consider the new one as the value of the variable.
5- Repeat steps 3 and 4 until the new value is equal to the old value of the variable. Then output the value of the variable and stop.
6- In case the new value does not approach the old value (the difference increases at each iteration) stop calculation and try another initial value or another rearrangement of the given equation.

Example 2-1

Find the roots of the following equation:

$$2x^2 - 5x + 3 = 0$$

(Analytical solutions: $x = 1.5$ and $x = 1$)

Solution

The first step in algorithm should be performed manually. So, the equation can be rearranged in the forms:

$$x = \frac{2x^2 + 3}{5}$$

or

$$x = \sqrt{\frac{5x - 3}{2}}$$

Through this first example, we will see how develop a program of the numerical method form the scratch depending on the algorithm listed above. Because the method is based on repeated trials, the main part of the code will be a loop (a for-loop or a while-loop) to make the required repetitions. Let's start with a for-loop.

```
x = 0;                      % initial guess value (step 2)
for iteration = 1:100       % start a for-loop
  xnew = (2*x^2+3)/5;       % the new value of x (step 3)
  if xnew ~= x              % if the new x not equal x (step 4)
       x = xnew;            % x takes value of the new x (step 4)
  else                      % in case x equals to new x (step 5)
       break                % end the loop (step 5)
  end
end
disp(x)                     % display (output) x (step 5)
```

The output in the command window will be displayed as

```
 1.00000
>>
```

This is a good result since 1.0 is one of the roots according to the analytical solution. Let's see in how many iterations the solution was approached by printing the value of x inside the loop at the corresponding value of iteration as following

```
x = 0;
for iteration = 1:100
   xnew = (2*x^2 + 3)/5;
   fprintf('iteration = %d\tx = %f\n',iteration,x)
   if xnew ~= x
       x = xnew;
   else
       break
   end
end
```

The output resulting from the `fprintf` function will be

```
iteration = 1   x = 0.000000
iteration = 2   x = 0.600000
iteration = 3   x = 0.744000
       :            :
iteration = 58  x = 0.999999
iteration = 59  x = 0.999999
iteration = 60  x = 1.000000
       :            :
iteration = 98  x = 1.000000
iteration = 99  x = 1.000000
iteration = 100 x = 1.000000
```

Why the program did not stop at the iteration 60, where it approached the solution, and continued to 100? The answer can simply be discovered by printing the value of xnew in the same fprintf function and setting the displayed decimal digits of x and xnew: to 15:

```
fprintf('iteration=%d\tx=%.15f\txnew=%.15f\n',iteration,x,xnew)
```

The values at the last iteration are displayed as

```
iteration=100   x=0.999999999939488     xnew=0.999999999951591
```

The result shows that the equality has never occurred even at the last iteration because of the difference after the 10^{th} decimal digit.
Since the solution of any numerical method is obtained by using the computer, the values are directly affected by the precision of the data types of the variables. This means that the *exact equality* between two real (or double precision) variables is extremely difficult if not impossible.
The usual practice in numerical methods is to specify the **degree of accuracy** that is required in the solution. For example, an accuracy up to the second decimal digit can be acceptable for economical calculations when the values are in Dollars, or if major measurements are made in meters in a carpentry workshop, the accuracy of 3 decimal digits gives exact results since one millimeter is the smallest measurable minor unit.
According to this concept, the condition should be modified in a way that can consider the required degree of the accuracy for the problem. In many references, it is called the **tolerance** and it represents the acceptable **absolute maximum difference** between values of the variable of the equation resulted from the two successive iterations, in other words, the difference between the old and new values. Therefore, all what we have to do is to suggest a value for the tolerance and modify the condition as following

```
if abs(x - xnew) >= 0.000001 %compared to the tolerance
```

With the degree of accuracy of 0.000001 (or 1.0E-6), the last line of the output becomes

```
iteration=50    x=0.999995760196990     xnew=0.999996608164782
```

So, the run ended at the 50th iteration when the condition is satisfied. It is recommended to display the results at a same or a close number of decimal digits to the tolerance. Thus, the precision specifier can be set to %.5f in the fprintf function and the last two lines of the output will be

```
iteration=49    x=0.99999       xnew=1.00000
iteration=50    x=1.00000       xnew=1.00000
```

Finally, instead of cluttering the command window by the output of each iteration, it is preferred to displays the final new value and the corresponding number of iterations at the end of the program. Thus, this is the final simplest code

```
x = 0;
for iteration = 1:100
    xnew = (2*x^2 + 3)/5;
    if abs(x - xnew) >= 0.000001
        x = xnew;
    else
        break
    end
end
fprintf('iteration=%d\tx=%.5f\n',iteration,xnew)
```

Convergence and Divergence

The solution obtained for the quadratic equation in the example showed that the values of x and xnew were getting closer at each iteration and at the iteration number 50 both values were considered equal when the required condition of accuracy was satisfied. This behavior is called the *convergence* of values to a specific solution. The Figure 2-1 shows the convergence of the solution of example.

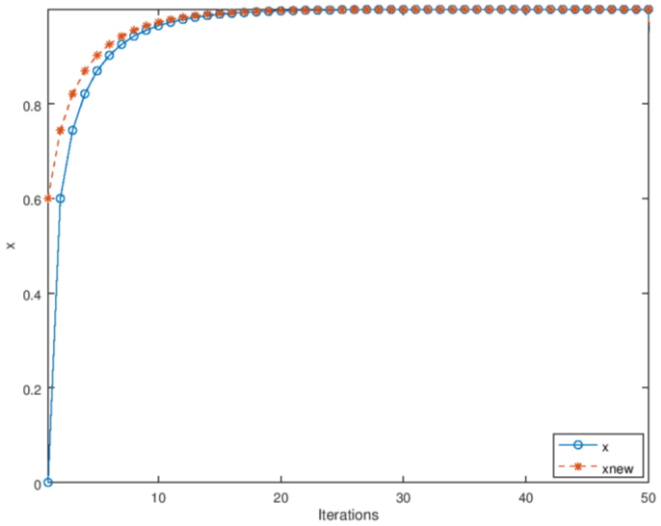

Figure 2-1. The differences between values of *x* trials (the vertical distance) at each iteration.

On the contrary, the *divergence* occurs when the difference between the values of the variable gets larger at each iteration until the final value of iterations loop is reached. Divergence can be resulted from the type of the equation, the initial guess and/or method of rearrangement. Figure 2-2 shows the behavior of the variables in case of divergence. The equation is: $x\cos(x) - 1 = 0$

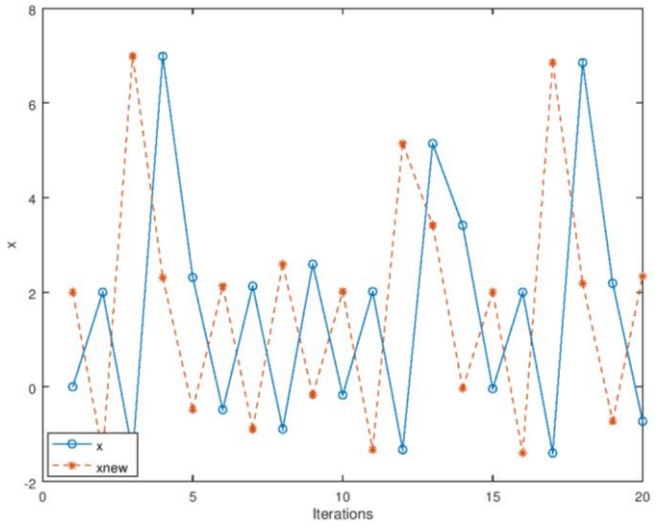

Figure 2-2. The irregular differences between x trial values.

In order to get the second root of the quadratic equation of example, the second rearrangement of the equation should be used.

Newton-Raphson Method

The Newton-Raphson method has the same procedure of the simple iteration with a main difference in the step one. Instead of simple rearrangement of the given equation, a new equation should be formulated by using the given equation and its first derivative with respect to its variable. The formula to be put in the code is as following:

$$x_{new} = x - \frac{f(x)}{f'(x)}$$

Example 2-2

Find the roots of the following equation:

$$2x^2 - 5x + 3 = 0$$

Solution

Let's write the equation as a function then find its first derivative:

$$f(x) = 2x^2 - 5x + 3$$

So,

$$f'(x) = 4x - 5$$

Now, the formula of the method becomes

$$x_{new} = x - \frac{2x^2 - 5x + 3}{4x - 5}$$

By plugging this formula in the same code of simple iterations method we get

```
x = 0;
for iteration = 1:100
   xnew = x-(2*x^2-5*x+3)/(4*x-5);
   if abs(x - xnew) >= 0.000001
      x = xnew;
   else
      break
   end
end
fprintf('iteration=%d\tx=%.5f\n',iteration,xnew)
```

the output will be:

```
iteration=7     x=1.00000
>>
```

Amazing! The root have been approached at seven iterations only. This demonstrates the efficiency of the Newton's method since it could converge about 7 times faster than the simple iterations method.

Another advantage is that only one formulation of the given equation can be used in computing the other root(s). For the equation given in the example, the second root can be simply obtained by setting the initial guess value to 2 (`x = 2;`) and the output will be

```
iteration=6      x=1.50000
>>
```

Bisection Method

This method is based on the fact that a root of an equation is point where its curve crosses the *x*-axis. For example, the graph of the equation $y = 2x^2 - 5x + 3$ is

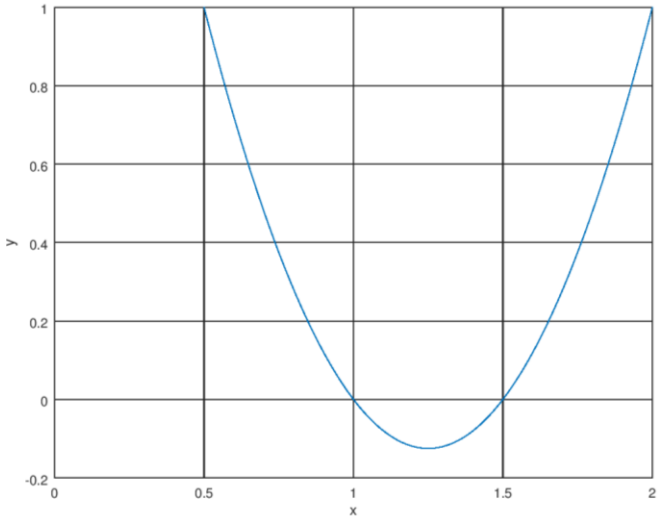

Figure 2-3. The curve of the equation $y = 2x^2 - 5x + 3$

The curve crosses the *x*-axis at 1 and 1.5 (i.e. when $y = 0$) which are the roots of the equation. Also, it can be noticed that the values of *y* for points located to one side of a root have opposite sign to those at the other side. Accordingly, we can simply imagine that the bisection method as a search for these points along the *x*-axis where sign of *y*(*x*) changes. Practically, it searches for the points that are embraced between two values of different signs. Then interval between the values is halved and the half having two values of different sign is selected for the next halving and so on, as shown in Figure 2-4. By halving the intervals at each time the requested point (root) can be found and by changing starting search points the other roots can be determined in the same way.

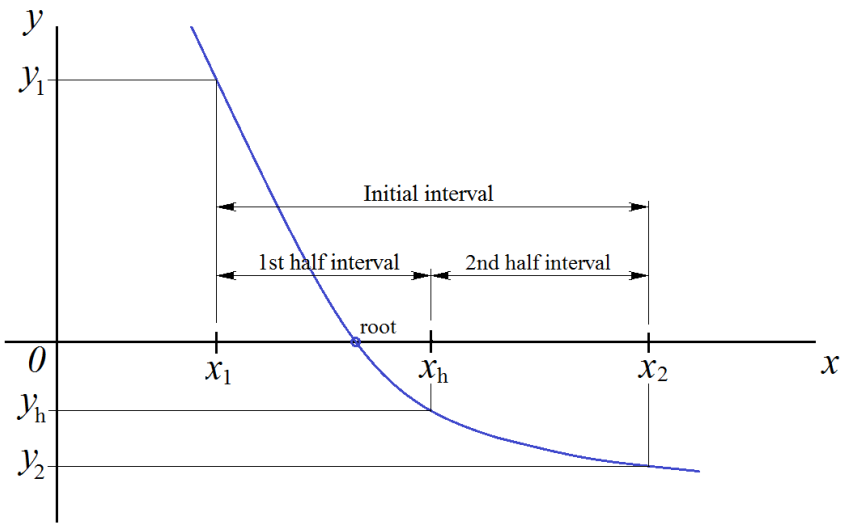

Figure 2-4. Bisecting the interval [x_1 x_2] at x_h

The algorithm:

1- Input the initial interval of x where the root is expected
2- Calculate corresponding values of y_1 and y_2
3- Check for the sign difference between y-values
4- If y-values have the same sign, stop
5- Calculate the value of x_h at the middle of the interval
6- Check for the sign difference between the y-values of the first half interval
7- In case of opposite signs, let the new value of x_2 equal to x_h
8- In case of similar signs, let the new value of x_1 equal to x_h
9- If the values of y_1 and y_2 approaches zero, print the x-value and stop
10- Else repeat steps 5 to 10

Example 2-3

Find the roots of the following equation:

$$2x^2 - 5x + 3 = 0$$

Solution

Let's write the equation as a function

$$y = 2x^2 - 5x + 3$$

Let's "translate" the algorithm to a computer code to solve the equation:

```
x1= 0;              % first value of interval
x2= 1.3;            % end value of interval
y1 = 2*x1^2-5*x1+3;
y2 = 2*x2^2-5*x2+3;
if y1*y2 > 0        % test of entered interval
  disp('No roots exist within given interval')
```

```
  return                  % terminates the program
end
for i = 1:100             % assume 100 bisections are enough
  xh = (x1+x2)/2;         % calculation the half value
  y1 = 2*x1^2-5*x1+3;     % calculation of y1
  yh = 2*xh^2-5*xh+3;     % calculation of yh
  if y1*yh < 0            % check for the sign change in first half
    x2 = xh;              % let x2 be the midpoint value
  else                    % in case sign change in the second half
    x1 = xh;              % let x1 be the midpoint value
  end
  if abs(y1) < 1.0e-6     % condition of approach to solution
    disp('Root:')
    disp(x1)
    disp('Number of bisections:')
    disp(i)
    break           % terminate the for-loop
  end
end    % end of for-loop
```

The run of the program displays:

```
Root:
 1.00000
Number of bisections:
 22
```

Changing the initial interval to `x1 = 1.1; x2 = 2;` gives the second root:

```
Root:
 1.5000
Number of bisections:
 21
```

This could be the simplest working program of bisection method. Of course, some additional modifications can be added for better response for special input data. For example, if the user defines of the of the initial values (`x1` and `x2`) as 1.0 which is one of the roots, the program will terminated after showing the message "*No roots exist within given interval*." So, another condition should check whether `y1*y2` is equal to zero and if it is, another check should be made to find which one of initial values is a root of the equation. Although the code works, it contains a programming burden. The full equation is used four times to calculate `y1`, `y2` and `yh`. In MATLAB, there are two ways to define a one-line function so that it can be used in the program in a similar way to that in the mathematical writing:

1- inline function like the example:

```
>> y = inline('x*sin(x)','x')
```

2- anonymous function like

```
>> y = @(x)x*sin(x)
```

Despite the fact that the inline functions are still valid, they are replaced by the anonymous functions because of their flexibility especially in passing functions as parameters to other functions.

Another modification can be the use of input statement in defining initial values. It enables the user to input the values at the run-time and, consequently, the code will remain unchanged.

The following code contains the inline and input statements.

```
% Definition of an anonymous function y(x)
y = @(x)2*x^2-5*x+3;
% Reading x1 and x2 during the run
x1 = input('Enter x1: ');        % input x1 in the command window
x2 = input('Enter x2: ');        % input x2 in the command window
if y(x1)*y(x2) > 0               % the functions called here
  disp('No roots exist within given interval')
  return
end
for i = 1:100
  xh = (x1+x2)/2;
  if y(x1)*y(xh) < 0             % the functions called here
    x2 = xh;
  else
    x1 = xh;
  end
  if abs(y(x1)) < 1.0e-6         % the function called here
    disp('Root:')
    disp(x1)
    disp('Number of bisections:')
    disp(i)
    break
  end
end
```

Chapter 5

Numerical Integration

The main idea of limited integration is to computed the area under a curve of a function between two limits, *a* and *b*, by means of analytical integration rules.

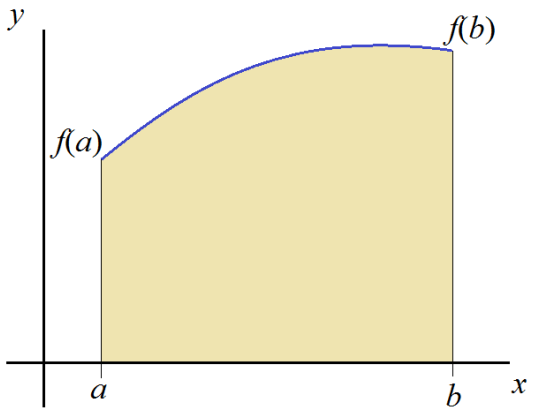

Figure 5-1. The area under the curve of $f(x)$ over the domain [a b].

The numerical integration, which is also called *quadrature*, is to compute the area under the curve of a given function by dividing the area to strips or regions that the area of each of them can be calculated by using simple mathematical rules then all areas are summed to get the total area under the curve from *a* to *b*.

The numerical integration is performed in one (or more) of the following cases:

a) The function does not have an explicit integral
b) A curve of empirical data is to be integrated
c) The integration is a part of a compound numerical method

The accuracy of the numerical integration method highly depends on the number of divisions that cover the area under the curve precisely. The method is more efficient when it yields more accurate result with lesser divisions.

Trapezoidal Rule

The trapezoidal rule is the most basic numerical integration method. As shown in the figure 5-2, the area under the curve is divided to vertical trapezoids having equal width, *h*, and the upper points coincide with the curve.

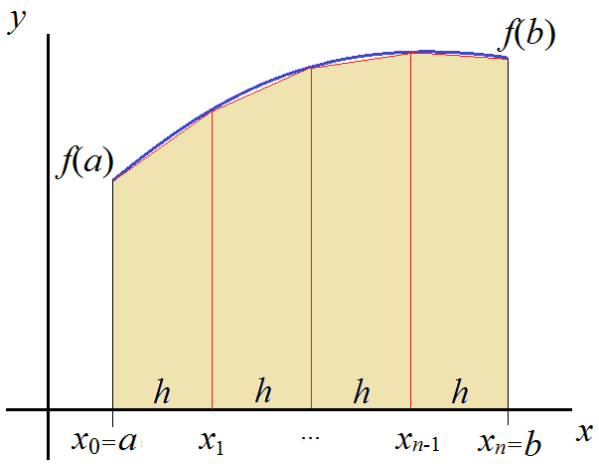

Figure 5-2. Dividing the area under the curve along the domain [*a b*] to trapezoids of equal width.

The area of the first section is

$$A = h[f(x_0) + f(x_1)]/2$$

The integral will be equal to the sum of the trapezoidal areas:

$$I = \frac{h}{2}[f(x_0) + f(x_1)] + \frac{h}{2}[f(x_1) + f(x_2)] + \cdots + \frac{h}{2}[f(x_{n-2}) + f(x_{n-1})] + \frac{h}{2}[f(x_{n-1}) + f(x_n)]$$

So,

$$I = h\left\{\frac{1}{2}[f(x_0) + f(x_n)] + f(x_1) + f(x_2) + \cdots + f(x_{n-2}) + f(x_{n-1})\right\}$$

or

$$I = h\left\{\frac{1}{2}[f(x_a) + f(x_b)] + f(x_1) + f(x_2) + \cdots + f(x_{n-2}) + f(x_{n-1})\right\}$$

Which can be programmed with in a single for-loop. Since the step size, *h*, is constant, the notation of x_i can be implemented in the code as $x_1 \rightarrow$ a+h, $x_2 \rightarrow$ a+2h and so on.

Example 5-1

Find the value of the integral

$$\int_0^{\frac{\pi}{2}} x \sin x \, dx$$

answer: The analytical integration value is 1.0.

Solution

The code can be made in the following steps:

Step 1: Define the function, the limits of the integration and the number of divisions.

```
f = @(x)x*sin(x);
a = 0;
b = pi/2;
n = 5;
```

Note that the value of π is predefined in MATLAB.

Step 2: Calculate the value of step size and initialize summation variable.

```
h = (b-a)/n
S = 0.5*(f(a)+f(b));
```

Step 3: construct a for-loop from i=1 to n-1 and add values of $f(a + ih)$ to summation variable inside the loop. After the end of the for-loop multiply the value of summation by h to get the value of the integral.

```
for i = 1:n-1
  S = S + f(a + i*h);
end
Intg = h * S;
```

Finally, by adding display statements, the code becomes as following:

```
f = @(x)x*sin(x);
a = 0;
b = pi/2;
n = 5;
h = (b-a)/n
S = 0.5*(f(a)+f(b));
for i = 1:n-1
  S = S + f(a + i*h);
end
Intg = h * S;
disp('The value of integral:')
disp(Intg)
```

The output:

```
The value of integral:
 1.0082654
```

To reduce the error, let's increase the number of divisions to 10. The result becomes

```
The value of integral:
 1.0020587
```

In case of `n = 100`:

```
The value of integral:
 1.0000205
```

So, using the trapezoidal rule, a 100 divisions are needed to get a solution accurate to the fourth decimal digit in this problem.

Simpson's Rules

Simpson's rules use weighing factors to calculate the integrals to improve the accuracy with lesser number of divisions. Unlike the trapezoidal rule which considers two points, x_i and x_{i+1}, to calculate a trapezoid's area, Simpson's rules use more than two points, i.e. multiple strips, in each turn. The values of $f(x)$ at the multiple points are adjusted by weighing factors to minimize the error.

Simpson's 1/3 rule:

This method calculates the area of two strips at a time, thus, three values of x are taken into account at each turn. To cover the whole domain exactly, the number of strips, n, should even, as shown in Figure 5-3.

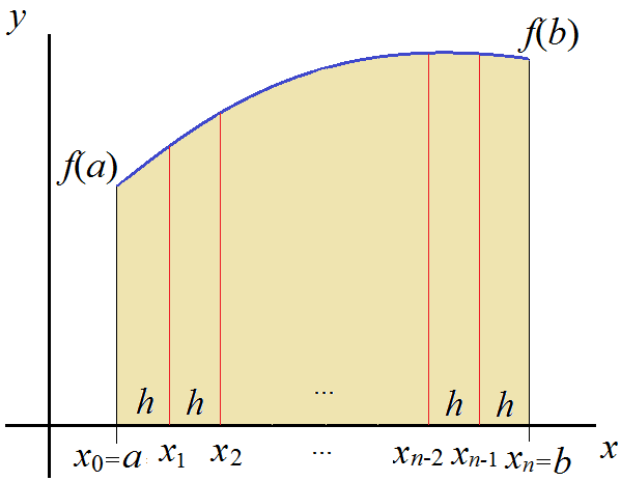

Figure 5-3. Dividing the area under the curve along the domain [a b] to even number of vertical sections of equal width.

The formula of Simpson's 1/3 rule for the first two strips is:

$$A = \frac{1}{3}h[f(x_0) + 4f(x_1) + f(x_2)]$$

Thus, the sum of all strip pairs will be

$$I = \frac{1}{3}h[f(x_0) + 4f(x_1) + f(x_2)] + \frac{1}{3}h[f(x_2) + 4f(x_3) + f(x_4)] + \cdots$$
$$+ \frac{1}{3}h[f(x_{n-4}) + 4f(x_{n-3}) + f(x_{n-2})]$$
$$+ \frac{1}{3}h[f(x_{n-2}) + 4f(x_{n-1}) + f(x_n)]$$

To simplify programming, the formula can be rewritten as

$$I = \frac{1}{3}h\{[f(x_0) + f(x_n)] + 4[f(x_1) + f(x_3) + \cdots + f(x_{n-3}) + f(x_{n-1})] + 2[f(x_2) + f(x_4) + \cdots + f(x_{n-4}) + f(x_{n-2})]\}$$

Finally, putting the terms of similar multipliers in summation form yields

$$I = \frac{1}{3}h\left\{[f(a) + f(b)] + \sum_{i=1,3,5}^{n-1} 4f(x_i) + \sum_{i=2,4,6}^{n-2} 2f(x_i)\right\}$$

With keeping in mind that *n* must be an even number for correct calculation procedure.

The programming procedure is very similar to that of the trapezoidal rule with one additional loop for the second summation. So, the first for-loop will be from 1 to *n*-1 with increment 2

```
S = f(a)+f(b);
for i = 1:2:n-1
  S1 = S1 + f(a + i*h);
end
```

and the second for-loop will be from 2 to n-2 with increment 2

```
for i = 2:2:n-2
  S = S + 2*f(a + i*h);
end
```

Example 5-2

Find the value of the integral of Example 5-1 by using Simpson's 1/3 rule.

Solution

The program:

```
f = @(x)x*sin(x);
a = 0;
b = pi/2;
n = 6;
h = (b-a)/n;
S = f(a)+f(b);
for i = 1:2:n-1
  S = S + 4*f(a + i*h);
end
for i = 2:2:n-2
  S = S + 2*f(a + i*h);
end
Intg = h/3 * S;
disp('The value of integral:')
disp(Intg)
```

The output:

```
The value of integral:
 0.9999206
```

If `n = 10`:

```
The value of integral:
 0.9999898
```

This value is more accurate than that of the trapezoidal rule to more decimal places at the same number of divisions.

At `n = 34`,

```
The value of integral:
 0.9999999
```

Simpson's 3/8 rule:

It is very similar to Simpson's 1/3 rule except that three strips are taken into account and, consequently, four points will be included at each time. The formula for the first three strips is

$$A = \frac{3}{8}h[f(x_0) + 3f(x_1) + 3f(x_2) + f(x_3)]$$

Thus, the sum of all strip pairs will be

$$I = \frac{3}{8}h[f(x_0) + 3f(x_1) + 3f(x_2) + f(x_3)] + \frac{3}{8}h[f(x_3) + 3f(x_4) + 3f(x_5) + f(x_6)] + \cdots + \frac{3}{8}h[f(x_{n-3}) + 3f(x_{n-2}) + 3f(x_{n-1}) + f(x_n)]$$

Thus,

$$I = \frac{3}{8}h\left\{[f(a) + f(b)] + \sum_{i=1,4,7}^{n-2} 3[f(x_i) + f(x_{i+1})] + \sum_{i=3,6,9}^{n-3} 2f(x_i)\right\}$$

The number of strips, *n*, must be a multiple of 3 and summation for-loop increment should be 3. The algorithm can be applied to the previous code by minor changes.

Example 5-3

Find the value of the integral of Example 5-1 by using Simpson's 3/8 rule.

Solution

The program:

```
f = @(x)x*sin(x);
a = 0;
b = pi/2;
n = 6; %n must be divisible by 3
h = (b-a)/n;
S = f(a)+f(b);
for i = 1:3:n-2
  S = S + 3*(f(a + i*h) + f(a + (i+1)*h));
```

```
end
for i = 3:3:n-3
  S = S + 2*f(a + i*h);
end
Intg = 3*h/8 * S;
disp('The value of integral:')
disp(Intg)
```

The output:

```
The value of integral:
 0.9998189
```

At `n = 12`:

```
The value of integral:
 0.9999889
```

At `n = 33`:

```
The value of integral:
 0.9999998
```

The results show that, for this problem, Simpson's 1/3 rule is slightly more accurate than 3/8 rule for equal or close numbers of divisions.

Double Integration

The double integration can be performed by using for-loops for the outer and inner integrals of the function from the lower to upper limits of each integral.

For Simpson's rules the following modifications should be done:

1- The loops should go through the extended form of the Simpson's integral. For example, for the 1/3 rule the utilized form in the algorithm is

$$I = \frac{1}{3}h[f_0 + 4f_1 + 2f_2 + 4f_3 + 2f_4 + \cdots + 2f_{n-4} + 4f_{n-3} + 2f_{n-2} + 4f_{n-1} + f_n]$$

and for the 3/8 rule it will be

$$I = \frac{3}{8}h[f_0 + 3f_1 + 3f_2 + 2f_3 + 3f_4 + \cdots + 3f_{n-4} + 2f_{n-3} + 3f_{n-2} + 3f_{n-1} + f_n]$$

where $f = f(x, y)$. When double integration is performed, the factor for each term of the first integration should multiplied by the factor of the term in the second integration.

2- Number of divisions for each integration can be different but both should satisfy Simpson's rules condition. For 1/3 rule, they should be even numbers and for 3/8 rule they should be multiple of 3. Accordingly, each integration can have different step size.

3- The final summation should by multiplied by the common factor of each integration. For Simpson's 1/3 rule it equals to

$$\frac{h_x}{3}\frac{h_y}{3} = \frac{h_x h_y}{9}$$

and for 3/8 rule it is equal to

$$\frac{3h_x}{8}\frac{3h_y}{8} = \frac{9h_x h_y}{64}$$

Example 5-4

Find the value of the integral I by using Simpson's 1/3 double integration.

$$I = \int_{-1}^{1}\int_{1}^{2}(x^2y + xy^2)dxdy$$

(The analytical solution is 1.0)

Solution

Step 1: Definition of the function, limits of the integration and step sizes.

```
f = @(x,y) x^2*y+x*y^2;    % the given function
ax = 1;                    % lower limit of x
bx = 2;                    % upper limit of x
ay = -1;                   % lower limit of y
by = 1;                    % upper limit of y
nx = 20;                   % number of divisions in x domain
ny = 40;                   % number of divisions in y domain
hx = (bx-ax) / nx;  % x step size
hy = (by-ay) / ny;  % y step size
```

Step 2: Initializing summation variable and construction of summation loops: the outer for *dy* and the inner for *dx* according to the given equation.

```
for i = 1 : ny+1           % loop of the outer integral
  if i==1 || i==ny+1
     p = 1;                % factor of the first and last terms
  elseif mod(i,2) == 0
      p = 4;               % factor for terms multiplied by 4
  else
      p = 2;               % factor for terms multiplied by 2
  end
  for j = 1 : nx+1         % loop of the inner integral
    if j==1 || j==nx+1
      q = 1;               % factor of the first and last terms
    elseif mod(j,2) == 0
      q = 4;               % factor for terms multiplied by 4
    else
      q = 2;               % factor for terms multiplied by 2
    end
    S = S + p*q * f(ax + (j-1)*hx, ay + (i-1)*hy);
  end    % end of the j-loop of the inner integral
end    % end of the i-loop of the outer integral
```

Step 3: Multiplication of summation value by the common factor and displaying the result.

```
Integral = hx*hy/9 * S;
```

```
disp('The value of the integral:')
disp(Integral)
```

The output:

```
The value of the integral:
 1.00000000000000
```

Here, the long format display is used just to show the level of accuracy of the method at 20 and 40 divisions for each domain.

MechTutor.com