Department of Physics
University of Dhaka
Computational Physics

If the function is polynomial then one can find the root of the function analytically or even numerically. But if the function is transcendental, the analytical solution is not possible one must need numerical solution.

Transcendental Function: A transcendental function is an analytic function that does not satisfy a polynomial equation, in contrast to an algebraic function. In other words, a transcendental function "transcends" algebra in that it cannot be expressed in terms of a finite sequence of the algebraic operations of addition, multiplication, and root extraction.

Examples of transcendental functions include the exponential function, the logarithm, and the trigonometric functions.

Example: $f(x) = x^{\pi}$, $f(x) = e^x$, $f(x) = x^x$, $f(x) = sin(x)$, $f(x) = log_e(x)$ etc.

We will discuss about the **Bisection method, Newton Raphson method and the Secant method.**

Bisection Method

- Given a function f(x) on floating number x and two numbers a and b such that $f(a)*f(b) < 0$ and f(x) is continuous in [a, b]. Here f(x) represents algebraic or transcendental equation. Find root of function in interval [a, b] (Or find a value of x such that f(x) is 0).

- Algebraic function are the one which can be represented in the form of polynomials like $f(x) = a_1x^3 + a_2x^2 + ... + e$ where $a_0, a_1, a_2, ...$ are constants and x is a variable. Transcendental function are non algebraic functions, for example $f(x) = sin(x) * x - 3$ or $f(x) = e^x + x^2$ or $f(x) = ln(x) + x$....

- The method is also called the interval halving method, the binary search method or the dichotomy method. This method is used to find root of an equation in a given interval that is value of 'x' for which $f(x) = 0$.

  The method is based on The Intermediate Value Theorem which states that if f(x) is a continuous function and there are two real numbers a and b such that $f(a) * f(b) < 0$. Then f(x) has at least one zero between a and b. If for a function $f(a) < 0$ and $f(b) > 0$) or $f(a) > 0$ and $f(b) < 0$, then it is guaranteed that it has at least one root between them.

  Assumption:

  1. f(x) is a continuous function in interval [a, b].
  2. $f(a) * f(b) < 0$

  Steps:

  1. Find middle point $c = (a + b)/2$.
  2. If $f(c) == 0$, then c is the root of the solution.
  3. Else $f(c)! = 0$

- ⋆ **If** value $f(a) * f(c) < 0$ then root lies between a and c. So we recur for a and c.
- ⋆ **Else If** $f(b) * f(c) < 0$ then root lies between b and c. So we recur b and c.
- ⋆ **Else** given function doesn't follow one of assumptions.

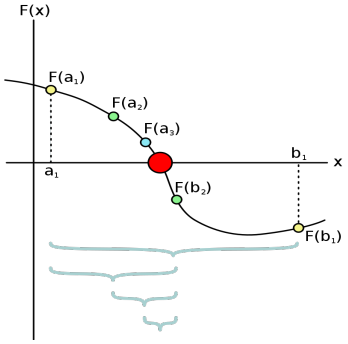


Figure 1: The bisection method.

- **Advantage of the Bisection method:**
  - ⋆ The bisection method is always convergent. Since the method brackets the root, the method is guaranteed to converge.
  - ⋆ As iterations are conducted, the interval gets halved. So one can guarantee the decrease in the error in the solution of the equation.
- **Drawbacks:**
  - ⋆ The convergence of bisection method is slow as it is simply based on halving the interval..
  - ⋆ If one of the initial guesses is closer to the root, it will take larger number of iterations to reach the root.
  - ⋆ If a function $f(x)$ is such that it just touches the x-axis such as $f(x) = x^2$ it will be unable to find the lower guess, x_l and upper guess, x_u , such that $f(x_l) * f(x_u) < 0$.
  - ⋆ d) For functions $f(x)$ where there is a singularity and it reverses sign at the singularity, bisection method may converge on the singularity.An example include $f(x) = 1/x$

```
// C++ program for implementation of Bisection Method for // solving equations
#include< iostream >
using namespace std;
#define EPSILON 0.01

    // An example function whose solution is determined using
// Bisection Method. The function is x^3 - x^2 + 2
double func(double x)
{ return x*x*x - x*x + 2; } // Prints root of func(x) with error of EPSILON
void bisection(double a, double b)
{ if (func(a) * func(b) >= 0)
{ cout << "You have not assumed right a and b";
return;
}
```

```
    double c = a;
while ((b − a) >= EPSILON)
{ // Find middle point
c = (a+b)/2;

    // Check if middle point is root
if (func(c) == 0.0)
break;

    // Decide the side to repeat the steps
else if (func(c) * func(a) < 0)
b = c;
else
a = c;
}
cout << "The value of root is : " << c; }
// Driver program to test above function
int main()
{
// Initial values assumed
double a =-200, b = 300;
bisection(a, b);
return 0;
}
```

Newton-Raphson Method

This is probably the most well known (also known as Newtons) method for finding function roots. The method can produce faster convergence by cleverly implementing some information about the function f. Unlike the bisection method, Newtons method requires only one starting value and does not need to satisfy any other serious conditions (except maybe one!). This is a one-point method. Newton-Raphson method is based on the principle that if the initial guess of the root of f(x)=0 is at x_i, then if one draws the tangent to the curve at $f(x_i)$, the point x_{i+1} where the tangent crosses the x-axis is an improved estimate of the root (Figure 2).

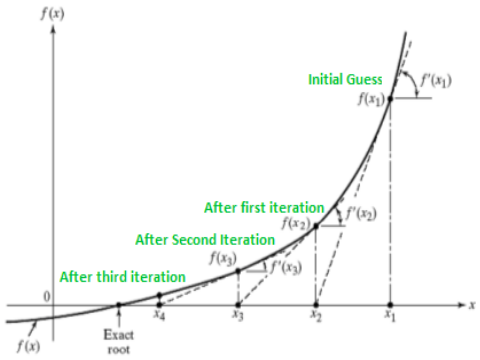


Figure 2: Geometrical illustration of Newton-Raphson method.

Let us start by presenting one of the ways that we can actually obtain such a numerical root finding scheme. Let a be an approximate root of $f(x) = 0$ and let $b = a + h$ be the corrected root so that $f(b) = 0$ By Taylor series exapnasion we have,
$f(b) = 0$
$f(a + h) = 0$
$f(a) + hf'(a) + \frac{h^2}{2!}f''(a) + ... = 0$
Neglecting the second and higher order derivative, we have

$$f(a) + hf'(a) = 0$$

whice give

$$h = -\frac{f(a)}{f'(a)}$$

Therefore,

$$b = a - \frac{f(a)}{f'(a)}$$

The next point,

$$c = b - \frac{f(b)}{f'(b)}$$

$$d = c - \frac{f(c)}{f'(c)}$$

In general

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

That is if we choose the initial guess x_0 close enough to the root of the function f(x) we are essentially guaranteed that we will find the true root!
In short the method is implemented as follows:

- Obtain a starting guess x_0 and find a formula for the derivative f' of f.
- Produce the next iterate x_n through the following equation

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

- Stop if any of the following occurs: required tolerance is reached, maximum iterates exceeded, $f'(x_{n-1}) = 0$.

Advantage: Fast converging.
Disadvantages:

- Calculating the required derivative itself $f'(x_n)$ for every iteration may be a costly task for some functions f.
- May not produce a root unless the starting value x_0 is close to the actual root of the function.
- May not produce a root if for instance the iterations get to a point x_{n-1} such that $f'(x_{n-1}) = 0$. Then the method fails!

Find the root of the equation $f(x) = x^3 - x^2 + 2$ by using Newton-Rapson Method.

```
#include< iostream >
#define EPSILON 0.001
using namespace std;

    double func(double x)
{
return x*x*x - x*x + 2;
}

    double derivFunc(double x)
{
return 3*x*x - 2*x;
}

    void newtonRaphson(double x)
{
double h = func(x) / derivFunc(x);
while (abs(h) >= EPSILON)
{
h = func(x)/derivFunc(x);
x = x - h;
}
cout << "The value of the root is : " << x;
}
int main()
{
double x0 = -20; // Initial values assumed
newtonRaphson(x0);
return 0;
}
```

The Secant Method

One of the main drawbacks of the Newton-Raphson method is that you have to evaluate the derivative of the function. With availability of symbolic manipulators such as Maple, Mathcad, Mathematica and Matlab, this process has become more convenient. However, it is still can be a laborious process. To overcome this drawback, the derivative, $f'(x)$ of the function, f(x) is approximated as

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

Substituting $f'(x_n)$ in the equation of the N-R namely in

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

We have,

$$x_{n+1} = x_n - \frac{f(x_n) * (x_n - x_{n-1})}{f(x_n) - f(x_{n-1})}$$

The above equation is called the Secant method. This method now requires two initial guesses, but unlike the bisection method, the two initial guesses do not need to bracket the root of the equation. The Secant method may or may not converge, but when it converges, it converges faster than the bisection method. However, since the derivative is approximated, it converges slower then Newton-Raphson method. Let us examine one example, if we let the first guess $x_{n-1} = x1$, and $x_n = x2$ and $x_{n+1} = x0$ then the above formula can be written as:

$$x0 = \frac{f(x2) * (x1) - x2 * f(x1)}{f(x2) - f(x1)}$$

Now, Let us solve the equation $f(x) = x^3 + x - 1$ by using secant method.

```
#include < iostream >
#include < cmath >
using namespace std;
double f(double x)
{
double f = pow(x, 3) + x - 1;
return f;
}
void secant(double x1, double x2, double E)
{
double n = 0, xm, x0, c;
if (f(x1) * f(x2) < 0) {
do {
x0 = (x1 * f(x2) - x2 * f(x1)) / (f(x2) - f(x1));
c = f(x1) * f(x0);
x1 = x2;
x2 = x0;
n++;
if (c == 0)
break;
xm = (x1 * f(x2) - x2 * f(x1)) / (f(x2) - f(x1));
} while (fabs(xm - x0) >= E);
cout << "Root of the given equation=" << x0 << endl;
cout << "No. of iterations = " << n << endl; } else
cout << "Can not find a root in the given inteval";
}
int main()
{

    double x1 = 0.0, x2 = 1.0, E = 0.0001;
secant(x1, x2, E);
return 0;
}
```

Exercise:

1. Find the root of the $f(x) = xe^x - 1 = 0$ using all the three methods, corrected up to three decimal places.
2. Find the root of the $cos(x) - xe^x = 0$ using all the three methods, corrected up to three decimal places.

3. Determine the smallest positive root of $x - e^{-x}$ up to three significant figure.
4.Find a root of $xsinx + cosx = 0$
5. Find the square root of 18 using Newton-Raphson method (Hints: you need to solve $x^2 - n = 0$ where n is the number and x is its square root.)

Numerical Differentiation and Integration
Computational Physics
Department of Physics, University of Dhaka

1. Numerical Differentiation:

Finite Differences
Finite differences are a way of approximating derivatives. The derivative is the limit of $(\Delta f/\Delta x)$ as $\Delta x \to 0$. A finite difference approximation is, roughly speaking, $(\Delta f/\Delta x)$ evaluated for a small value of Δx.
Several variations are possible. Use h in place of Δx and let

$$f_{i-1} = f(x-h)$$

$$f_i = f(x)$$

$$f_{i+1} = f(x+h)$$

Then we have the following approximations to the derivative$f^{(x)}$:

1. Forward finite difference: $f'(x) = \frac{f_{i+1}-f_i}{h} + O(h) \approx (f_{i+1} - f_i)/h$
2. Backward finite difference: $f'(x) = \frac{(f_i - f_{i-1})}{h} + O(h) \approx (f_i - f_{i-1})/h$
3. Central finite difference: $f'(x) = \frac{(f_{i+1} - f_{i-1})}{2h} + O(h^2) \approx (f_{i+1} - f_{i-1})/2h$

Example
Consider$f(x) = x^3$. Its derivative at $x = 5$ is $3 \times 5^2 = 75$. Take $h = 0.1$. Then
Forward difference $\frac{\left[(5+0.1)^3\right]}{(0.1)} = 76.51$
Backward difference $\frac{\left[5^3-(5-0.1)^3\right]}{(0.1)} = 73.51$
Central difference $\frac{\left[(5+0.1)^3-(5-0.1)^3\right]}{(2\times 0.1)} = 75.01$
Central difference is clearly the best estimate.
Errors in the approximation: The forward and the backward difference formulas have first order i.e. $O(h)$ error while the central difference formula has second order i.e. $O(h^2)$ error. These can be seen by expanding the functions at $x + nh$ and $x - nh$ in Taylor series, where n is a positive integer:

$$f(x+nh) = f(x) + nhf'(x) + \frac{(nh)^2}{(2!)}f''(x) + \cdots$$

$$f(x-nh) = f(x) - nhf'(x) + \frac{(nh)^2}{(2!)}f''(x) - \cdots$$

Then we get: $f_{i+1} - f_{i-1} = f(x+h) - f(x-h) = f(x) + xf'(x) + \frac{h^2}{(2!)}f''(x) + \cdots - \left(f(x) - hf'(x) + \frac{h^2}{(2!)f'(x)} - \cdots\right) = 2hf'(x) + \frac{2h^3}{(3!)f'(x)} + \cdots = 2hf'(x) + O(h^3)$

$$\Rightarrow f'(x) = \frac{(f(x+h) - f(x-h))}{2h} + O\left(h^2\right)$$

Higher order approximations to the first derivative can be obtained by using more Taylor series, more terms in the Taylor series, and cleverly weighting the various expansions in a sum. For example, we get,

1. Forward difference approximation with second order error:

$$f'(x) = \frac{[-f(x+2h) + 4f(x+h) - 3f(x)]}{2h} + O\left(h^2\right)$$

The formula can be derived as follows:

- All discrete approximations to derivatives are linear combinations of functional values at the nodes

$$f_i^p = \frac{a_\alpha f_\alpha + a_\beta f_\beta + \ldots\ldots\ldots\ldots\ldots + a_\lambda f_\lambda}{h^p} + E$$

- The total number of nodes used must be at least one greater than the order of differentiation to achieve minimum accuracy O(h).
- To obtain better accuracy, you must increase the number of nodes considered.
- For central difference approximations to even derivatives, a cancellation of truncation error terms leads to one order of accuracy improvement.

We need to derive a formula for first order derivative (f_i') with second order accuracy ($O(h^2)$):

- First derivative with $O(h)$ accuracy $\Rightarrow$ the minimum number of nodes is 2
- First derivative with $O(h^2)$ accuracy $\Rightarrow$ the minimum number of nodes is 3

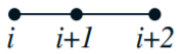


Figure 1: One need 3 nodes for second order accuracy.

- The first forward derivative can therefore be approximated to $O(h^2)$ as:

$$\left(\frac{df}{dx}\right)_{x=x_i} - E = \frac{\alpha_1 f_i + \alpha_2 f_{i+1} + \alpha_3 f_{i+2}}{h}$$

- The Taylor series expansion about x_i give:

$$f_i = f_i$$

$$f_{i+1} = f_i + hf_i' + \frac{h^2}{2}f_i'' + \frac{h^3}{6}f_i''' + O(h^4)$$

$$f_{i+2} = f_i + 2hf_i' + 4\frac{h^2}{2}f_i'' + \frac{4h^3}{3}f_i''' + O(h^4)$$

- Substituting into our assumed form of f_i' and re-arranging

$$\frac{\alpha_1 f_i + \alpha_2 f_{i+1} + \alpha_3 f_{i+2}}{h} = \frac{\alpha_1 + \alpha_2 + \alpha_3}{h}f_i + (\alpha_2 + 2\alpha_3)f_i' + (\frac{\alpha_2}{2} + 2\alpha_3)hf_i''$$

$$+(\frac{1}{6}\alpha_2 + \frac{4}{3}\alpha_3)h^2 f_i''' + O(h^3)$$

- In order to have our desired accuracy, the coefficient of f_i' must equal unity and the coefficient of f_i and f_i'' must vanish, which gives

$$\frac{\alpha_1 + \alpha_2 + \alpha_3}{h} = 0$$
$$\alpha_2 + 2\alpha_3 = 1$$
$$(\frac{\alpha_2}{2} + 2\alpha_3)h = 0$$

Solving this simultaneous equation gives $\alpha_1 = -3/2, \alpha_2 = 2, \; and \; \alpha_3 = -1/2$

- Thus the equation now becomes

$$\frac{-\frac{3}{2}f_i + 2f_{i+1} - \frac{1}{2}f_{i+2}}{h} = (0)f_i + (2-1)f_i' + (0)f_i'' + \left(\frac{1}{6}.2 - \frac{4}{3}.\frac{1}{2}\right)h^2 f_i''' + O(h^3)$$

This gives,

$$f_i' = \frac{-3f_i + 4f_{i+1} - f_{i+2}}{2h} + \frac{1}{3}h^2 f_i''' + O(h^3)$$

- So the formula become

$$f_i' = \frac{-3f_i + 4f_{i+1} - f_{i+2}}{2h} + E$$

Where $E = \frac{1}{3}h^2 f_i'''$
Which can be written as

$$f'(x) = \frac{[-f(x+2h) + 4f(x+h) - 3f(x)]}{2h} + O\left(h^2\right)$$

Similarly one can derive the backward and central difference formula for higher order accuracy;

2. Backward difference approximation with second order error:

$$f'(x) = \frac{[3f(x) - 4f(x-h) + f(x-2h)]}{(2h)} + O\left(h^2\right)$$

3. Centered difference approximation with fourth order error:

$$f'(x) = \frac{[-f(x+2h) + 8f(x+h) - 8f(x-h) + f(x-2h)]}{(12h)} + O\left(h^4\right)$$

Higher Derivatives:
Higher order derivatives can be approximated in the same way by using Taylor expansions. For example:

1. Forward difference approximation to $f''(x)$ is:

$$f''(x) = \frac{[f(x+2h) - 2f(x+h) + f(x)]}{(h^2)} + O(h)$$

One can derive this formula as before by considering 3 nodes for O(h) order accuracy of f'':

$$f_i'' - E = \frac{\alpha_1 f_i + \alpha_2 f_{i+1} + \alpha_3 f_{i+2}}{h^2}$$

- Expanding in Taylor series and rearranging we have

$$\frac{\alpha_1 f_i + \alpha_2 f_{i+1} + \alpha_3 f_{i+2}}{h^2} = \frac{\alpha_1 + \alpha_2 + \alpha_3}{h^2} + \left(\frac{\alpha_2 + 2\alpha_3}{h}\right) f_i' + \frac{1}{2}(\alpha_2 + 4\alpha_3) f_i'' + (\alpha_2 + 8\alpha_3)\frac{h}{6} f_i''' + O(h^2)$$

- In order to compute f'' we must have:

$$\frac{\alpha_1 + \alpha_2 + \alpha_3}{h^2} = 0$$

$$\left(\frac{\alpha_2 + 2\alpha_3}{h}\right) = 0$$

$$\frac{1}{2}(\alpha_2 + 4\alpha_3) = 1$$

  Give $\alpha_1 = 1$, $\alpha_2 = -2$, and $\alpha_3 = 1$

- Therefore

$$f_i'' = \frac{f_{i+2} - 2f_{i+1} + F_i}{h^2} + E$$

  Where $E = -h f_i'''$

The higher order accuracy can be obtain subsequently by increasing number of nodes:
$f_i'' \rightarrow$ Require 3 nodes for order of O(h) accuracy.
$f_i'' \rightarrow$ Require 4 nodes for order of $O(h^2)$ accuracy.
$f_i'' \rightarrow$ Require 5 nodes for order of $O(h^3)$ accuracy.
$f_i'' \rightarrow$ Require 6 nodes for order of $O(h^4)$ accuracy.

2. Centered difference approximations are

$$f''(x) = \frac{[f(x+h) - 2f(x) + f(x-h)]}{(h^2)} + O\left(h^2\right)$$

$$f''(x) = \frac{[-f(x+2h) + 16f(x+h) - 30f(x) + 16f(x-h) - f(x-2h)]}{(12h^2)} + O\left(h^4\right)$$

A list of finite difference approximations is given below:

1. First order:

$$F\ f'(x) = \left(\frac{1}{h}\right)[f(x+h) - f(x)]$$

$$B\ f'(x) = \left(\frac{1}{h}\right)[f(x) - f(x-h)]$$

2. Second order:

$$F\ f'(x) = \left(\frac{1}{2h}\right)[-3f(x) + 4f(x+h) - f(x+2h)]$$

$$f''(x) = \left(\frac{1}{h^2}\right)[f(x) - 2f(x+h) + f(x+2h)]$$

$$C\ f'(x) = \left(\frac{1}{2h}\right)[f(x+h) - f(x-h)]$$

$$f''(x) = \left(\frac{1}{h^2}\right)[f(x+h) - 2f(x) + f(x-h)]$$

$$B\ f'(x) = \left(\frac{1}{2h}\right)[3f(x) - 4f(x-h) + f(x-2h)]$$

$$f''(x) = \left(\frac{1}{h^2}\right)[f(x) - 2f(x-h) + f(x-2h)]$$

Etc.

Don't let *h* be too small:

The smaller the h the better the finite differences estimate. That's certainly true "early on", but at some point you run into trouble. The answer is lack of precision. We only have 16 digits of precision in our machine (i.e. the computer).

To see this limitation, let us consider the function $\sin x$ at $x = \pi/10$. The values of $\sin \pi/10$ and $\sin\left(\frac{\pi}{10} + 10^{-10}\right)$ agree to 10 digits:

$$\sin\left(\frac{\pi}{10}\right) = 0.3090169943749474$$

$$\sin\left(\frac{\pi}{10} + 10^{-10}\right) = 0.3090169944700531$$

So when we subtract one from the other and divide by $h = 10^{-10}$, we get

$$\frac{\left[\sin\left(\frac{\pi}{10} + 10^{-10}\right) - \sin\left(\frac{\pi}{10}\right)\ \right]}{\left(10^{-10}\right)} = \frac{[0.3090169944700531 - 0.3090169943749474]}{\left(10^{-10}\right)} = 0.951057$$

Hence we lose 10 digits of precision! So the best we can hope for is 6 digits of accuracy. Hence, the best h is chosen by a trade-off between accuracy of the estimate and the available precision.

A simple function can be differentiate very easily at one point very easily by using any of the above method. For example a simple central difference method as bellow:

```
    #include<iostream>
#include<cmath>
using namespace std;

    double f(double x){
return 3*x*x+5*x;}

    double deff(double x, double h){
double dy=f(x+h)-f(x-h);
return dy/(2*h);}
int main(){
cout<<deff(1,0.1)<<endl;
return 0;}
```

Example-1

The following program computes the sine function at discrete points, stores the results in an array, then computes numerically the first derivative of the function. The results are compared to the exact form.

```
/* numdiff1.cpp */
#include <iostream>
#include<fstream>
#include<cmath>
using namespace std;
const char *FILENAME = "numdiff1.txt"; //*FILENAME is an output stream
#define N_POINTS 300 /* Array size */
int main()
{
    int i;
    float angle[N_POINTS], f[N_POINTS], f1_3[N_POINTS];
    float d_angle = 0.0;
    ofstream fout(FILENAME); //create output object associated with file
/* —compute the data— */
    d_angle = 2.0*M_PI / (float) (N_POINTS-1);
/* M_PI is defined in the math library to be equal to pi */
    for(i=0; i< N_POINTS; i++) {
    angle[i] = i*d_angle;
    f[i] = sin(angle[i]);
    }
/* — 2-point derivative at beginning and end of the lattice — */
    f1_3[0] = ( f[1] - f[0] ) / d_angle;
    f1_3[N_POINTS-1] = (f[N_POINTS-1] - f[N_POINTS-2]) / d_angle;
/* — 3-point derivative everywhere else ———— */
    for(i = 1; i < N_POINTS-1; i++ )
    {
    f1_3[i] = ( f[i+1] - f[i-1] ) / (2*d_angle);
    }
/* —- output results and error—- */
    for ( i = 0; i < N_POINTS; i++)
    {
    cout<< angle [i] << " " << f [i] << " " << f1_3 [i] << " " << fabs(f1_3 [i] – cos(angle
[i])) << endl;
/* —— write to file — */
    fout<< angle[i] << " " << f[i] << " "<< f1_3 [i] << " " << fabs(f1_3[i] – cos(angle[i]))
<<endl;
    }
    fout.close(); /*close file */
    return 0;
}
```

Plotting Data and Saving the Plot

To plot these data using gnuplot, use the following command in the **terminal** window:

gnuplot

This will start the gnuplot program. Within the gnuplot prompt, i.e. right after "gnuplot>", that you will see by default, write the following commands:

plot 'numdiff1.txt' using ($1) : ($2) with line

Close the plot that should pop up in a window. Then to save the plot in a "postscript" type of file using the following commands:
set terminal postscript enhanced colot
set output 'yourname_diff1_err.ps'
plot 'numdiff1.txt' u ($1) : ($2) w l
set terminal X11
quit
The last command will return you back to the shell terminal window by quitting gnuplot. ($1) in the above refers to the values in the first column of the file while ($2) refers to the second column values etc. and the above plots ($2) vs ($1). Using ($4) vs ($1) saves the postscript file of the error.
Note that the error is relatively small. Note also that there is a systematic error introduced by the 3-point form which causes oscillations in the error function having the same wavelength as that of the original (sine) function.

Difference between double and float

Instead of using float type of variables, change the code above to use double type of variables (replace float by double at all places) and redo the error plot. Depending on machines, you should get slightly different plots.

Example-2 Round-off errors

One might think that using the higher-order derivative forms would always be better; this is in general the case, but with an important caveat! These forms may involve large cancellations in the numerator as h becomes small, thereby producing bad results due to round-off. The following code illustrates this point. It computes the derivative of $\sin x$ at 45 degrees.

```
/* numdiff2.cpp */
#include <iostream>
#include <fstream>
#include <cmath>
using namespace std;
int main()
{
    float x = M_PI/4.0, f1_3, f1_5, f1_e;
    float d_angle;
    int i;
    d_angle = 5.0;
    cout << " \n\nTable of errors in 3- and 5- point derivatives\n\n" ;
    cout << " angle 3-point 5-point \n" ;
for (i =1; I < 10; i++)
{
    d_angle = d-angle / 10.0;
    f1_3 = (sin(x+d_angle) – sin(x-d_angle)) / (2.0*d_angle);
    f1_5 = (sin(x-2*d_angle) – 8*sin(x-d_angle) + 8*sin(x+d_angle) – sin(x+2*d_angle))/ (12.0*d_angle);
    f1_e = cos(x) ;
    cout<< d_angle<< " "<<f1_3 – f1_e<< " "<<f1_5 – f1_e<<endl;
    }
    return 0;
}
```

Run the above code to verify the loss of precision. The minimum error is for d_angle = 0.005 for the 3-point formula, and 0.05 for the 5-point formula. The error becomes larger for smaller

intervals due to arithmetic errors. Note the use of single precision in the code to make effect more evident!

Richardson Extrapolation:

We have seen how two different expressions can be combined to eliminate the leading error term and thus yield a more accurate expression. It is also possible to use a single expression to achieve the same goal. This general technique is due to L. F. Richardson, a meteorologist who pioneered numerical weather prediction in the 1920s.

Let's start with the central difference formula, and imagine that we have obtained the usual approximation for the derivative,

$$f'(x) = \frac{(f(x+h) - f(x-h))}{2h} - \frac{h^2}{6} f'''(x) + ..$$

Using a different step size we can obtain a second approximation to the derivative. Then using these two expressions, we can eliminate the leading term of the error. In practice, the second expression is usually obtained by using an h twice as large as the first, so that

$$f'(x) = \frac{(f(x+2h) - f(x-2h))}{4h} - \frac{4h^2}{6} f'''(x) + ..$$

Dividing this expression by 4 and subtracting it from the previous one eliminates the error! Well, actually only the leading term of the error is eliminated, but it still sounds great! Solving for $f'(x)$, we obtain

$$f'(x) = \frac{(f(x-2h) - 8f(x-h) + 8f(x+h) - f(x+2h))}{12h} + O(h^4)$$

a 5-point central difference formula with error given by $E = \frac{h^4}{30} f^{iv}(x)$
Of course, we can do the same thing with other derivatives: using the 3-point expression of Equation

$$f''(x) = \frac{[f(x+h) - 2f(x) + f(x-h)]}{(h^2)} + O\left(h^2\right)$$

We can easily derive the 5-point expression:

$$f''(x) = \frac{[-f(x+2h) + 16f(x+h) - 30f(x) + 16f(x-h) - f(x-2h)]}{(12h^2)} + O\left(h^4\right)$$

With error $E = \frac{h^4}{90} f^{vi}(x) + ...$

Now, there are two different ways that Richardson extrapolation can be used. The first is to obtain "new" expressions, as we'vejust done, and to use these expressions directly. Be forewarned, however, that these expressions can become rather cumbersome. The other is indirect computational scheme, which we are going to discuss bellow. This is the same scheme as before but here we execute this numerically.

- Let $D_1(h)$ be the approximation to the derivative obtained from the 3-point central difference formula with step size h, and imagine that both $D_1(2h)$ and $D_1(h)$ have been calculated.
- Since the error goes as the square ofthe step size, $D_1(2h)$ must contain four times the error contained in $D_1(h)$.

- The difference between these two approximations is then three times the error of the second. But the difference is something we can easily calculate, so in fact we can calculate the error.

- $D_2(h)$, is then obtained by simply subtracting this calculated error from the second approximation,

$$D_2(h) = D_1(h) - \left[\frac{D_1(2h) - D_1(h)}{2^2 - 1}\right]$$

$$D_2(h) = \frac{4D_1(h) - D_1(2h)}{3}$$

- Of course, $D_2(h)$ is not the exact answer, since we've only accounted for the leading term in the error. Since the central difference formulas have error terms involving only even powers of h, the error in $D_2(h)$ must be $O(h^4)$

- $D_2(2h)$ contains 2^4 times as much error as $D_2(h)$, and so this error can be removed to yield an even better estimate of the derivative,

$$D_3(h) = D_2(h) - \left[\frac{D_2(2h) - D_2(h)}{2^4 - 1}\right]$$

$$D_3(h) = \frac{16D_2(h) - D_2(2h)}{15}$$

- This processes can be continued indefinitely, with each improved estimated given by

$$D_{i+1}(h) = D_i(h) - \left[\frac{D_i(2h) - D_i(h)}{2^{2i} - 1}\right]$$

$$D_{i+1}(h) = \frac{2^{2i}D_i(h) - D_i(2h)}{2^{2i} - 1}$$

```
    #include <iostream>
#include <cmath>
#include <iomanip>

    using namespace std;
double f( double x)
{ double f;
f=sin(x);
return f;
}
void Derivative(double x, int n, double h, double D[10][10])

    {

    int i, j;

    for (i=0; i<n; i++)

    {
```

```
    D[i][0]=(f(x+h)-f(x-h))/(2*h);

    for (j=0; j<=(i-1); j++)

    {
    D[i][j+1]=D[i][j]+(D[i][j]-D[i-1][j])/(pow(4.0,double(j+1))-1);

    }

    h=h/2;

    }

    }

    int main()
    {

    double D[10][10];

    int n=10, digits=5;

    double h=1, x=0;

    Derivative(x, n, h, D);
cout.setf(ios::fixed );

    cout.setf(ios::showpoint);

    cout << setprecision(digits) << endl;
for(int i=0; i<n; i++)
{
for(int j=0;j<i+1;j++)

    { cout << setw(digits+2) << D[i][j]<< " ";
}

    cout << endl;
}
cout.unsetf(ios::fixed);
cout.unsetf(ios::showpoint);

    return 0;

    }
```

1. **Numerical Integration**

There are two main reasons for you to need to do numerical integration:

1. Analytical integration may be impossible or infeasible or,

2. You may wish to integrate tabulated data rather than known functions.

In this section we outline the main approaches to numerical integration. Which one is preferable depends on the results required, and in part on the function or data to be integrated.

1. (a) **Constant Rule**

Perhaps the simplest form of numerical integration is to assume that the function $f(x)$ is constant over the interval being integrated. Clearly this is not going to be a very accurate method of integrating, and indeed leads to an ambiguous result, depending on whether the constant is selected from the lower or the upper limit of the integral.
Integration of a Taylor series expansion of $f(x)$ shows the error in this approximation to be:

$$I = \int_{x_0}^{x_0+\Delta x} f(x)\,dx = \int_{x_0}^{x_0+\Delta x} \left[f(x_0) + f'(x_0)(x-x_0) + \frac{f''(x_0)}{2}(x-x_0)^2 + \cdots \right] dx =$$

$$f(x_0)\Delta x + \frac{1}{2}f'(x_0)(\Delta x)^2 + \frac{1}{6}f''(x_0)(\Delta x)^3 + \cdots = f(x_0)\Delta x + O\left((\Delta x)^2\right)$$

In the constant rule, the integral I is approximated as

$$I \approx f(x_0)(x_0 - \Delta x - x_0) = f(x_0)\Delta x$$

if the constant is taken from the lower limit (i.e. x_0). Similar analysis shows that
$I = f(x_0+\Delta x)\Delta x$ if the constant is taken from the upper limit (i.e. $x_0+\Delta x$. In both cases the error is $O\left((\Delta x)^2\right)$, with the coefficient being derived from $f'(x_0)$ or $f'(x_0+\Delta x)$. Clearly we can do much better than this, and as a result, this rule is not used in practice.

1. (a) **Trapezium Rule**

Consider the Taylor series expansion of $f(x)$ around x_0, integrated from x_0 to $x_0+\Delta x$:

$$I = \int_{x_0}^{x_0+\Delta x} \left[f(x_0) + f'(x_0)(x-x_0) + \frac{f''(x_0)}{2}(x-x_0)^2 + \cdots \right] dx =$$

$$f(x_0)\Delta x + \frac{1}{2}f'(x_0)(\Delta x)^2 + \frac{1}{6}f''(x_0)(\Delta x)^3 + \cdots =$$

$$\left[\frac{1}{2}f(x_0) + \frac{1}{2}\left(f(x_0) + f'(x_0)\Delta x + \frac{1}{2}f''(x_0)(\Delta x)^2 + \cdots\right) - \frac{1}{12}f''(x_0)(\Delta x)^2 + \cdots\right]\Delta x =$$

$$\frac{1}{2}\left[f(x_0) + f(x_0+\Delta x)\right]\Delta x + O\left((\Delta x)^3\right) \approx \frac{1}{2}\left[f(x_0) + f(x_0+\Delta x)\right]\Delta x$$

This approximation represented by $I \approx \frac{1}{2}\left[f(x_0) + f(x_0+\Delta x)\right]\Delta x$, is called the trapezium rule, based on its geometric interpretation of approximating the area under the curve by a trapezium. It is exact for polynomials up to and including degree 1, i.e. $f(x) = ax + c$.
Compound Trapezium Rule
The error in the Trapezium Rule is proportional to $(\Delta x)^3$. Thus if we were to halve Δx, the error would be decreased by a factor of eight. However, the size of the domain would be halved, thus requiring the Trapezium Rule to be evaluated twice and the contributions summed. The net result is the error decreasing by a factor of four $\left(\frac{1}{\left(2\times\frac{1}{8}\right)}\right)$ rather than eight. The Trapezium

rule used in this manner is sometimes called the Compound Trapezium Rule, but more often simply the Trapezium Rule.
Suppose we need to integrate from $x_0 = a$ to $x_N = b$. We shall subdivide this interval into N sub-intervals of size $\Delta x = \frac{(b-a)}{N} = (x_N - x_0)/N$. The Compound Trapezium Rule approximation to the integral is therefore (noting that $x_N = x_0 + N\Delta x$):

$$I = \int_{x_0}^{x_N} f(x)\,dx = \sum_{i=0}^{N-1} \int_{x_0+i\Delta x}^{x_0+(i+1)\Delta x} f(x)\,dx \approx \frac{\Delta x}{2} \sum_{i=0}^{N-1} \left[f(x_0 + i\Delta x) + f(x_0 + (i+1)\Delta x)\right] =$$

$$\frac{\Delta x}{2}\left[f(x_0) + 2f(x_0 + \Delta x) + 2f(x_0 + 2\Delta x) + \cdots \qquad + 2f(x_0 + (N-1)\Delta x + f(xN))\right]$$

Note that, while the error in each step is $O\left((\Delta x)^3\right)$ (from the Trapezium Rule), the cumulative error is N times this or $O\left((\Delta x)^2\right) \sim O\left(N^{-2}\right)$ which is bigger.
Example-3 Trapezoidal Rule
Consider the simple integrator that makes use of the Compound Trapezium Rule:

```
/* Implementation of Trapezoidal Rule of Integration */
#include<iostream>
using namespace std;
double integtrapzd(double (*func)(double), double a, double b, int n)
{
    double x=0.0, sum=0.0, del=0.0;
    int i=0, j=0;
    if (n==1)
    { return 0.5*(b-a)*( (*func) (a) + (*func) (b) ; }
    else if (n>1)
    {
    del= (b-a)/n;
    sum += (*func) (a);
    x = a + del;
    for(j=1; j<n; j++, x += del) sum += 2.0 * (*func) (x);
    sum += (*func) (b);
    return sum*0.5*del;
    }
else
    {
    cout<< "Number of interval numint has to be >= 1" << endl;

    exit(1);
    }
}
double myfunc1(double x) { return x*x; }
int main()
{
    double res=0.0, low=0.0, up=1.0;
    int numint=10;
    res = integtrapzd(myfunc1, low, up, numint);
    cout<< "res: "<<res<<endl;
    return 0;
```

```
}
```
Example-4 Error in Trapezoidal Rule

Consider the following modifications in the code for observing the dependence of the error on the number of sub-intervals numint:

```
...
#include<iostream>
#include<cmath>
#include<fstream>
...
const char *FILENAME = "trapzderror.txt";
...
double integtrapzd(double (*func) (double), double a, double b, double n)
{
    double x=0.0, sum=0.0, del=0.0; int i=0, j=0;
if (n==1)

    { return 0.5*(b-a)*( (*func) (a) + (*func) (b)) ; }
else if (n>1)
{

del = (b-a) / n;
    sum += (*func) (a);
x = a + del;
for(j=1; j<n; j++, x += del){
x+=del;
sum += 2.0 * (*func) (x);}

    sum += (*func) (b);

    return sum*0.5*del;
}

    else { cout<< "Number of interval numint has to be >= 1" << endl; }
}

}
double myfunc2 (double x) { return x*x*x ; }
int main()
{
    double res=0.0, low=0.0, up=1.0, error=0.0;
    double i=1.0, numint=10.0;
...
ofstream fout(FILENAME) ; //create output object associated with the file
for(i=1.0; i<1000000000.0; i*=10.0) //up to 10^8
{
res = integtrapzd(myfunc2, low, up, i);
    cout<< "res: " << res << "error" << res -0.25 <<endl;
/* — Write to file —*/
    fout<< i << " " << res << " " << res -0.25 << endl;
```

```
    }
    return 0;
}
```

Plot the error file trapzderror.txt using the gnuplot program. Write gnuplot in the terminal window to start the program prompt:
gnuplot
Within the gnuplot prompt, i.e. right after "gnuplot>", that you will see by default, write the following commands:
plot "trapzderror.txt" using (log($1)/log(10): (log($3)/log(10) with line
Close the plot that should pop up in a window. Then to save the plot in a "postscript" type of file using the following commands:
set terminal postscript enhanced color
set output "yourname_trapzd_err.ps"
plot "trapzderror.txt" u (log($1)/log(10): (log($3)/log(10) w l
set terminal X11
quit
You should see that the error becomes minimum for certain small values of Δx but increases if Δx is further decreased.

1. (a) **Simpson's Rule**

An alternative approach to decreasing the step size Δx for the integration, is to increase the accuracy of the functions (i.e. increase the number of terms of the Taylor expansion) used to approximate the integrand.
Consider integrating the Taylor series expansion of $f(x)$ around $x = x_0$, over an interval of length $2\Delta x$:

$$I = \int_{x_0}^{x_0+2\Delta x} f(x)\,dx = \int_{x_0}^{x_0+2\Delta x} \left[f(x_0) + f'(x_0)(x-x_0) + \frac{f''(x_0)}{2}(x-x_0)^2 + \cdots \right] dx =$$

$$f(x_0)(2\Delta x) + f'(x_0)\frac{1}{2}(2\Delta x)^2 + \frac{1}{2.3}f''(x_0)(2\Delta x)^3 + \frac{1}{3.2.4}f'''(x_0)(2\Delta x)^4 + \frac{1}{4.3.2.5}f^{iv}(x_0)(2\Delta x)^5 + \cdots$$

$$= 2f(x_0)\Delta x + 2f'(x_0)(\Delta x)^2 + \frac{4}{3}f''(x_0)(\Delta x)^3 + \frac{2}{3}f'''(x_0)(\Delta x)^4 + \frac{4}{15}f^{iv}(x_0)(\Delta x)^5 + \cdots =$$

$$\frac{\Delta x}{3}\left[f(x_0) + 4\left(f(x_0) + f'(x_0)\Delta x + \frac{1}{2}f''(x_0)(\Delta x)^2 + \frac{1}{6}f''(x_0)(\Delta x)^3 + \frac{1}{24}f^{iv}(x_0)(\Delta x)^4 + \cdots \right) + \right.$$

$$\left. \left(f(x_0) + f'(x_0)(2\Delta x) + \frac{1}{2}f''(x_0)(2\Delta x)^2 + \frac{1}{3.2}f'''(x_0)(2\Delta x)^3 + \frac{1}{4.3.2}f^{iv}(x_0)(\Delta x)^4 + \cdots \right) \right]$$

$$\left[-\frac{1}{30}f^{iv}(x_0)(\Delta x)^4 \right] = \frac{\Delta x}{3}\left(f(x_0) + 4f(x_0+\Delta x) + f(x_0+2\Delta x) + O(\Delta x)^5 \right) \tag{1}$$

Whereas the error in the Trapezium rule was $O\left((\Delta x)^3\right)$,Simpson's rule is two orders more accurate at $O\left((\Delta x)^5\right)$, giving exact integration of cubics.
Compound Simpson's Rule
To improve the accuracy when integrating over larger intervals, say the interval $x_0 = a$ to $x_N = b$, we may again be subdivide into N steps. The three-point evaluation for subinterval requires that there are an even number of subintervals. Hence we must be able to express the

number of intervals as $N = 2m$. The compound Simpsons rule is then (using $a = x_0$, $b = x_N$, $N = 2m = \frac{b-a}{\Delta x}$):

$$I = \int_a^b f(x)\,dx \approx \frac{\Delta x}{3}\sum_{i=0}^{m-1}\left[f\left(a+2i\Delta x\right)+4f\left(a+(2i+1)\,\Delta x\right)+f\left(a+(2i+2)\,\Delta x\right)\right] =$$

$$\frac{\Delta x}{3}\left[f(a)+4f\left(a+\Delta x\right)+2f\left(a+2\Delta x\right)+4f\left(a+3\Delta x\right)+\cdots \qquad +4f\left(a+(N-1)\,\Delta x\right)+f(b)\right]$$

The corresponding error is $N \times O\left((\Delta x)^5\right) \sim O\left((\Delta x)^4\right) \sim O\left(N^{-4}\right)$

```
    #include<iostream>
#include<cmath>
using namespace std;
float f(float(x))
{
return (pow(x,3)+pow(x,2)-(4*x)-5);
}
double simpson(double a, double b, int n){
int i;
long double d, I=0,J=0,A,K=0,E=0;
d=(b-a)/n;
for(i=1;i<n;i++)
{
if((i%2)!=0)
{ I=I+f(a+(i*d));
}
}
for(i=2;i<n-1;i++)
{
if((i%2)==0)
{
J=J+f(a+(i*d));
}
}
A=(d/3)*(f(a)+(4*I)+(2*J)+f(b));
return A;
}
int main()
{
int i;
long double a,b,d,n,I=0,J=0,A,K=0,E=0;
cout<< "Given f(x) = x^3 + 2x^2 - 4x - 5 "<<endl;
cout<<"Enter lower limit "<<endl;
cin>>a;
cout<<"Enter Upper Limit "<<endl;
cin>>b;
cout<<"Enter the number of intervals : "<<endl;
cin>>n;
cout<<"The Value of integral under the enterd limits is : "<<endl;
cout<<simpson(a,b,n)<<endl;
```

```
return 0;
}
```

Simpson's $\frac{3}{8}$'th Rule and Boole's Rule

We can improve the accuracy of these procedures by using higher-order polynomials. Using cubic and quartic polynomials yield, respectively, Simpson's $\frac{3}{8}$ and Boole's rules:

$$\int_{x_0}^{x_0+3\Delta x} f(x)\,dx = \frac{3\Delta x}{8}\left[f_0 + 3f_1 + 3f_2 + f_3\right] + O\left((\Delta x)^5\right)$$

$$\int_{x_0}^{x_0+4\Delta x} f(x)\,dx = \frac{2\Delta x}{45}\left[7f_0 + 32f_1 + 12f_2 + 32f_3 + 7f_4\right] + O\left((\Delta x)^7\right)$$

We can similarly devise compounded form of these rules.

```
    #include<iostream>
#include<cmath>
using namespace std;
float f(float(x))
{
return (pow(x,3)+pow(x,2)-(4*x)-5);
}
double trap(double a, double b, int n){
double x, sum, del;
del=(b-a)/n;
sum=f(a);
sum+=f(b);
x=a+del;
for(int j=1;j¡n;j++){
x+=del;
sum+=2.0*f(x); }
return sum*0.5*del;
}
double simpson(double a, double b, int n){
int i;
long double d, I=0,J=0,A,K=0,E=0;
d=(b-a)/n;
for(i=1;i<n;i++)
{
if((i%2)!=0)
{ I=I+f(a+(i*d));
}
}
for(i=2;i<n-1;i++)
{
if((i%2)==0)
{
J=J+f(a+(i*d));
}
}
A=(d/3)*(f(a)+(4*I)+(2*J)+f(b));
```

```
return A;
}

    double simpson3(double a, double b, int n)
{int i;
long double d,A3, I=0, J=0, K=0;
d=(b-a)/n;
for(i=1;i¡n;i++){
if(((i%2)!=0)&&((i%3)!=0))
{
I=I+f(a+i*d);}
}
for(i=2;i<n;i++){
if(((i%2)==0)&&((i%3)!=0))
{J=J+f(a+i*d);
}
}
for(i=3;i<n;i++){
if((i%3)==0)
{K=K+f(a+i*d);
}
}
A3=(3*d/8)*(f(a)+3*I+3*J+2*K+f(b));
return A3;
}

    int main()
{
int i;
long double a,b,d,n,I=0,J=0,A,K=0,E=0;
cout<< "Given f(x) = x^3 + 2x^2 - 4x - 5 "<<endl;
cout<<"Enter lower limit "<<endl;
cin>>a;
cout<<"Enter Upper Limit "<<endl;
cin>>b;
cout<<"Enter the number of intervals : "<<endl;
cin>>n;
cout<<"The Value of integral under the enterd limits is : "<<endl;
cout<<trap(a,b,n)<<endl;
cout<<simpson(a,b,n)<<endl;
cout<<simpson3(a,b,n)<<endl;
return 0;
}
```

Advantages of simpson 3/8 rule: There are two advantages of the 3/8ths rule: First, the error term is smaller than Simpson's rule.

The second more important use of the 3/8ths rule is for uniformly sampled function integration. Suppose you have a function known at equally spaced points. If the number of points is odd, then the composite Simpson's rule works just fine. If the number of points is even, then

you have a problem at the end. One solution is to use the 3/8ths rule. For example, if the user passed 6 samples, then you use Simpson's for the first three points, and 3/8ths for the last 4 (the middle point is common to both). This preserves the order of accuracy without putting an arbitrary constraint on the number of samples.

Method	Equation	Error
Trapezoid	$(b-a)\frac{f(x_1)+f(x_2)}{2}$	$E_t = -\frac{1}{12} f''(\xi)(b-a)^3$
	$(b-a)\frac{f(x_1)+2\sum_{i=2}^{n} f(x_i)+f(x_{n+1})}{2n}$	$E_a = -\frac{(b-a)^3}{12n^3}\sum_{i=1}^{n} \bar{f}''$
1/3 Simpson's Rule	$(b-a)\frac{f(x_1)+4f(x_2)+f(x_3)}{6}$	$E_t = -\frac{(b-a)^5}{2880} f^{(4)}(\xi)$
	$(b-a)\frac{f(x_1)+4\sum_{i=2,4,6}^{n} f(x_i)+2\sum_{j=3,5,7}^{n-1} f(x_j)+f(x_{n+1})}{3n}$	$E_a = -\frac{(b-a)^5}{180n^4}\sum_{i=1}^{n} \bar{f}^{(4)}$
3/8 Simpson's Rule	$(b-a)\frac{f(x_1)+3f(x_2)+3f(x_3)+f(x_4)}{8}$	$E_t = -\frac{(b-a)^5}{6480} f^{(4)}(\xi)$

Figure 2: Comparison of all three methods.

Example-5- A simple integration scheme

```
/* integration1.cpp*/
#include <iostream>
#include <cmath>
using namespace std;
typedef double real; // convenient
const real x0 = 1;
const real v0 = 4;
const real acc = -0.5;
real xanalytic (real t) {
    return x0 + v0*t + 0.5*acc*t*t; }
real acc1 (real x, real v, real t) { return acc; }
void take_a_step(real &x, real &v, real &t, real dt) {
// set the acceleration.
    Real a = acc1(x, v, t);
// take a time step.
    x += v*dt + 0.5*a*dt*dt;
    v += a*dt;
    t += dt;
}
int main () {
```

```
    real t = 0, x = x0, v = v0;
    real dt = 0.01;
    real tp, xp, vp;
    cout<< t << " " << x << " " << xanalytic(t) << endl;
    while (x >= 0) {
    tp = t;
    xp = x;
    vp = v;
take_a_step ( x, v, t, dt);
//print the numerical and analytic results.
cout << t << " " << x << " " << xanalytic (t) <<endl;
}
    cerr << "Final t = " << t << " " . Analytic solution = " << (-v0 – sqrt (v0*v0-
2*acc*x0))/acc <<endl;
return 0;
}
```

1. (a) **Romberg's Method**

Romberg devised an ingenious method to reduce the error in the numerical estimate of an integral. In general, we can express an integral as a sum of its numerical estimate and error terms, having dependence on the length of the sub-interval h at some power of it:

$$I = A(h) + Ch^k + C'h^{k+1} + C''h^{k+2} + \cdots$$

For example, in the Compound Trapezoidal Rule, with $\Delta x = h$:

$$A(h) = T(h) = \frac{h}{2}\left[f(x_0) + 2f(x_0 + h) + 2f(x_0 + 2h) + \cdots + f(x_N)\right]$$

$$Ch^k + \cdots = O\left(h^2\right) \Rightarrow k = 2$$

Similarly, for Compound Simpson's Rule, $k = 4$, *etc.*
We can express the same integral by an estimate an error terms when we divide the step-size by two:

$$I = A\left(\frac{h}{2}\right) + C\left(\frac{h}{2}\right)^k + C'\left(\frac{h}{2}\right)^{k+1} + C''\left(\frac{h}{2}\right)^{k+2} + \cdots$$

Combining the above two expressions for I, using suitable weighting factors, we get,

$$\left(2^k - 1\right) I = 2^k A\left(\frac{h}{2}\right) - A(h) + 2^k C\frac{h^k}{2^k} - Ch^k + O\left(h^{k+1}\right) = \left[2^k A\left(\frac{h}{2}\right) - A(h)\right] + O\left(h^{k+1}\right)$$

$$\Rightarrow I = \frac{\left[2^k A\left(\frac{h}{2}\right) - A(h)\right]}{2^k - 1} + O\left(h^{k+1}\right) = B^{(1)}(h) + Dh^{k+1} + D'h^{k+2} + D''h^{k+3} + \cdots = B^{(1)}(h) + O\left(h^{k+1}\right)$$

Here, $B^{(1)}(h)$ is a better estimate of the integral (with error $\left(h^{k+1}\right)$). We can use the same trick to get even better estimates of the integral as shown below:

$$\left(2^k - 1\right) I = 2^{k+1} B^{(1)}\left(\frac{h}{2}\right) - B^{(1)}(h) + 2^{k+1} D\frac{h^{k+1}}{2^{k+1}} - Dh^{k+1} + O\left(h^{k+2}\right)$$

$$\Rightarrow I = \frac{\left[2^{k+1} B^{(1)}\left(\frac{h}{2}\right) - B^{(1)}(h)\right]}{2^k - 1} + O\left(h^{k+2}\right) = B^{(2)}(h) + O\left(h^{k+2}\right)$$

And so on.
As an example, we can form successive better approximations of the integral I, using Compound Trapezoidal Rule as follows:

$$I = T(h) + Ch^k + C'h^{k+2} + C''h^{k+4} + \cdots, \ k = 2$$

$T^{(1)}(h) = \frac{1}{3}\left[2^k T^{(0)}\left(\frac{h}{2}\right) - T^{(0)}(h)\right]$ error: $O\left(h^{k+2}\right) = O\left(h^4\right)$

$$T^{(2)}(h) = \frac{1}{15}\left[2^{k+2}T^{(1)}\left(\frac{h}{2}\right) - T^{(1)}(h)\right] \qquad error: \ O\left(h^6\right)$$

$$T^{(3)}(h) = \frac{1}{63}\left[2^{k+4}T^{(2)}\left(\frac{h}{2}\right) - T^{(2)}(h)\right] \qquad error: \ O\left(h^8\right)$$

Example-6 Romberg's Trick applied to Trapezoidal Rule

```
#include <iostream>
#include <cmath>
using namespace std;
double power(double x, int p) {
    double val=1.0;
    int j=0;
    for (j=1; j<=abs(p); j++) val *= x;
    //abs(p) has to be used to take into account of negative value of p
    if (p>0) { return val; }
    else if (p<0) { return (1.0/val); }
    else { return 1.0; }
}
double rombergtrap(double (*func)(double), double a, double b, int n, int m) {
    double TRP[10][10];
    double x=0.0, sum=0.0, res=0.0, del=0.0, h=0.0, delb2=0.0, powerfac=0.0, factor=0.0;
    int i=0, j=0, k=0, numi=0;
    h = del = (b-a)/n; numi = n;
for(k=0; k<=m; k++) {
    sum = (*func)(a);
    x = a+h;
    for(j=1; j<n; j++, x += h) { sum += 2.0 * (*func)(x); }
    sum += (*func)(b);
    TRP[0][k] = sum * 0.5 * h;
    h /= 2.0;
    n *= 2;
}
for(i=1; i<=m; i++) {
    for(j=m-i; j>=0; j–) {
    factor = power(2.0, (2*i) );
    TRP[i][j] = factor * TRP[i-1][j+1] – TRP[i-1][j];
    TRP[i][j] /= (factor – 1.0);
}
    }
    if( m!=0) {
    TRP[m][0] = power(2.0,2*m) * TRP[m-1][1] – TRP[m-1][0];
```

```
    TRP[m][0] /= (power(2.0,2*m) – 1.0) ;
    }
return TRP[m][0];
}
double myfunc1(double x) { return cos(x); }
int main() {
    double resromb=0.0, low=0.0, up=1.0;
    int m=0, numint=10;
    low =0.0; up=M_PI/2.0; //limits of the integration
for(m=0; m<=4; m++) {
    resromb = rombergtrap(myfunc1, low, up, numint, m );
    printf("Order of the trick m=%d, resromb=%16.14lf \n \n", m, resromb);
    }
return 0;
}
```

Example-7 The period of a simple pendulum for large angle amplitude (θ_M) is given in terms of the complete elliptic integral of the first kind $K(m)$ as

$$T = 4\Big(\frac{L}{g}\Big)^{1/2} K\Big(sin^2\big(\frac{\theta_M}{2}\big)\Big) = 4\Big(\frac{L}{g}\Big)^{1/2} \int_0^{\pi/2} \Big(1 - sin^2\big(\frac{\theta_M}{2}\big) sin^2\phi\Big)^{-1/2} d\phi$$

where $0 \leq \theta_M < \pi$. (a) Using Simpson's 1/3 rule, write a code in C++ function double pendulumT(double thetaM), that evaluates the period T for a given θ_M as the arument using the above definition. Choose the value of L/g such that, as $\theta_M \to 0$, T=1s. Call the function pendulumT() from the main() function with different values of θ_M as input. (b) Using the above function, plot T *vs* θ_M for $\theta_M \in [0, \pi/2]$

```
    //Pendulum time period by using trapizeum rule
    #include<iostream>
    #include<cmath>
    #include<fstream>
    using namespace std;
    double f(double x,double theta-M){
    double s=pow((sin(theta-M/2.0)),2.0);
    double t=s*pow((sin(x)),2.0);
    double u=1-t;
    return 4.0*sqrt(0.0253)*(1.0/sqrt(u)); //(L/g)=0.0253 }
    double pendulum-T(double theta-M) { double x=0.0,sum=0.0,h=0.0; double a=0,b=M_PI/2.0;
    int i,n=100;
    if(n==1){ return 0.5*(b-a)*(f(a,theta-M)+f(b,theta-M)); }
    else
    { h=(b-a)/n; sum+=f(a,theta-M);
    x=a+h;
    for(i=1;i¡n;i++)
    {
    sum+=2.0*f(x,theta-M);
    x+=h;
    }
    sum+=f(b,theta-M);
    double T=(0.5*h*sum);
```

```
return T;
}
}
int main()
{
ofstream fout("pendulum.dat");
for(double theta-M=0.0;theta-M<=(M_PI/2.0);theta-M+=M_PI/10000.0)
{
cout<<theta-M<<" "<<pendulum-T(theta-M)<<endl;
fout<<theta-M<<" "<<pendulum-T(theta-M)<<endl;
}
return 0;
}
//Pendulum time period by using simpson 1/3 rule
/*double pendulum-Tsim(double theta-M)
{
double a=0,b=M_PI/2.0;
int i,n=100;
double I=0.0,J=0.0;
double h=(b-a)/n;
for(i=1;i<n;i+=2){ //representing the odd numbers of the intervals i.e. 1,3,5.....,n-1; since the number of interval is always even for simpson 1/3 rule
I+=f(a+i*h,theta-M);
}
for(i=2;i<n;i+=2){ //representing the even numbers of the intervals i.e. 2,4,6.....,n-2; since the number of interval is always even for simpson 1/3 rule
J+=f(a+i*h,theta-M);
}
double T=(h/3)*(f(a,theta-M)+(4*I)+(2*J)+f(b,theta-M));
return T;
}
int main(){
ofstream fout("pendulum.dat");
for(double theta-M=0.0;theta-M<=(M_PI/2.0);theta-M+=M_PI/10000.0)
{
cout<<theta-M<<" "<<pendulum-Tsim(theta-M)<<endl;
fout<<theta-M<<" "<<pendulumTsim(theta-M)<<endl;
}
return 0;
}*/
```

3 Exercises

1. Write a code that evaluates the second derivative of $sin^2(x)$ and plot the errors in your result when you use forward, backward and centered difference approximations. Use both first order and second order approximation schemes. Plot the errors in gnuplot and save the error plots as postscript files.

2. Change the code in example 3 to implement the Simpson's rule.

3. Integrate the function $\sin x$ between the limits 0 to π using, (a) Trapezoidal, Simpson's rule, Simpson's 3/8 rule and Boole's rule. Compare the errors in calculation with the actual result in all these different rules.

4. Compute the following integrals correct to 7 decimal places

$$I_1 = \int_{\pi/4}^{\pi/2} \frac{\cos x \; ln \sin x}{1 + \sin x} dx \qquad (Ans: \; -0.02657996319303578798 \ldots..)$$

$$I_2 = \int_0^{3\pi/2} \tan x \; dx \qquad (Ans: 0.960547178929730494 68 \ldots\ldots..)$$

5. Using Romberg's trick applied to the Simpson's 1/3 rd rule.

6. Using Romberg's trick applied to Simpson's 3/8 th rule.

7. Write a code to use Romberg's trick using recursive calling of function. Your definition of the function that implements the trick may be like

```
double rombergtrapzd(double(*func)(double), double a, double b, int n, int mrom)
{
.......
res = ........ rombergtrapzd (.......); //Recursive calling
........
    return res;
}
```

1. Use a suitable numerical integration scheme to evaluate the following integral:

$$\int_0^{\pi/2} \frac{1}{1+sinx} dx$$

```
#include<iostream>
#include<cmath>
using namespace std;
double f(double x){
return 1/(1+sin(x));
}
//trapezium rule
double sine_trap(double a,double b,double n){
double x=0.0,sum=0.0,h=0.0;
int i;
if(n==1){
return 0.5*(b-a)*(f(a)+f(b));
}
else {
h=(b-a)/n;
sum+=f(a);
x=a+h;
for(i=1;i<n;i++){
sum+=2.0*f(x);
x+=h;
}
sum+=f(b);
return (0.5*h*sum);
}
}
// simpson's 1/3 rule
double sine_sim13(double a,double b,int n){
int i;
double I=0.0,J=0.0;
double h=(b-a)/n;
for(i=1;i<n;i+=2){     //representing the odd numbers of the intervals i.e. 1,3,5.....,n-1; since the number of interval is always even for Simpson 1/3 rule
I+=f(a+i*h);
}
for(i=2;i<n;i+=2){     //representing the even numbers of the intervals i.e. 2,4,6.....,n-2; since the number of interval is always even for Simpson 1/3 rule
J+=f(a+i*h);
}
double A=(h/3)*(f(a)+(4*I)+(2*J)+f(b));
return A;
}
double sine_sim38(double a,double b,int n){
double h,I=0.0,J=0.0,K=0.0,L;
h=(b-a)/n;
int i;
for(i=1;i<n;i+=3){     //representing k=1,4,7,.....,N-2; since the number of interval is always an integer multiple of 3 for Simpson 3/8 rule
I+=f(a+i*h);
```

```
}
for(i=2;i<n;i+=3){      //representing k=2,5,8,.....,N-1; since the number of interval is always an integer multiple of 3 for Simpson 3/8 rule
J+=f(a+i*h);
}
for(i=3;i<n;i+=3){      //representing k=3,6,9,.....,N-3; since the number of interval is always an integer multiple of 3 for Simpson 3/8 rule
K+=f(a+i*h);
}
L=f(a)+(3*I)+(3*J)+(2*K)+f(b);
return (3*h*L)/8;
}
int main(){
double a=0.0,b=M_PI/2;
int n=120;   //number of interval is an integer multiple of 2 and 3 for simpson 1/3 rule and simpson 3/8 rule respectively
cout<<"The result by trapezium rule is: "<<sine_trap(a,b,n)<<endl;
cout<<"The result by simpson's 1/3 rule is:
"<<sine_sim13(a,b,n)<<endl;
cout<<"The result by simpson's 3/8 rule is:
"<<sine_sim38(a,b,n)<<endl;
return 0;
}
```

2. Use a suitable numerical integration scheme to evaluate the following integral:

$$\int_{-1}^{1} e^{\frac{-1}{1+x^2}}\,dx$$

```
#include<iostream>
#include<cmath>
using namespace std;
double f(double x){
return exp(-1/(1+(x*x)));
}
//trapezium rule
double sine_trap(double a,double b,double n){
double x=0.0,sum=0.0,h=0.0;
int i;
if(n==1){
return 0.5*(b-a)*(f(a)+f(b));
}
else {
h=(b-a)/n;
sum+=f(a);
x=a+h;
for(i=1;i<n;i++){
sum+=2.0*f(x);
x+=h;
}
sum+=f(b);
return (0.5*h*sum);
```

```
}
}
// simpson's 1/3 rule
double sine_sim1_3(double a,double b,int n){
int i;
double I=0.0,J=0.0;
double h=(b-a)/n;
for(i=1;i<n;i+=2){      //representing the odd numbers of the intervals i.e. 1,3,5.....,n-1; since the number of interval is always even for Simpson 1/3 rule
I+=f(a+i*h);
}
for(i=2;i<n;i+=2){      //representing the even numbers of the intervals i.e. 2,4,6.....,n-2; since the number of interval is always even for Simpson 1/3 rule
J+=f(a+i*h);
}
double A=(h/3)*(f(a)+(4*I)+(2*J)+f(b));
return A;
}
double sine_sim3_8(double a,double b,int n){
double h,I=0.0,J=0.0,K=0.0,L;
h=(b-a)/n;
int i;
for(i=1;i<n;i+=3){      //representing k=1,4,7,.....,N-2; since the number of interval is always an integer multiple of 3 for Simpson 3/8 rule
I+=f(a+i*h);
}
for(i=2;i<n;i+=3){      //representing k=2,5,8,.....,N-1; since the number of interval is always an integer multiple of 3 for Simpson 3/8 rule
J+=f(a+i*h);
}
for(i=3;i<n;i+=3){      //representing k=3,6,9,.....,N-3; since the number of interval is always an integer multiple of 3 for Simpson 3/8 rule
K+=f(a+i*h);
}
L=f(a)+(3*I)+(3*J)+(2*K)+f(b);
return (3*h*L)/8;
}
int main(){
double a=-1,b=1;
int n=120;// multiple of both 2 and 3
cout<<"The result by trapezium rule is: "<<sine_trap(a,b,n)<<endl;
cout<<"The result by simpson's 1/3 rule is:
"<<sine_sim1_3(a,b,n)<<endl;
cout<<"The result by simpson's 3/8 rule is:
"<<sine_sim3_8(a,b,n)<<endl;
return 0;
}
```

3. The time dependent temperature at a point of a long bar is given by

$$T(t) = 100\left(1 - \frac{2}{\sqrt{\pi}}\int_0^{8/\sqrt{t}} e^{-\tau^2}\, d\tau\right)$$

(a) Write a C++ function double temp(double t) that evaluates the temperature at a time t using any suitable numerical integration technique.

(b) Write a complete C++ program to save data for $T(t)$ in the range $t\epsilon[10,200]$ with an increment of 0.01. Plot the data Tvs. t using gnuplot and save the figure as temp.png.

```
#include<iostream>
#include<cmath>
#include<fstream>
using namespace std;
double f(double t,double tau){
double a=2/sqrt(M_PI);
double b=exp(-(tau*tau));
double c=1-(a*b);
return 100*c;
}
// integration using simpson 1/3 rule
double temp(double t){
int i, n=100;
double a=0, b=8/sqrt(t);
double I=0.0,J=0.0;
double h=(b-a)/n;
for(i=1;i<n;i+=2){
I+=f(t,a+i*h);
}
for(i=2;i<n;i+=2){
J+=f(t,a+i*h);
}
double A=(h/3)*(f(t,a)+(4*I)+(2*J)+f(t,b));
return A;
}
int main(){
ofstream fout("temp.dat");
for(double t=10;t<=200;t+=0.01)
fout<<t<<"   "<<temp(t)<<endl;
return 0;}
```

4. The spherical Bessel's functions have the integral representation

$$j_n(z) = \frac{z^n}{2^{n+1}n!}\int_0^{\pi} cos(zcos\theta)(sin\theta)^{2n+1}d\theta$$

(a) (i) Write a C++ function double bessel (int n, double z), that evaluates the spherical Bessel's functions$j_n(z)$, using the above definition, at some point z (Use any suitable integration technique).

(ii) Plot the first four Bessel's function using gnuplot. Save the plot as sp_bessel.png.

(b) The Fresnel integral of the second kind may be defined in terms of the spherical Bessel's function by

$$S(x) = \frac{1}{2}\int_0^x j_0(z)\sqrt{(z)}dz$$

(i) Define a C++ function double fresnel (double x) using the above definition and the function bessel (n,z) in part (a) that will calculate $S(x)$.

(ii) Plot $S(x)$ for$x \in$ [0, 20] with an increment of 0.1 and by using gnuplot. Save your plot as fresnel.png.

```
#include<iostream>
#include<cmath>
#include<fstream>
using namespace std;
double k=2*M_PI;
double factorial (int n){
double fact=1;
for(int i=1;i<=n;i++)
fact*=i;
return fact;
}
double f(int n,double z,double theta){
double a=pow(z,n)/(pow(2,(n+1))*factorial(n));
double b=cos(z*cos(theta))*pow(sin(theta),(2*n)+1);
return a*b;
}
double bessel(int n, double z){
int i;
double a=0.0,b=M_PI;
int N=100;
double I=0.0,J=0.0;
double h=(b-a)/N;
for(i=1;i<N;i+=2){
I+=f(n,z,a+i*h);
}
for(i=2;i<N;i+=2){
J+=f(n,z,a+i*h);
}
double A=(h/3)*(f(n,z,a)+(4*I)+(2*J)+f(n,z,b));
return A;
}
double ff(double z)
```

```
{
return 0.5*bessel(0,z)*sqrt(z);
}
double Fresnel(double x){
int i;
double a=0.0,b=x;
int N=1000;
double I=0.0,J=0.0;
double h=(b-a)/N;
for(i=1;i<N;i+=2){
I+=ff(a+i*h);
}
for(i=2;i<N;i+=2){
J+=ff(a+i*h);
}
double A=(h/3)*(ff(a)+(4*I)+(2*J)+ff(b));
return A;
}

int main(){
ofstream fout("bessel.dat");
ofstream file("fresnel.dat");
double z;
for(z=0.0;z<=25.0;z+=0.01){
fout<<z<<"  "<<bessel(0,z)<<"  "<<bessel(1,z)<<"  "<<bessel(2,z)<<"
"<<bessel(3,z)<<endl;
}
for(double x=0.0;x<=20;x+=0.1){
file<<x<<"  "<<Fresnel(x)<<endl;
}
return 0;}
```

5. The Fresnel sine and cosine integrals are defined respectively as

$S(x) = \int_0^x sin\left(\frac{\pi u^2}{2}\right) du$ and $C(x) = \int_0^x cos\left(\frac{\pi u^2}{2}\right) du$

(a) (i) Define two C++ functions double Fr_sin(double x) and Fr_cos(double x) that will calculate the above given integrals by using any suitable numerical technique.

(ii) Write the values of the integrals in different columns by using the defined functions in (a) in a file for $x\epsilon[0,5]$. Plot the Fresnel integrals and save the plot as fresnel.png.

(b) Consider the following diffraction pattern

$$I = 0.5I_0\{[C(u_0) + 0.5]^2 + [S(u_0) + 0.5]^2\}$$

Here $I_0$is the incident intensity, I is the diffracted intensity and u_0 is proportional to the distance away from the knife edge.

(i) Write a C++ function double diffraction (double u0, double I0) that calls the functions Fr_sin() and Fr_cos and returns the diffracted intensity

(ii) Calculate I/I_0 for u_0 varying from −1.0 to +4.0 in steps of 0.1 and write them in a file. Plot your results of I/I_0 vs u_0 from the datafile using gnuplot and save the file. [Check your answer: at u0=1,2,3 and 4 the values of I/I0 are respectively 1.25923,0.843997,1.10763 and 0.922073]

```
#include<iostream>
#include<cmath>
#include<fstream>
using namespace std;
double f1(double u){
return cos((M_PI*u*u)/2.0);
}
double f2(double u){
return sin((M_PI*u*u)/2.0);
}
double Fr_cos(double x){
int i, n=100;
double a=0, b=x;
double I=0.0,J=0.0;
double h=(b-a)/n;
for(i=1;i<n;i+=2){
I+=f1(a+i*h);
}
for(i=2;i<n;i+=2){
J+=f1(a+i*h);
}
double A=(h/3)*(f1(a)+(4*I)+(2*J)+f1(b));
return A;
}
double Fr_sin(double x){
int i,n=100;
double a=0,b=x;
double I=0.0,J=0.0;
double h=(b-a)/n;
for(i=1;i<n;i+=2){
I+=f2(a+i*h);
}
for(i=2;i<n;i+=2){
J+=f2(a+i*h);
}
double A=(h/3)*(f2(a)+(4*I)+(2*J)+f2(b));
return A;
}
double diffraction(double u0,double I0){
return 0.5*I0*(pow((Fr_cos(u0)+0.5),2)+pow((Fr_sin(u0)+0.5),2));
}
```

```
int main(){
ofstream fout("Fresnel.dat");
ofstream file("I.dat");
for(double x=0.0;x<=4;x+=0.001)
fout<<x<<"   "<<Fr_cos(x)<<"  "<<Fr_sin(x)<<endl;
double u0, I0=5.0;// choose any value of I0 excluding zero
for(u0=-1.0;u0<=4.0;u0+=0.01)
file<<u0<<"   "<<diffraction(u0,I0)/I0<<endl;
return 0;}
```

6. In the scattering of neutrons (A = 1) with a nucleus of mass number A > 1, the average of the cosine of the scattering angle ψ (i.e. $u = \langle cos\psi \rangle$) is given by

$$u(A) = \langle cos\psi \rangle = \frac{1}{2}\int_0^{\pi} \frac{Acos\theta + 1}{(A^2 + 2Acos\theta + 1)^2} sin\theta d\theta$$

(a) Write a C++ function double scatt_angle (double A) that evaluates the average $u(A)$ for a mass number A using any suitable numerical integration technique.
(7)

(b) Save data from your program for u (A) in the range $A\epsilon[2,20]$. Plot the data using gnuplot and save the plot as angle.png.

```
#include<iostream>
#include<cmath>
#include<fstream>
using namespace std;
double f(double A,double theta){
double a=(A*cos(theta))+1;
double b=a*sin(theta);
double c=A*A+2*A*cos(theta)+1;
return (0.5*b)/(c*c);
}
double scat_angle(double A){
int i, n=100;
double a=0, b=M_PI;
double I=0.0,J=0.0;
double h=(b-a)/n;
for(i=1;i<n;i+=2){
I+=f(A,a+i*h);
}
for(i=2;i<n;i+=2){
J+=f(A,a+i*h);
}
double B=(h/3)*(f(A,a)+(4*I)+(2*J)+f(A,b));
return B;
}
int main(){
ofstream fout("angle.dat");
for(double A=2.0;A<=20.0;A+=0.01)
fout<<A<<"   "<<scat_angle(A)<<endl;
```

```
return 0;}
```

7. An integral representation of the Gauss' error function is given by: 06_G6

$$erf(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} dt$$

(a) Define a function double errint(double x) that evaluates the error function for the argument x usning any suitable numerical scheme.

(b) Plot your function errint(x) for $x\epsilon[-2,2]$. with an increment of 0.01 and save the plot as errf.png.

```
#include<iostream>
#include<cmath>
#include<fstream>
using namespace std;
double f(double t){
return (2/sqrt(M_PI))*exp(-t*t);
}
double errint(double x){
double a=0.0,b=x;
double y=0.0,sum=0.0,h=0.0;
int i;
int n=100;
if(n==1){
return 0.5*(b-a)*(f(a)+f(b));
}
else {
h=(b-a)/n;
sum+=f(a);
y=a+h;
for(i=1;i<n;i++){
sum+=2.0*f(y);
y+=h;
}
sum+=f(b);
return (0.5*h*sum);
}
}
int main(){
ofstream fout("error.dat");
for(double x=-2.0;x<=2.0;x+=0.1)
{cout<<x<<"      "<<errint(x)<<endl;
fout<<x<<"      "<<errint(x)<<endl;
}
return 0;}
```

8. The Gaussian curve is defined by

$$f(x) = \frac{1}{\sqrt{2\pi}} exp\left(\frac{-x^2}{2}\right)$$

(a) Write a C++ function double mygaussian(double a), that evaluates, using any suitable numerical integration scheme, the integral

$$I(a) = \frac{1}{\sqrt{2\pi}} \int_{-a}^{a} exp\left(\frac{-x^2}{2}\right) dx$$

for a given value of the parameter a as the argument of the function.

(b) Write a C++ program that writes the values of a and I(a) in a file for $a\epsilon[-2,2]$ using the above function mygaussian(). Plot I(a) vs. a for $a\epsilon[-2,2]$ and save the plot as gauss.png. From the plot, find the value of a at which $I(a) = \frac{1}{2}$.

```
#include<iostream>
#include<cmath>
#include<fstream>
using namespace std;
double f(double x){
return sqrt(1.0/(2*M_PI))*exp(-(x*x)/2.0);}
// By trapizeum rule
double mygaussian_trap(double a){
double x=0.0,sum=0.0,h=0.0;
int i;
double n=100;
if(n==1){
return 0.5*(a-(-a))*(f(-a)+f(a));
}
else {
h=(a-(-a))/n;
sum+=f(-a);
x=-a+h;
for(i=1;i<n;i++){
sum+=2.0*f(x);
x+=h;
}
sum+=f(a);
return (0.5*h*sum);
}
}
//By Simpson's 1/3 rule
double mygaussian_simp(double a){
int i,n=100;
double I=0.0,J=0.0;
double h=(a-(-a))/n;
for(i=1;i<n;i+=2){
I+=f(-a+i*h);
}
for(i=2;i<n;i+=2){
```

```
J+=f(-a+i*h);
}
double A=(h/3)*(f(-a)+(4*I)+(2*J)+f(a));
return A;
}
int main(){
ofstream fout("gaussian.dat");
for(double a=-4.0;a<=4.0;a+=0.001){
fout<<a<<"   "<<abs(mygaussian_simp(a))<<"
"<<abs(mygaussian_trap(a))<<"    "<<f(a)<<endl;
}
return 0;
}
```

9. The period of a simple pendulum for large angle amplitude (θ_M) is given as

$$T = 4\left(\frac{L}{g}\right)^{1/2} \int_0^{\frac{\pi}{2}} \left(1 - sin^2\left(\frac{\theta_M}{2}\right) sin^2\phi\right)^{-1/2} d\phi$$

Where $0 \leq \theta_M < \pi$.

(a) Write a C or C++ function double pendulum_T(double thetaM), that evaluates the period T for a given θ_M as the argument using the above definition. Choose the value of L/g such that, as $\theta_M \to 0$, T = 1 s.

Call the function pendulum_T() from the main() function with different values of θ_M as input.

(b) Using the above function, plot T vs.$\theta_M$for $\theta_M \epsilon [0, \pi/2]$.

Hint: Check values: $\theta_M = [10^0, 50^0, 90^0] \Rightarrow T = [1.00193, 1.05033, 1.18258]$.

```
#include<iostream>
#include<cmath>
#include<fstream>
using namespace std;
double f(double x,double theta_M){
double s=pow((sin(theta_M/2.0)),2.0);
double t=s*pow((sin(x)),2.0);
double u=1-t;
return 4.0*sqrt(0.0253)*(1.0/sqrt(u));//(L/g)=0.0253
}
//Pendulum time period by using trapizeum rule
double pendulum_T(double theta_M){
double x=0.0,sum=0.0,h=0.0;
double a=0,b=M_PI/2.0;
int i,n=100;
if(n==1){
return 0.5*(b-a)*(f(a,theta_M)+f(b,theta_M));
}
else {
h=(b-a)/n;
```

```
sum+=f(a,theta_M);
x=a+h;
for(i=1;i<n;i++){
sum+=2.0*f(x,theta_M);
x+=h;
}
sum+=f(b,theta_M);
double T=(0.5*h*sum);
return T;
}
}
/*
//Pendulum time period by using Simpson 1/3 rule
double pendulum_T(double theta_M){
double a=0,b=M_PI/2.0;
int i,n=100;
double I=0.0,J=0.0;
double h=(b-a)/n;
for(i=1;i<n;i+=2){
I+=f(a+i*h,theta_M);
}
for(i=2;i<n;i+=2){
J+=f(a+i*h,theta_M);
}
double T=(h/3)*(f(a,theta_M)+(4*I)+(2*J)+f(b,theta_M));
return T;
}
*/
int main(){
ofstream fout("pendulum.dat");
for(double theta_M=0.0;theta_M<=(M_PI/2.0);theta_M+=M_PI/10000.0){
cout<<theta_M<<"   "<<pendulum_T(theta_M)<<endl;
fout<<theta_M<<"   "<<pendulum_T(theta_M)<<endl;
}
return 0;
}
```

10. The Bessel's function has the integral representation

$$J_n(x) = \frac{2}{\pi^{\frac{1}{2}}\left(n-\frac{1}{2}\right)!}\left(\frac{x}{2}\right)^n \int_0^{\pi/2} cos(xsin\theta)cos^{2n}\theta\, d\theta \quad for \quad n > \frac{-1}{2}$$

(a) (i) Write a C++ function double bessel(int n, double x), that evaluates the Bessel's functions Jn(x), using the above definition, at some point x. [3]

(ii) Write a complete C++ program to plot the first four Bessel's functions in a single plot and save the plot as bessel.png

(b) The fraction of light incident normally on a circular aperture that is transmitted is given

$$T = 1 - \frac{1}{2kq}\int_0^{2kq} J_0(x)dx$$

where a is the radius of the aperture and $k = 2\pi/\lambda$ is the wavenumber.

(i) Using the function bessel(n,x), define a C++ function double transmitted (double q) that evaluates the fraction above.

(ii)Plot transmitted(q) for ($k = 2\pi$) vs. q for $a\epsilon[0,4]$. Save the plot as transmitted.png.

```
#include<iostream>
#include<cmath>
#include<fstream>
using namespace std;
double k=2*M_PI;
double factorial (double n){
double fact=1;
for(int i=1;i<=n;i++)
fact*=i;
return fact;
}
double f(int n,double x,double theta){
double a=(2*pow((x/2.0),n))/(sqrt(M_PI)*factorial(0.5*(2*n-1)));
double b=cos(x*sin(theta))*pow(cos(theta),(2*n));
return a*b;
}
double bessel(int n, double x){
int i;
double a=0.0,b=M_PI/2.0;
int N=100;
double I=0.0,J=0.0;
double h=(b-a)/100.0;
for(i=1;i<N;i+=2){
I+=f(n,x,a+i*h);
}
for(i=2;i<N;i+=2){
J+=f(n,x,a+i*h);
}
double A=(h/3)*(f(n,x,a)+(4*I)+(2*J)+f(n,x,b));
return A;
}
double ff(double q, double x)
{
return 1-bessel(0,x)/(2*k*q);
}
double transmitted(double q){
int i;
double a=0.0,b=2*k*q;
int N=1000;
double I=0.0,J=0.0;
double h=(b-a)/N;
```

```
for(i=1;i<N;i+=2){
I+=ff(q,a+i*h);
}
for(i=2;i<N;i+=2){
J+=ff(q,a+i*h);
}
double A=(h/3)*(ff(q,a)+(4*I)+(2*J)+ff(q,b));
return A;
}
int main(){
ofstream fout("bessel.dat");
ofstream file("transmited.dat");
double x;
for(x=0.0;x<=25.0;x+=0.01){
fout<<x<<"  "<<bessel(0,x)<<"  "<<bessel(1,x)<<"  "<<bessel(2,x)<<"
"<<bessel(3,x)<<endl;
}
for(double q=0.0;q<=4;q+=0.01){
file<<q<<"  "<<transmitted(q)<<endl;
}
return 0;}
```

11. The spherical Bessel's functions have the following integral representation

$$j_n(x) = \frac{x^n}{2^{n+1}n!}\int_0^{\pi} \cos(xcos\theta)\,(sin\theta)^{2n+1}d\theta$$

(a) Write a C++ function **double Bessel (double x, int n)**, that evaluates the Bessel's functions $j_n(x)$, using the above definition, at some point x by using any suitable integration technique. Then plot the first four Bessel's function for $x\epsilon[0,25]$ using gnuplot. Save the plot as **bessel.png**.

(b) An analysis of the antenna radiation pattern for a system of circular aperture involves the function

$$g(u) = \int_0^1 f(r)j_0(ur)rdr$$

where $f(r) = 1 - r^2$
Write a C++ function **double ant_radiation (double u)** from the above definition and plot $g(u)$ Vs. u for $u\epsilon[0,25]$. Save the plot as **radiation.png**.

```
#include<iostream>
#include<cmath>
#include<fstream>
using namespace std;
double k=2*M_PI;
double factorial (int n){
double fact=1;
for(int i=1;i<=n;i++)
```

```
fact*=i;
return fact;
}
double f(int n,double z,double theta){
double a=pow(z,n)/(pow(2,(n+1))*factorial(n));
double b=cos(z*cos(theta))*pow(sin(theta),(2*n)+1);
return a*b;
}
double bessel(int n, double z){
int i;
double a=0.0,b=M_PI;
int N=100;
double I=0.0,J=0.0;
double h=(b-a)/N;
for(i=1;i<N;i+=2){
I+=f(n,z,a+i*h);
}
for(i=2;i<N;i+=2){
J+=f(n,z,a+i*h);
}
double A=(h/3)*(f(n,z,a)+(4*I)+(2*J)+f(n,z,b));
return A;}
double f1(double r){
return 1-(r*r);
}
double ff(double r,double u){
return f1(r)*bessel(0,u*r)*r;
}
double ant_radiation(double u){
int i;
double a=0.0,b=1.0;
int N=100;
double I=0.0,J=0.0;
double h=(b-a)/N;
for(i=1;i<N;i+=2){
I+=ff(a+i*h,u);
}
for(i=2;i<N;i+=2){
J+=ff(a+i*h,u);
}
double A=(h/3)*(ff(a,u)+(4*I)+(2*J)+ff(b,u));
return A;
}
int main(){
ofstream fout("bessel.dat");
ofstream file("radiation.dat");
double z;
for(z=0.0;z<=25.0;z+=0.01){
fout<<z<<"  "<<bessel(0,z)<<"  "<<bessel(1,z)<<"  "<<bessel(2,z)<<"
"<<bessel(3,z)<<endl;
}
for(double u=0.0;u<=25;u+=0.1){
```

```
file<<u<<"  "<<ant_radiation(u)<<endl;
}
return 0;}
```

12. The complete elliptic integral of the first and second kind are defined respectively as

$$K(m) = \int_0^{\frac{\pi}{2}} (1 - msin^2\theta)^{-\frac{1}{2}} d\theta \qquad \text{and} \qquad E(m) = \int_0^{\frac{\pi}{2}} (1 - msin^2\theta)^{\frac{1}{2}} d\theta$$

(a) Define two functions **double K(double m)** and **double E(double m)** to find the values of $K(m)$ and $E(m)$ by using any suitable numerical integration technique.

4

(b) Call the functions **K()** and **E()** into the **main()** in order to plot the complete elliptic integrals in a single plot for $m\epsilon[0.0,1.0]$ with an increment of 0.01. Save the plot as **elliptic.png**.

3

(c) The period of a simple pendulum for large angle amplitude (θ_M)is given in terms of the complete elliptic integral of the first kind $K(m)$ as

$$T = 4\left(\frac{L}{g}\right)^{\frac{1}{2}} K\left(\sin^2\left(\frac{\theta_M}{2}\right)\right) \quad where\ 0 \le \theta_M < \pi$$

(i) Write a C++ function **double penduT(double thetaM)** that will call the function **K()** and evaluate the time period T for a given θ_M using the above definition. Choose L/g such that, as $\theta_M \to 0, T = 1\ s$.

2

(ii) Call the function **penduT()** into the **main()** in order to plot T Vs. θ_M for $\theta_M\epsilon[0,\pi/2]$. Take the increment of θ_M as $\frac{\pi}{30}$. Save the plot as **period.png**.

2

Check values: $\theta_M = [10°, 50°, 90°] => T = [1.00193, 1.05033, 1.18258]$.

```
#include<iostream>
#include<cmath>
#include<fstream>
using namespace std;
double f1(double m, double theta){
double a=1-m*pow(sin(theta),2);
return 1/sqrt(a);
}
double f2(double m, double theta){
double a=1-m*pow(sin(theta),2);
return sqrt(a);
}
double K(double m){
int i;
double a=0.0, b=M_PI/2,n=1000;
double I=0.0,J=0.0;
double h=(b-a)/n;
```

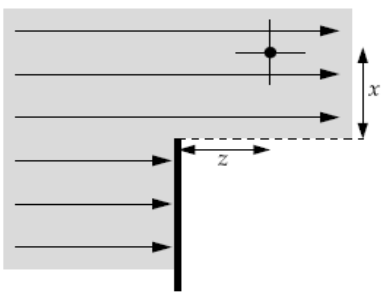
```
for(i=1;i<n;i+=2){
I+=f1(m,a+i*h);
}
for(i=2;i<n;i+=2){
J+=f1(m,a+i*h);
}
double A=(h/3)*(f1(m,a)+(4*I)+(2*J)+f1(m,b));
return A;
}
double E(double m){
int i;
double a=0.0, b=M_PI/2,n=1000;
double I=0.0,J=0.0;
double h=(b-a)/n;
for(i=1;i<n;i+=2){
I+=f2(m,a+i*h);
}
for(i=2;i<n;i+=2){
J+=f2(m,a+i*h);
}
double A=(h/3)*(f2(m,a)+(4*I)+(2*J)+f2(m,b));
return A;
}
double penduT(double thetaM){
return 4*sqrt(0.0253)*K(pow(sin(thetaM/2),2));          //(L/g)=0.0253


}
int main(){
ofstream fout("ellip.dat");
for(double m=0.0;m<=1.0;m+=0.01)
fout<<m<<"   "<<K(m)<<"   "<<E(m)<<endl;
ofstream file("time.dat");
for(double thetaM=0.0;thetaM<=M_PI/2;thetaM+=M_PI/30)
file<<thetaM<<"   "<<penduT(thetaM)<<endl;
return 0;}
```

13. Suppose a plane wave of wavelength λ such as light or a sound wave is blocked by an object with a straight edge, represented by the solid line at the bottom of this figure:
The wave will be diffracted at the edge and the resulting intensity at the position (x, z) marked by the dot is given by near-field diffraction theory to be

$$I = \frac{I_0}{8}([2C(u)+1]^2 + [2S(u)+1]^2)$$

Where I_0 is the intensity of the wave before diffraction and $u = x\sqrt{\frac{2}{\lambda z}}$ and the Fresnel integrals are defined as $C(u) = \int_0^u \cos(\frac{\pi t^2}{2})dt$ and $S(u) = \int_0^u \sin(\frac{\pi t^2}{2})dt$



(a) Define two functions **Fr_cos(double u)** and **double Fr_sin(double u)** to evaluate the above integrals by using any suitable numerical technique.

(b) Define another function **double diffraction(double x)** that will calculate I/I_0 by using the above definition and for a given value of x (as shown in the figure).

(c) Write a complete C++ program to calculate I/I_0 and make a plot of it as a function of x in the range $-5m$ to $5m$ for the case of a sound wave with wavelength $\lambda = 1m$, measured $z = 3m$ past the straight edge. Save the plot as **intensity.png**.

```
#include<iostream>
#include<cmath>
#include<fstream>
using namespace std;
double f1(double t){
return cos((M_PI*t*t)/2.0);
}
double f2(double t){
return sin((M_PI*t*t)/2.0);
}
double Fr_cos(double x){
int i, n=100;
double a=0, b=x;
double I=0.0,J=0.0;
double h=(b-a)/n;
for(i=1;i<n;i+=2){
I+=f1(a+i*h);
}
for(i=2;i<n;i+=2){
J+=f1(a+i*h);
}
double A=(h/3)*(f1(a)+(4*I)+(2*J)+f1(b));
return A;
}
double Fr_sin(double x){
int i,n=100;
double a=0,b=x;
double I=0.0,J=0.0;
double h=(b-a)/n;
for(i=1;i<n;i+=2){
I+=f2(a+i*h);
}
for(i=2;i<n;i+=2){
J+=f2(a+i*h);
}
double A=(h/3)*(f2(a)+(4*I)+(2*J)+f2(b));
return A;
}
double u(double x){
double lambda=1, z=3;
return x*sqrt(2/(lambda*z));
}
double diffraction(double x){
```

```
return (pow((2*Fr_cos(u(x)))+1,2)+pow((2*Fr_sin(u(x)))+1,2))/8;
}
int main(){
ofstream fout("Fresnel.dat");
ofstream file("si.dat");
for(double x=-6.0;x<=6;x+=0.01)
fout<<x<<"   "<<Fr_cos(x)<<"  "<<Fr_sin(x)<<endl;
double x;
for(x=-5.0;x<=5.0;x+=0.01)
file<<x<<"   "<<diffraction(x)<<endl;
return 0;}
```

14.

```
//A C++ Program To evaluate a Definite Integral by Gauss Quadrature
Formula
#include<iostream>
#include<cmath>
#include<iomanip>

using namespace std;

//The given Function of Integration
double f(double x)
{
    return x*x*x;
}

//Legendre's Polynomial Pn(x)
double Pn(double x, int n){
if(n==0) return 1;
else if (n==1) return x;
else return (((2*n-1)*x*Pn(x,n-1))-((n-1)*Pn(x,n-2)))/n;
}

//Derivative of Legendre's Polynomial Pn(x) i.e. d/dx (Pn(x))
double d_Pn(double x,int n){
return (n/((x*x)-1))*(x*Pn(x,n)-Pn(x,n-1));
}

//The Gaussian Quadrature rule
void gauss_qua(double a,double b,int n){
//First find the solution of the n-th Legendre polynomial  i.e. the n sample points xk (k = 1,2,...,n)
between the interval [-1,1]
if (n<=0)
cout<<"The number of sample points can not be negative or zero";
double x0[n],x[n],w[n],x_prime[n],w_prime[n],c,h;
int i;
for(i=0;i<n;i++){
x0[i]=cos((M_PI*((i+1)-0.25))/(n+0.5));
```

/*initial guess of the k-th root of the n-th Legendre polynomial by Tricomi's approximation $x(n,k) = cos(\pi * (k-0.25)/(n+0.5))$; where the values of k starts from 1 i.e. k=1,2,..,n. Thus to keep the correct values of k , which is represented by i in the loop , we add 1 to i, otherwise the values of k will be counted as k=0,1,....,n-1 which is not correct at all, but the array representation of the root needs to start i from 0;*/

```
//Imprived root by Newton-Raphson method
c=x0[i];
h=Pn(c,n)/d_Pn(c,n);
while (abs(h)>=0.00000000001){
c=c-h;
h=Pn(c,n)/d_Pn(c,n);
}
cout<<c<<endl;
x[i]=c;    // We store the root in an array
// Once the root is i.e. the sample point is found we calculate the weight
w[i]= (2/((1-c*c)*(d_Pn(c,n)*d_Pn(c,n))));
}
// Once the sample points (roots) and weights are calculated between [-1,1] (from the Legendre polynomial) we need to rescale the values for our given interval [a,b]
for(i=0;i<n;i++){
x_prime[i]=0.5*(((b-a)*x[i])+(b+a));
w_prime[i]=0.5*(b-a)*w[i];
}
//Now everything is ready and we have to perform the last step of the algorith i.e. use Gaussian Quadrature method defined in the equation
double result=0.0;
for(i=0;i<n;i++){
result+=(w_prime[i]*f(x_prime[i]));
}
// Now the desired result is shown below
cout<<"The value of integration is: "<<result<<endl;
/* If we use return type function then simply replace the cout line by return result
 return result;*/
}

int main(){
double a,b;
int n;
cout<<"Enter the lower limit of integration: ";
cin>>a;
cout<<"Enter the upper limit of integration: ";
cin>>b;
cout<<"Enter the order n of the legendre polynomial: ";
cin>>n;
gauss_qua(a,b,n);
/* When the gauss_qua(a.b.n) is return type write the following
cout<<"The value of integration is: "<<gauss_qua(a,b,n)<<endl;*/
return 0;}
```

15. Period of an anharmonic oscillator is given by

$$T = \sqrt{8m}\int_0^A \frac{dx}{\sqrt{V(A)-V(x)}}$$

where A is the amplitude of oscillation.

(a) Suppose the potential is $V(x) = x^4$ and the mass of the particle is $m = 1\,kg$. Write a C++ function **double tperiod(double A)** that evaluates the time period of the oscillator for the given amplitude **A** using Gaussian quadrature with 20 points.

[The required relations for Gaussian quadrature method are as follows

$$nP_n(x) = (2n-1)xP_{n-1}(x) - (n-1)P_{n-2}(x)$$

$$P_n'(x) = \frac{n}{x^2-1}\{xP_n(x) - P_{n-1}(x)\}$$

Where $P_n(x)$ the Legendre polynomial of degree n and $P_n'(x)$ is the derivative of $P_n(x)$]

(b) Use the function to make a graph of the period for the amplitudes ranging from $A = 0\ to\ A = 2$. You should find that the oscillator gets faster as the amplitude increases.

```
#include<iostream>
#include<cmath>
#include<iomanip>
#include<fstream>
using namespace std;
double V(double x){
return pow(x,4);
}
//The given Function of Integration
double f(double x, double A)
{double m=1.0;
double a=sqrt(8*m);
double b=sqrt(V(A)-V(x));
    return a/b;
}

//Legendre's Polynomial Pn(x)
double Pn(double x, int n){
if(n==0) return 1;
else if (n==1) return x;
else return (((2*n-1)*x*Pn(x,n-1))-((n-1)*Pn(x,n-2)))/n;
}

//Derivative of Legendre's Polynomial Pn(x)i.e.d/dx(Pn(x))
double d_Pn(double x,int n){
return (n/((x*x)-1))*(x*Pn(x,n)-Pn(x,n-1));
}

//The Gaussian Quadrature rule
double tperiod(double A){
```

```
//First find the solution of the n-th legendre polynomial i.e. the n sample points x_k (k=1,2,...,n) between the interval [-1,1]
double a=0,b=A;
int n=20;
double x0[n],x[n],w[n],x_prime[n],w_prime[n],c,h;
int i;
for(i=0;i<n;i++){
x0[i]=cos((M_PI*((i+1)-0.25))/(n+0.5));
/*initial guess of the k-th root of the n-th legendre polynomial by Tricomi's approximation x(n,k)=cos(PI*(k-0.25)/(n+0.5)); where the values of k starts from 1 i.e. k=1,2,..,n. Thus to keep the correct values of k , which is represented by i in the loop , we add 1 to i, otherwise the values of k will be counted as k=0,1,....,n-1 which is not correct at all, but the array representation of the root needs to start i from 0;*/
//Imprived root by Newton-Raphson method
c=x0[i];
h=Pn(c,n)/d_Pn(c,n);
while (abs(h)>=0.00000000001){
c=c-h;
h=Pn(c,n)/d_Pn(c,n);
}
x[i]=c;    // We store the root in an array
// Once the root is i.e. the sample point is found we calculate the weight
w[i]= (2/((1-c*c)*(d_Pn(c,n)*d_Pn(c,n))));
}
// Once the sample points (roots) and weights are calculated between [-1,1] (from the Legendre polynomial) we need to rescale the values for our given interval [a,b]
for(i=0;i<n;i++){
x_prime[i]=0.5*(((b-a)*x[i])+(b+a));
w_prime[i]=0.5*(b-a)*w[i];
}
//Now everything is ready and we have to perform the last step of the algorith i.e. use Gaussian Quadrature method defined in the equation
double result=0.0;
for(i=0;i<n;i++){
result+=(w_prime[i]*f(x_prime[i],A));
}
// Now the desired result is shown below
/*For void type function
cout<<"The value of integration is: "<<result<<endl;*/
//If we use return type function then simply replace the cout line by return result
 return result;
}

int main(){
ofstream fout("t.dat");
double A;
for(A=0;A<=2;A+=0.01)
fout<<A<<"     "<<tperiod(A)<<endl;
return 0;}
```

16. Debye's theory of solids gives the heat capacity of a solid at temperature T to be

$$C_v = 9V\rho K_B \left(\frac{T}{\theta_D}\right)^3 \int_0^{\theta_D/T} \frac{x^4 e^x}{(e^x-1)^2}\,dx$$

where V is the volume of the solid, ρ is the number density of atoms, k_B is Boltzmann's constant, and θ_D is the so-called Debye temperature, a property of solids that depends on their density and speed of sound.

a) Write a C++ function **double sp_heat(double T)** that calculates C_V for a given value of the temperature, for a sample consisting of 1000 cubic centimeters of solid aluminum, which has a number density of $\rho = 6.022 \times 10^{28}\ m^{-3}$ and a Debye temperature of $\theta_D = 428K$. Use Gaussian quadrature to evaluate the integral, with 50 sample points.

b) Use your function to make a graph of the heat capacity as a function of temperature from $T = 5K$ *to* $T = 500K$. Save the graph as **spheat.png**.

```
#include<iostream>
#include<cmath>
#include<fstream>
#define V pow(10,-3)
#define rho 6.022*pow(10,28)
#define KB 1.38*pow(10,-23)
#define theta_D 428
using namespace std;
double debye(double x,double T){
double a=exp(x);
double b=a*pow(x,4);
double c=pow((a-1),2);
double d=b/c;
double e=pow((T/theta_D),3.0);
double f=9.0*V*rho*KB;
return (f*e*d);
}
//Legendre's Polynomial Pn(x)
double Pn(double x, int n){
if(n==0) return 1;
else if (n==1) return x;
else return (((2*n-1)*x*Pn(x,n-1))-((n-1)*Pn(x,n-2)))/n;
}

//Derivative of Legendre's Polynomial Pn(x)i.e.d/dx (Pn(x))
double d_Pn(double x,int n){
return (n/((x*x)-1))*(x*Pn(x,n)-Pn(x,n-1));
}

//Calculation of specific heat by using the Gaussian Quadrature rule
double sp_heat(double T){
double a=0.0;          //The lower limit of integration
double b=theta_D/T;    //The upper limit of integration
```

```
int n=6;            //The order n of the Legendre polynomial
//First find the solution of the n-th legendre polynomila i.e. the n sample points x_k (k=1,2,...,n) between the interval [-1,1]
if (n<=0)
cout<<"The number of sample points can not be negative or zero";
double x0[n],x[n],w[n],x_prime[n],w_prime[n],c,h;
int i;
for(i=0;i<n;i++){
x0[i]=cos((M_PI*((i+1)-0.25))/(n+0.5));

//Imprived root by Newton-Raphson method
c=x0[i];
h=Pn(c,n)/d_Pn(c,n);
while (abs(h)>=0.00000000001){
c=c-h;
h=Pn(c,n)/d_Pn(c,n);
}
x[i]=c;

w[i]= (2/((1-c*c)*(d_Pn(c,n)*d_Pn(c,n))));
}

for(i=0;i<n;i++){
x_prime[i]=0.5*(((b-a)*x[i])+(b+a));
w_prime[i]=0.5*(b-a)*w[i];
}

double result=0.0;
for(i=0;i<n;i++){
result+=(w_prime[i]*debye(x_prime[i],T));
}
return result;
}

int main(){

ofstream fout("debye.dat");
for(double T= 5.0;T<=500;T+=0.005){
cout<<T<<"\t\t"<<sp_heat(T)<<endl;
fout<<T<<"\t\t"<<sp_heat(T)<<endl;
}
return 0;
}
```

17. Integrate $f(x) = e^{-x^2}$ over the range from zero to infinity

//let $z = x/(1+x)$ i.e. $x = z/(1-z)$; then this change of variable gives the same result but in the range 0 to 1 ;; see lecture notes

```
#include<iostream>
```

```
#include<cmath>
#include<fstream>
using namespace std;
double f(double z){
double a=(z*z)/((1-z)*(1-z));
double b=exp(-a)/((1-z)*(1-z));
return b;}
//Legendre's Polynomial P_n(x)
double Pn(double x, int n){
if(n==0) return 1;
else if (n==1) return x;
else return (((2*n-1)*x*Pn(x,n-1))-((n-1)*Pn(x,n-2)))/n;
}

//Derivative of Legendre's Polynomial P_n(x) i.e. d/dx (P_n(x))
double d_Pn(double x,int n){
return (n/((x*x)-1))*(x*Pn(x,n)-Pn(x,n-1));
}

//The Gaussian Quadrature rule
void gauss_qua(double a,double b,int n){
if (n<=0)
cout<<"The number of sample points can not be negative or zero";
double x0[n],x[n],w[n],x_prime[n],w_prime[n],c,h;
int i;
for(i=0;i<n;i++){
x0[i]=cos((M_PI*((i+1)-0.25))/(n+0.5));

//Imprived root by Newton-Raphson method
c=x0[i];
h=Pn(c,n)/d_Pn(c,n);
while (abs(h)>=0.00000000001){
c=c-h;
h=Pn(c,n)/d_Pn(c,n);
}
x[i]=c;  w[i]= (2/((1-c*c)*(d_Pn(c,n)*d_Pn(c,n))));
}
for(i=0;i<n;i++){
x_prime[i]=0.5*(((b-a)*x[i])+(b+a));
w_prime[i]=0.5*(b-a)*w[i];
}

double result=0.0;
for(i=0;i<n;i++){
result+=(w_prime[i]*f(x_prime[i]));
}
// Now the desired result is shown below
cout<<"The value of integration is: "<<result<<endl;
// If we use return type function then simply replace the cout line by return result
// return result;
}
```

```
int main(){
double a,b;
int n;
cout<<"Enter the lower limit of integration: ";
cin>>a;
cout<<"Enter the upper limit of integration: ";
cin>>b;
cout<<"Enter the order n of the legendre polynomial: ";
cin>>n;
gauss_qua(a,b,n);
// When the gauss_qua(a.b.n) is return type write the following
// cout<<"The value of integration is: "<<gauss_qua(a,b,n)<<endl;
return 0;}
```

// the answer is sqrt(M_PI)/2=0.886226925453. . .

18. The eigenfunctions of the one-dimensional simple harmonic oscillator are

$$u_n(x) = \frac{1}{\sqrt{2^n n!\sqrt{\pi}}} H_n(x)\, e^{-x^2/2}$$

where $H_n(x)$ is the Hermite polynomial of order n and we have used a system of units in which $\sqrt{\hbar/m\omega} = 1$.

(a) Define a function double hermite (double x, int N) that will evaluates the Hermite polynomial of order n at a point x by using the following recursion relation

$$H_n(x) = 2xH_{n-1}(x) - 2(n-1)H_{n-2}(x)$$

(b) define a function double f (double x, int N) that will evaluates the eigenfunctions by using the above given definition. Plot the eigenfunctionds for n= 0,1,2,3 and for $x\epsilon[-5,5]$ with an increment of 0.001.

(C) Evaluate $\int_{-\infty}^{+\infty} u_n(x)u_m(x)dx$ for $n, m\epsilon[0,5]$. Can you interpret the result?

```
#include<iostream>
#include<cmath>
#include<fstream>
using namespace std;
double hermite(double x, int N){
if(N==0) return 1;
else if (N==1) return 2*x;
else return (2*x*hermite(x,N-1)-2*(N-1)*hermite(x,N-2));
}
int fact(int N){
if (N>1)
return N*fact(N-1);
```

```
else return 1;
}
double f(double x,int N){
double a=1/(sqrt(pow(2,N)*fact(N)*sqrt(M_PI)));
return a*hermite(x,N)*exp(-(x*x)/2);
}

double eigen(double z,int N){          //The change of variable is done to change the limit of integratio from [−∞,+∞] to [-1,+1]
return (f((z/(1-z*z)),N)*f((z/(1-z*z)),N)*(1+z*z))/((1-(z*z))*(1-(z*z)));
 }
//when we use both m and n in the calculation of the integration i.e. u_n(x) * u_m(x)
double d_eigen(double z,int N,int M){
return (f((z/(1-z*z)),N)*f((z/(1-z*z)),M)*(1+z*z))/((1-(z*z))*(1-(z*z)));
}

//Start of Gaussian_Quadrature process to solve the integration
//Legendre's Polynomial P_n(x)
double Pn(double x, int n){
if(n==0) return 1;
else if (n==1) return x;
else return (((2*n-1)*x*Pn(x,n-1))-((n-1)*Pn(x,n-2)))/n;
}

//Derivative of Legendre's Polynomial P_n(x) i.e. d/dx (P_n(x))
double d_Pn(double x,int n){
return (n/((x*x)-1))*(x*Pn(x,n)-Pn(x,n-1));
}

//The Gaussian Quadrature rule
double gauss_qua(int N,int M){    //The order of Hermite polynomial and the eigenfunction is represented by N which is given by small n in question.
//The degree to calculate Legendre polynomial and Guassian quadrature fomula is represented by n
double a=-1,b=1;          //limit of integration
int n=30;                 //order of Legendre polynomial i.e. the number of sample points for Gaussian quadrature rule

if (n<=0)
cout<<"The number of sample points can not be negative or zero";
double x0[n],x[n],w[n],x_prime[n],w_prime[n],c,h;
int i;
for(i=0;i<n;i++){
x0[i]=cos((M_PI*((i+1)-0.25))/(n+0.5));
//Imprived root by Newton-Raphson method
c=x0[i];
h=Pn(c,n)/d_Pn(c,n);
while (abs(h)>=0.00000000001){
c=c-h;
```

```
h=Pn(c,n)/d_Pn(c,n);
}
x[i]=c;  w[i]= (2/((1-c*c)*(d_Pn(c,n)*d_Pn(c,n))));
}
for(i=0;i<n;i++){
x_prime[i]=0.5*(((b-a)*x[i])+(b+a));
w_prime[i]=0.5*(b-a)*w[i];
}
double result=0.0;
for(i=0;i<n;i++){
//result+=(w_prime[i]*eigen(x_prime[i],N));
result+=(w_prime[i]*d_eigen(x_prime[i],N,M));//when we use both m and n in the calculation of the integration i.e. u_n(x) * u_m(x) then add the term int M
}
// Now the desired result is shown below
//cout<<"The value of integration is: "<<result<<endl;
// If we use return type function then simply replace the cout line by return result
 return result;
}
int main(){
ofstream fout("eigen.dat");
for(double x=-5;x<=5;x+=0.001)
{
fout<<x<<"   "<<f(x,0)<<"   "<<f(x,1)<<"   "<<f(x,2)<<"
"<<f(x,3)<<endl;}

int N,M    ;
for(N=0;N<=4;N++){
for(M=0;M<=4;M++){
cout<<N<<"  "<<M<<"  "<<gauss_qua(N,M)<<endl;
}
cout<<endl;
}

return 0;}
```

Department of Physics
University of Dhaka
Computational Physics
ORDINARY DIFFERENTIAL EQUATION

An ODE is an equality involving a function and its derivatives, where the unknown function is a function of one variable. An ODE of order n is an equation of the form

$$f(x, y(x), y'(x), y''(x), ...y^n(x)) = 0$$

where you have a single independent variable x, a vector-valued function, f, and its derivatives. Problems involving ordinary differential equations can always be reduced to the study of sets of first-order differential equations. For example, the equation

$$\frac{d^2y}{dx^2} + q(x)\frac{dy}{dx} = r(x)$$

can be written as two first-order equations:

$$\frac{dy}{dx} = z(x)$$

$$\frac{dz}{dx} = r(x) - q(x)z(x)$$

where z is a new variable. The usual new choice for the new variables is to let them just be derivatives of each other (and the original variable). This exemplifies the procedure for an arbitrary ODE. Thus, the generic problem for an ordinary differential equation is reduced to a set of n coupled first-order differential equations.

The second thing to remember with ODEs is that the problem specification is incomplete without knowledge of the nature of the problems boundary conditions. Boundary conditions can be as simple as discrete numerical values or as complex as nonlinear equations, but are not maintained outside of thier specified constraints. Typically, the numerical methods used in solving ODEs are directed by the nature of the boundary conditions and you usually need to constrain the solution with one boundary condition per unknown. The two catergories of boundary conditions are illustrated below.

1. **Initial Value Problem:** If all of the boundary conditions are specified at the starting time, its an initial value problem. An example of such a problem arises when trying to describe a balls vertical movement over time as it is tossed in the air. Here, one must specify the balls starting position and velocity to have a solution.

2. **Boundary Value Problem:** A different kind of problem arises if the boundary conditions are specified at more than one x. For instance, say we want to throw a ball in a waste basketthis involves getting the ball to be at a particular point at a given time. Another way of phrasing this is that you know where you are throwing from and the final position, but dont care about the velocity. Explicitly, in this case the boundary conditions are y(0) = 0 and y(1) = k.

- **Euler's Method** This is a numerical methods that will allow us to approximate solutions to differential equations. There are many different methods that can be used to approximate solutions to a differential equation and in fact whole classes can be taught

just dealing with the various methods. We are going to look at one of the oldest and easiest to use here. This method was originally devised by Euler and is called, oddly enough, Eulers Method. Let us start with a general first order differential equation along with the initial condition,

$$\frac{dy}{dt} = f(t, y), \qquad y(t_0) = y_0$$

Where $f(y, t)$ is a known function and the values in the initial condition are also known numbers. If f and f_y are continuous functions then there is a unique solution to the ODE in some surrounding of $t = t_0$. So let us assume that everything is nice and continuous so that we know that a solution will in fact exist. And we want to approximate the solution near $t = t_0$. Well start with the two pieces of information that we do know about the solution. First, we know the value of the solution at $t = t_0$ from the initial condition. Second, we also know the value of the derivative at $t = t_0$. We can get this by plugging the initial condition into $f(t, y)$ into the differential equation itself. So, the derivative at this point is.

$$\left.\frac{dy}{dt}\right|_{t=t_0} = f(t_0, y_0)$$

Now, recall from Calculus class that these two pieces of information are enough for us to write down the equation of the tangent line to the solution at $t = t_0$. The tangent line is

$$y = y_0 + f(t_0, y_0)(t - t_0)$$

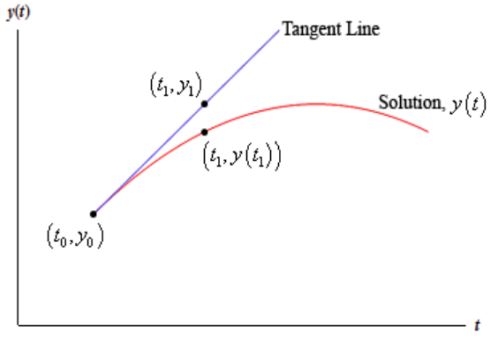


Figure 1: The tangent line

If t_1 is close enough to t_0 then the point y_1 on the tangent line should be fairly close to the actual value of the solution at t_1, or $y(t_1)$. Finding y_1 is easy enough. All we need to do is plug t_1 in the equation for the tangent line.

$$y_1 = y_0 + f(t_0, y_0)(t_1 - t_0)$$

Now, we would like to proceed in a similar manner, but we dont have the value of the solution at t_1 and so we wont know the slope of the tangent line to the solution at this point. This is a problem. We can partially solve it however, by recalling that y_1 is an approximation to the solution at t_1. If y_1 is a very good approximation to the actual value of the solution then we can use that to estimate the slope of the tangent line at t_1.

So, lets hope that y_1 is a good approximation to the solution and construct a line through the point (t_1, y_1) that has slope $f(t_1, y_1)$. This gives

$$y = y_1 + f(t_1, y_1)(t - t_1)$$

Now, to get an approximation to the solution at $t = t_2$ we will hope that this new line will be fairly close to the actual solution at t_2 and use the value of the line at t_2 as an approximation to the actual solution. This gives.

$$y_2 = y_1 + f(t_1, y_1)(t_2 - t_1)$$

We can continue in this fashion. Use the previously computed approximation to get the next approximation. So,

$$y_3 = y_2 + f(t_2, y_2)(t_3 - t_2)$$
$$y_4 = y_3 + f(t_3, y_3)(t_4 - t_3)$$

and so on In general, if we have t_n and the approximation to the solution at this point, y_n, and we want to find the approximation at t_{n+1} all we need to do is use the following.

$$y_{n+1} = y_n + f(t_n, y_n)(t_{n+1} - t_n)$$

If we define $f(t_n, y_n) = f_n$ and $(t_{n+1} - t_n) = h$, then the formula become,

$$y_{n+1} = y_n + hf_n$$

So, how do we use Eulers Method? Its fairly simple. We start with the first equation and then decide if we want to use a uniform step size or not. Then starting with (t_0, y_0) we repeatedly evaluate the last equation depending on whether we chose to use a uniform step size or not. We continue until weve gone the desired number of steps or reached the desired time. This will give us a sequence of numbers y_1 , y_2 , y_3 , y_n that will approximate the value of the actual solution at t_1 , t_2 , t_3 , t_n.

What do we do if we want a value of the solution at some other point than those used here? One possibility is to go back and redefine our set of points to a new set that will include the points we are after and redo Eulers Method using this new set of points. However this is cumbersome and could take a lot of time especially if we had to make changes to the set of points more than once.

In practice you would need to write a computer program to do these computations for you. In most cases the function f(t,y) would be too large and/or complicated to use by hand and in most serious uses of Eulers Method you would want to use hundreds of steps which would make doing this by hand prohibitive. So, here is a bit of pseudo-code that you can use to write a program for Eulers Method that uses a uniform step size, h.

- ⋆ define $f(t, y)$
- ⋆ input t_0 and y_0
- ⋆ input step size, h and the number os steps, n.
- ⋆ for i form 1 to n do
  - ◇ $m = f(t_0, y_0)$
  - ◇ $y_1 = y_0 + h * m$
  - ◇ $t_1 = t_0 + h$
  - ◇ Print t_1 and y_1
  - ◇ $t_0 = t_1$

- ◇ $y_0 = y_1$

⋆ end

The pseudo-code for a non-uniform step size would be a little more complicated, but it would essentially be the same.

⋆ Solve the differential equation

$$\frac{dy}{dt} + 2y = 2 - e^{-4t}, \qquad y(0) = 1$$

```
#include<iostream>
#include<fstream>
#include<cmath>
using namespace std;
ofstream fout("euler1.dat");
double N=500, h=0.1;
double f(double t, double y){
return 2-exp(-4*t)-2*y; }
void euler1(double h, double t0, double y0){
double y=y0;
for (double t=t0;t<=5.0; t+=h){
fout<<t<<" "<<y<<" "<< (1 + 0.5 * exp(-4 * t) - 0.5 * exp(-2 * t)) <<endl;
y+=h*f(t,y); }
}
int main(){
euler1(0.1,0.0,1.0);
return 0; }
```

The obtained result is plotted with the exact result below;

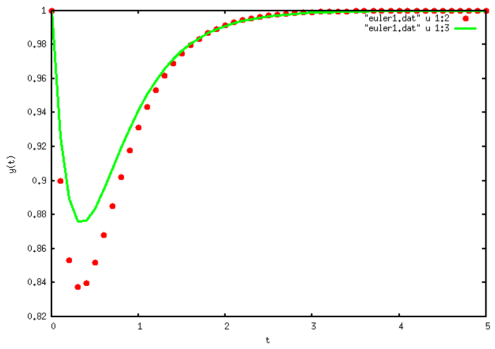


Figure 2: The green line is the exact result and red dot is the calculation using Euler's method.

It is clear from the figure that the Euler's method does not work where the function is oscillatory. And it works better where the function is smooth. The exact solution of the above equation is

$$y(t) = 1 + 0.5e^{-4t} - 0.5e^{-2t}$$

⋆ Let us solve another example;

$$\frac{dy}{dt} - y = -0.5e^{t/2}sin(5t) + 5e^{t/2}cos(5t) \qquad y(0) = 0$$

The exact solution of the above equation is $y(t) = e^{t/2}sin(5t)$

```
#include<iostream>
#include<fstream>
#include<cmath>
using namespace std;
ofstream fout("euler2.dat");
double N=500, h=0.1;
double f(double t, double y){
return y-0.5*exp(0.5*t)*sin(5*t)+5*exp(0.5*t)*cos(5*t); }
void euler1(double h, double t0, double y0){
double y=y0;
for (double t=t0;t<=5.0; t+=h){
fout<<t<<" "<<y<<" "<<(exp(0.5*t)*sin(5*t))<<endl;
y+=h*f(t,y); }
}
int main(){
euler1(0.1,0.0,0.0);
return 0;
}
```

The obtained results are presented in the figure below.

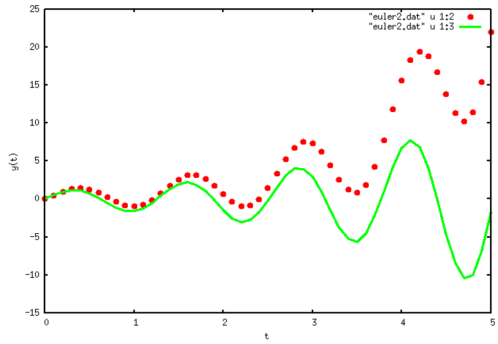


Figure 3: The green line is the exact result and red dot is the calculation using Euler's method.

However, unlike the last example increasing t sees an increasing error. This behavior is fairly common in the approximations. We shouldnt expect the error to decrease as t increases as we saw in the last example. Each successive approximation is found using a previous approximation. Therefore, at each step we introduce error and so approximations should, in general, get worse as t increases.

So, what is the difference betwen y_{n+1} and the real y_{n+1}? Lets look at the accuracy of this method. The standard approach to do this is to expand in Taylor series and compare.

$$y(x_n + h) = y(x_n) + hy'(x_n) + \frac{h}{2^2}y''(x_n) + \mathcal{O}(h^3)$$

$$y_{n+1} - y(x_n + h) = \frac{h}{2^2}y''(x_n) + \mathcal{O}(h^3) = \mathcal{O}(h^2)$$

The error is of order h^2 and this makes it a first order accurate method (the method cancels out errors up to order 1). We can think of this error intuitively, that the method assumes the function is straight, which will systematically wander towards the outside of the curve.

◇ **Backward Euler method:** Instead of using

$$y_{n+1} = y_n + hf(x_n, y_n)$$

Use

$$y_{n+1} = y_n + hf(x_{n+1}, y_{n+1})$$

This time use a backward difference for approximating the derivative at $t = t_{n+1}$. The unknown y_{n+1} appears implicitly in this equation hence named implicit. It still needs to be solved for as a function of y_n.

◇ **Trapezoidal rule:**

$$y_{n+1} = y_n + \frac{h}{2}[f(x_n, y_n) + f(x_{n+1}, y_{n+1})]$$

```
#include<iostream>
#include<fstream>
#include<cmath>
using namespace std;
ofstream fout("euler2.dat");
double N=500, h=0.001;


double f(double t, double y){
return y-0.5*exp(0.5*t)*sin(5*t)+5*exp(0.5*t)*cos(5*t);
}
void euler1(double h, double t0, double y0){
double y1=y0;
double y=0.0;
for (double t=t0;t<=5.0; t+=h){
fout<<t<<" "<<y1<<" "<<y<<" "<<(exp(0.5*t)*sin(5*t))<<endl;
y1+=h*f(t,y);
y+=0.5*h*(f(t,y)+f(t+h,y1)); }
}
int main(){
euler1(0.1,0.0,0.0);


return 0; }
```

- **2nd order ODE:** For second order ODE one need to decouple the equation into two first order ODE. For example

$$\frac{d^2x}{dt^2} = -kx$$

can be written as

$$v = \frac{dx}{dt} \qquad \text{and} \quad \frac{dv}{dt} = -kx$$

And, one apply Euler's method on both v and x simultaneously as follows,

$$x_{n+1} = x_n + f(v, x, t) * h$$

$$v_{n+1} = v_n + f(x, t) * h$$

Simple Harmonic Pendulum

A very widely used problem in computational physics is to solve the 2^{nd} order differential equation of a Simple Harmonic Pendulum. The 2^{nd} order differential equation of a simple pendulum can be given as

$$\frac{d^2\theta}{dt^2} + \frac{g}{L}\theta = 0$$

, One can decouple the equation into two 1^{st} order differential equation as follows:

$$\frac{d\theta}{dt} = \omega$$

$$\frac{d\omega}{dt} = -\frac{g}{L}\theta$$

- One can rightly use Euler's method for this as follows:
  - ⋆ initialize $t = 0$, $\theta(0)$ and $\omega(0)$
  - ⋆ repeat until $t \leq t_{max}$
    - ◇ $\omega(t + dt) = \omega(t) - \frac{g}{L} * \theta(t) * dt$
    - ◇ $\theta(t + dt) = \theta(t) + \omega(t) * dt$
    - ◇ $t = t + dt$

  Euler algorithm is unstable- energy does not conserved in this approach (You Will See in a Few Seconds!!)

- Euler-Cromer method. A very simple modification of Euler's method is done by Cromer, and it work very nicely for 2^{nd} order differential equation. The modification is to **Use the Updated velocity while calculating updated position.** The euler-cromer algorithm is as follows.
  - ⋆ initialize $t = 0$, $\theta(0)$ and $\omega(0)$
  - ⋆ repeat until $t \leq t_{max}$
    - ◇ $\omega(t + dt) = \omega(t) - \frac{g}{L} * \theta(t) * dt$
    - ◇ $\theta(t + dt) = \theta(t) + \omega(t + dt) * dt$
    - ◇ $t = t + dt$

  Euler-Cromer method conserve energy in each period of a simple harmonic motion.

Example:In the following example we will solve the equation of a simple harmonic motion by using both the Euler method and Euler-Cromer method. And will compare the results.

```
#include <cmath>
#include <cstdlib>
#include <fstream>
#include <iostream>
using namespace std;
const double g = 9.80;
const double L = 1;
double t;
double theta;
double omega;
```

```
void EulerStep( double t, double theta, double omega, double dt){
double omegaOld;
ofstream file("euler-file.dat");
for(int i=0;i<=100;i++){
omegaOld=omega;
omega -= (g/L) * theta * dt;
theta += omegaOld * dt;
t += dt;
file<< t <<" " << theta <<" " << omega<<endl; } }
void EulerCromerStep( double t, double theta, double omega,double dt){
ofstream file("cromer-file.dat");
for(int i=0;i <=100;i++){
omega -= (g/L) * theta * dt;
theta += omega * dt;
t += dt;
file<< t <<" " << theta <<" " << omega<<endl;
} }
int main(){
t = 0;
theta = 0;
omega = 1;
double dt=0.1;
EulerCromerStep(t, theta, omega,dt);
EulerStep(t, theta, omega,dt);
return 0; }
```

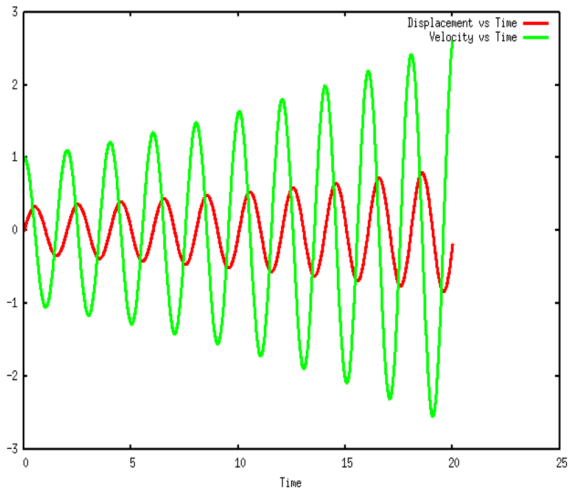


Figure 4: The green line is the Displacement vs Time graph and the Green line is the Velocity vs Time graph, Obtained from the Euler's method.

It is clear from the figure that the displacement amplitude and also the velocity amplitude is increasing with time, which means that the energy is not conserved in the system. Now let us see the same results obtained in the Euler-Cromer method. In the Euler-Cromer method (figure below) we see that both the displacement and the velocity amplitude remain constant over time. So, in the Euler-Cromer method the energy remain constant.

One can plot the phase diagram obtained in both the above to check if energy really conserve or not.

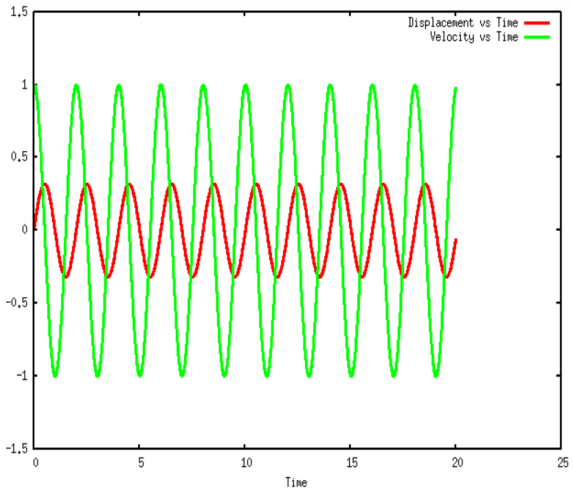


Figure 5: The green line is the Displacement vs Time graph and the Green line is the Velocity vs Time graph, Obtained from the Euler-Cromer method.

Runge-Kutta Method

Runge-Kutta (RK) methods are based on Taylor expansion formulae, but yield in general better algorithms for solutions of an ODE. The basic philosophy is that it provides an intermediate step in the computation of y_{i+1}
Consider the following differential equation

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

and

$$y(t) = \int f(t, y)dt$$

and

$$y_{i+1} = y_i + \int_{t_i}^{t_{i+1}} f(t, y)dt$$

To demonstrate the philosophy behind RK methods, let us consider the second-order RK method, RK2. The first approximation consists in Taylor expanding $f(t, y)$ around the center of the integration interval t_i to t_{i+1}, i. e at $t_i + h/2$, h being the step size. Using the midpoint formula for an integral, defining $y(t_i + h/2) = y_{i+1/2}$ and $t_i + h/2 = t_{i+1/2}$, we obtain

$$\int_{t_i}^{t_{i+1}} f(t, y)dt \approx hf(t_{i+1/2}, y_{i+1/2}) + O(h^3)$$

Which means that we have

$$y_{i+1} = y_i + hf(t_{i+1/2}, y_{i+1/2}) + O(h^3)$$

However, we do not know the value of $y_{i+1/2}$. Here comes thus the next approximation, namely, we use Euler's method to approximate $y_{i+1/2}$. We then have

$$y_{(i+1/2)} = y_i + \frac{h}{2}\frac{dy}{dt} = y(t_i) + \frac{h}{2}f(t_i, y_i)$$

This means that we can define the following algorithm for the second-order Runge-Kutta method, RK2.

$$k_1 = hf(t_i, y_i)$$

,

$$k_2 = hf(t_{(i+1/2)}, y_i + k_1/2)$$

with the final value

$$y_{i+1} \approx y_i + k_2 + O(h^3).$$

The difference between the previous one-step methods is that we now need an intermediate step in our evaluation, namely $t_{(i+h/2)} = t_{(i+1/2)}$ where we evaluate the derivative f. This involves more operations, but the gain is a better stability in the solution. The fourth-order Runge-Kutta, RK4, which we will employ in the solution of various differential equations below, is easily derived. The steps are as follows. We start again with the equation

$$y_{i+1} = y_i + \int_{t_i}^{t_{i+1}} f(t, y)dt$$

but instead of approximating the integral with the midpoint rule, we use now Simpsons rule at $t_i + h/2$, h being the step. Using Simpsons formula for an integral, defining $y(ti + h/2) = y_{(i+1/2)}$ and $t_i + h/2 = t_{(i+1/2)}$, we obtain

$$\int_{t_i}^{t_{i+1}} f(t, y)dt \approx \frac{h}{6}[f(t_i, y_i) + 4f(t_{i+1/2}, y_{i+1/2}) + f(t_{i+1}, y_{i+1})] + O(h^5)$$

This means that, we have

$$y_{i+1} = y_i + \frac{h}{6}[f(t_i, y_i) + 4f(t_{i+1/2}, y_{i+1/2}) + f(t_{i+1}, y_{i+1})] + O(h^5)$$

However, we do not know the values of $y_{i+1/2}$ and y_{i+1}. The fourth-order Runge-Kutta method splits the midpoint evaluations in two steps, that is we have

$$y_{i+1} = y_i + \frac{h}{6}[f(t_i, y_i) + 2f(t_{i+1/2}, y_{i+1/2}) + f(t_{i+1}, y_{i+1}) + 2f(t_{i+1/2}, y_{i+1/2}) + f(t_{i+1}, y_{i+1})]$$

since we want to approximate the slope at $y_{i+1/2}$ in two steps. The first two function evaluations are as for the second order Runge-Kutta method. The algorithm is as follows

- ⋆ We compute first

  $$k_1 = hf(t_i, y_i)$$

  which is nothing but the slope at t_i. If we stop here we have Euler's method.

- ⋆ Then we compute the slope at the midpoint using Eulers method to predict $y_{i+1/2}$, as in the secondorder Runge-Kutta method. This leads to the computation of

  $$k_2 = hf(t_i + h/2, y_i + k_1/2).$$

- ⋆ The improved slope at the midpoint is used to further improve the slope of $y_{i+1/2}$ by computing

  $$k_3 = hf(t_i + h/2, y_i + k_2/2).$$

- ⋆ With the latter slope we can in turn predict the value of y_{i+1} via the computation of

  $$k_4 = hf(t_i + h, y_i + k_3).$$

⋆ The final algorithm becomes then

$$y_{i+1} = y_i + \frac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4)$$

Thus, the algorithm consists in first calculating k_1 with t_i, y_1 and f as inputs. Thereafter, we increasethe step size by $h/2$ and calculate k_2, then k_3 and finally k_4. With this caveat, we can then obtain the new value for the variable y. It results in four function evaluations, but the accuracy is increased by two orders compared with the second-order Runge-Kutta method. The fourth order Runge-Kutta methodhas a global truncation error which goes like $O(h^4)$. Fig. gives a geometrical interpretation of the fourth-order Runge-Kutta method.

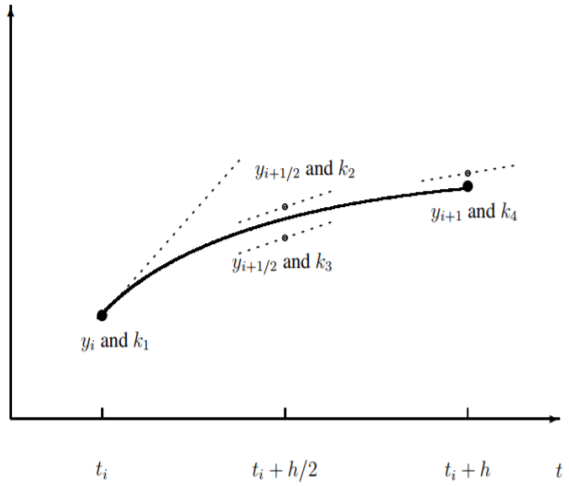


Figure 6: Geometrical interpretation of the fourth-order Runge-Kutta method. The derivative is evaluated at four points, once at the intial point, twice at the trial midpoint and once at the trial endpoint. These four derivatives constitute one Runge-Kutta step resulting in the final value.

Example1:
Solve the differential equation

$$\frac{dx}{dt} = t\sqrt{x}$$

with the initial condition $t_0 = 0$, and $x_0 = 1$. Given the exact solution is

$$x = \frac{(t^2 + 4)^2}{16}$$

Solution:

```
#include<iostream>
#include<fstream>
#include<cmath>

using namespace std;

double f(double indep, double dep){
return indep*sqrt(dep);
```

```
}
double rk4(double h, double t0, double x0){
double k1=h*f(t0,x0);
double k2=h*f(t0+(h/2.0), x0+(k1/2.0));
double k3=h*f(t0+(h/2.0),x0+(k2/2.0));
double k4=h*f(t0+h,x0+k3);
return x0+(k1+2*k2+2*k3+k4)/6.0;
}
int main(){
ofstream fout("rk1.dat");
double t=0.0, x=1.0, dt=0.1;
for(t=0.0;t≤10.0; t+=dt){
fout<<t<<" "<<x<<" "<<((t*t)+4)*((t*t)+4)/16.0<<endl;
x=rk4(dt,t,x);
}
return 0;
}
```

Example 2:
Solve the 2^{nd} order differential equation

$$\frac{d^2x}{dt^2} = -\frac{kx}{m}$$

using rk4 method. Use the initial condition $t_0 = 0$, $x_0 = 0.5$ $v_0 = 4.0$ and also given m=7.2 and k=150.
Note: One need to write this 2^{nd} order differential equation into two coupled 1^{st} order differential equation. $\frac{dx}{dt} = v$ and $\frac{dv}{dt} = kx/m$.

```
#include<iostream>
#include<fstream>
#include<cmath>


using namespace std;
ofstream fout("rk2.dat");
double k=150.0, m=7.2, t=0.0, x=0.5, v=4.0, dt=0.01;


double f(double t, double x, double v){
return v;
}
double g(double t, double x, double v){
return -k*x/m;
}
void rk4(double h, double t0, double x0, double v0){
double k1=h*f(t0,x0,v0);
double l1=h*g(t0,x0,v0);
double k2=h*f(t0+(h/2.0), x0+(k1/2.0),v0+(l1/2.0));
double l2=h*g(t0+(h/2.0), x0+(k1/2.0),v0+(l1/2.0));
double k3=h*f(t0+(h/2.0),x0+(k2/2.0),v0+(l2/2.0));
double l3=h*g(t0+(h/2.0),x0+(k2/2.0),v0+(l2/2.0));
```

```
double k4=h*f(t0+h,x0+k3,v0+l3);
double l4=h*g(t0+h,x0+k3,v0+l3);
t=t0+h;


x= x0+(k1+2*k2+2*k3+k4)/6.0;
v= v0+(l1+2*l2+2*l3+l4)/6.0;
fout<<t<<" "<<x<<" "<<v<<endl;
}
int main(){
fout<<t<<" "<<x<<" "<<v<<endl;
while(t¡5){
rk4(dt,t,x,v);
}
return 0;
}
```

Results:
The time vs displacement and time vs velcity graph,

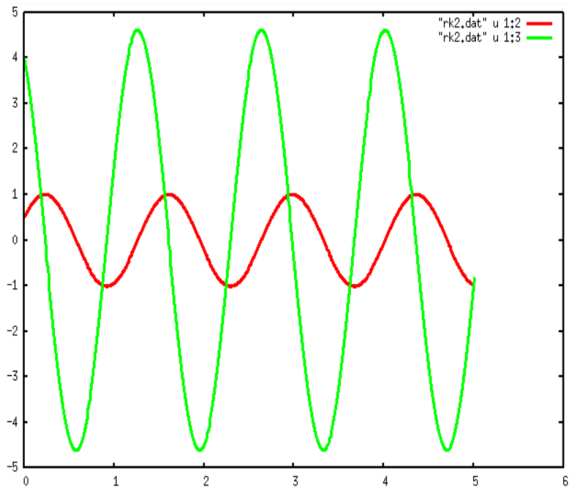


Figure 7: Time vs displacement(small amplitude), and time vs velocity(big amplitude) graph.

The phase digram is shown below

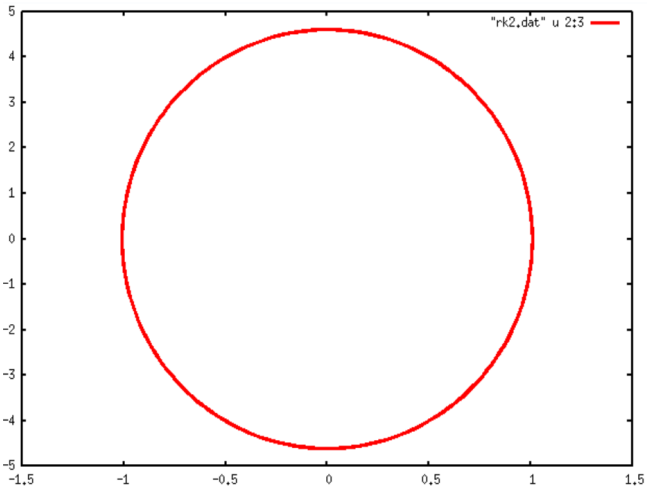


Figure 8: displacement vs velocity. The circular shape of the graph ensure the energy conservation during the SHM

Verlet Algorithm

In this section, the Störmer-Verlet method is presented, with some of its interesting properties, in the special case of a second order ODE of the form $\ddot{q} = F(q)$, where the right-hand side does not depend on q. This equation is common in physics.

history: Verlet integration is a numerical method used to integrate Newton's equations of motion. It is frequently used to calculate trajectories of particles in molecular dynamics simulations and video games. The verlet integrator offers greater stability than the much simpler Euler method, as well as other properties that are important in physical systems such as time-reversibility and area preserving properties. At first it may seem natural to simply calculate trajectories using Euler integration. However, this kind of integration suffers from many problems, as discussed at Euler integration. Stability of the technique depends fairly heavily upon either a uniform update rate, or the ability to accurately identify positions at a small time delta into the past. Verlet integration was used by Carl Störmer to compute the trajectories of particles moving in a magnetic field (hence it is also called Störmer's method) and was popularized in molecular dynamics by French physicist Loup Verlet in 1967.

Consider the Taylor expansion of $q(t+h)$,

$$q(t+h) = q(t) + \frac{dq(t)}{dt}h + \frac{1}{2}\frac{d^2q(t)}{dt^2}h^2 + \frac{1}{3!}\frac{d^3q(t)}{dt^3}h^3 + O(h^4)$$

and

$$q(t-h) = q(t) - \frac{dq(t)}{dt}h + \frac{1}{2}\frac{d^2q(t)}{dt^2}h^2 - \frac{1}{3!}\frac{d^3q(t)}{dt^3}h^3 + O(h^4)$$

After adding together we have,

$$q(t+h) = 2q(t) - q(t-h) + \frac{d^2q(t)}{dt^2}h^2 + O(h^4)$$

So we can write $q(t+h) - 2q(t) + q(t-h) = h^2F(q_n)$ It is a two-step method since q_{n+1} is computed using q_n and q_{n-1}. This formulation is not very interesting.

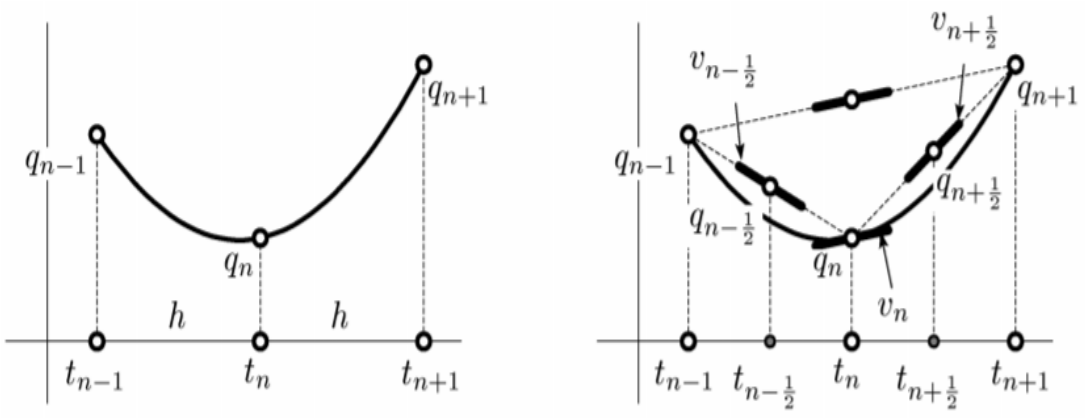


Figure 9: St$\ddot{o}$rmer-Verlet method. Left: two-step formulation. Right: one-step formulation.

We know that we can transform a second order ODE into a first order system of differential equations of dimension 2, in order to do so, we introduce the unknown function $v = \dot{q}$ and the system becomes $v = \dot{q}$, $\dot{v} = F(q)$.
In physics often this type of system is closely related to the motion of a physical body. In this sense, we can see v as the velocity, q as the displacement (as functions of the time) and F describes some kind of force acting on the physical body. From the central difference method we can write

$$v_n = \frac{q_{n+1} - q_{n-1}}{2h}$$

- Position at time $t + \delta t$ require postion of two more previous steps namely at t and $t = \delta t$
- A two steps method, not self-starting.
- It is very simple and stable.
- Global error is of the $O(\delta t^2)$

Velocity Verlet Algorithm: Improved accuracy compared to standard Verlet Start with position and velocity expansions:

$$q(t + h) = q(t) + v(t)h + \frac{1}{2}a(t)h^2 + ...$$

and

$$v(t + h) = v(t) + \frac{1}{2}[a(t) + a(t + h) + ...$$

In each integration cycle:

- Calculate velocity at mid-point

$$v(t + \frac{\delta t}{2}) = v(t) + \frac{1}{2}a(t)\delta t$$

- Calculate position at the next step

$$q(t + \delta t) = q(t) + v(t + \frac{\delta t}{2})\delta t$$

$$q(t + \delta t) = q(t) + [v(t) + \frac{\delta t}{2}a(t)]\delta t$$

Now from the orginal forced equation $a(t) = F(q(t))$, So,

$$q(t+\delta t) = q(t) + [v(t) + \frac{\delta t}{2}F(q(t))]\delta t$$

Which can be write as

$$q_{n+1} = q_n + h[v_n + \frac{h}{2}F(q_n)]$$

- Update velocity

$$v(t+\delta t) = v(t+\frac{\delta t}{2}) + \frac{\delta t}{2}a(t+\delta t)$$

Which can be write as

$$v_{n+1} = v_n + \frac{h}{2}F(q_n) + \frac{h}{2}F(q_{n+1})$$

Example-1: Solve the differential equation of SHM by using Verlet algorithm:

```
# include<iostream>
# include<cmath>
# include<fstream>
using namespace std;
ofstream fout ("verlet.dat");
double k=150.0,m=7.2,t=0.0,x=0.5,v=4.0,h=0.01;
double fn1(double t,double x,double v) {
double F=(-k*x)/m;
return F; }
double fn2(double t,double x,double v)
{ return v; }
void verlet(double h,double t0,double x0,double v0) {
for(double t=t0;t<10;t+=h)
{
double xold=x;
cout<<t<<" "<<x<<" "<<v<<endl;
fout<<t<<" "<<x<<" "<<v<<endl;
x+=h*v+((h*h)/2.0)*fn1(t,x,v);
v+=(h/2.0)*(fn1(t,xold,v)+fn1(t,x,v));
}
}
int main() {
verlet(h,t,x,v);
return 0; }
```

Example-2 Verlet method to solve dampled harmonic motion:

```
# include<iostream>
# include<cmath>
# include<fstream>
using namespace std;
double verlet(double m,double gamma,double k,double t) {
double h=0.01,x=2,v=0;
```

```
    for(double i=0;i<=t; i+=h)
    {
    double xold=x;
    x+=h*v+((h*h)/2.0)*((k*x+gamma*v)*h)/m;
    v-=(h/(2*m))*((k*xold+gamma*v)+(k*x+gamma*v));
    }
    return x;
    }
    int main()
    {
    ofstream fout ("damp-verlet.dat");
    double dt=0.1;
    for(double t=0; t<=50;t+=dt)
    {
    cout <<t<<" "<< verlet(1,sqrt(0.5),2,t) <<" "<< verlet(1,sqrt(8),2,t) <<" "<< verlet(1,sqrt(18),2,t)
    fout <<t<<" "<< verlet(1,sqrt(0.05),2,t) <<" "<< verlet(1,sqrt(8),2,t) <<" "<< verlet(1,sqrt(18),2,t
endl;
    }
    return 0;
    }
```

Department of Physics
University of Dhaka
Computational Physics
Random Number

- The numbers that generates in a sequence which have no relation with each other. Random numbers are useful for a variety of purposes, such as generating data encryption keys, simulating and modeling complex phenomena and for selecting random samples from larger data sets. They have also been used aesthetically, for example in literature and music, and are of course ever popular for games and gambling. When discussing single numbers, a random number is one that is drawn from a set of possible values, each of which is equally probable, i.e., a uniform distribution. When discussing a sequence of random numbers, each number drawn must be statistically independent of the others. **We do not get actually random but a Pseudo Random Number**.

- **Pseudo Random Number Generator:** As the word pseudo suggests, pseudo-random numbers are not random in the way you might expect, at least not if you're used to dice rolls or lottery tickets. Essentially, PRNGs are algorithms that use mathematical formulae or simply precalculated tables to produce sequences of numbers that appear random. A good example of a PRNG is the linear congruential method. A good deal of research has gone into pseudo-random number theory, and modern algorithms for generating pseudo-random numbers are so good that the numbers look exactly like they were really random. It usually use the following algorithm $X_{n+1} = (aX_n + b)\%m$,
  Where X is a sequence of random value,
  m is a positive number,
  a is a multiplier $0 < a < m$,
  c is the increment $0 \leq c < m$,
  X_0 is the seed $0 \leq X_0 < m$
  For example if one use a=7, b=0, m=11, and X_0=1 he/she is suppose to get the numbers {7, 5, 2, 3, 10, 4, 6, 9, 8, 1}, Once he/she get 1 which is same as our seed, the number sequence will repeat. Since all integers are less than m the sequence must repeat after at least m-1 iterations, i.e. the maximal period is m -1. ($x_0 = 0$ is a fixed point and cannot be used.)

  - ⋆ The numbers follow a sequence.
  - ⋆ It follow same sequence if started from the same seed.
  - ⋆ By changing the seed one can change the randomness.
  - ⋆ No correlations
  - ⋆ Long periods
  - ⋆ Follow well-defined distribution
  - ⋆ Fast implementation
  - ⋆ Reproductibility

  We can generate a pseudo-random number in the range from 0.0 to 32,767 using —bf rand() function from $< cstdlib >$ library. The maximum value is library-dependent, but is guaranteed to be at least 32767 on any standard library implementation. We can check it from RAND_MAX.

```
#include< iostream >
#include< cstdlib >
using namespace std;
int main()  {
int x;
x=rand();
cout<<x;
x=rand();
cout<<x;
cout<<RAND_MAX;   //To check the maximum value of the random number.
return 0;}
```

- We can set the range of generated numbers using % (modulus) operator by specifying a maximum value. For instance, to generate a whole number within the range of **1 to 100:**

```
#include < iostream >
#include < cstdlib >
using namespace std;

int main()
{
int i = 0;
while(i++ < 10) {
int r = (rand() % 100) + 1;
cout << r <<" "; }
return 0;
}
```

- One can also generate random number in a certain range, for example if one wants to generate number between -9 to +9 he may write the following code:

```
#include < iostream >
#include < cstdlib >
using namespace std;

int main()
{
int i = 0;
while(i++ < 10) {
int r = (rand() % 19) + (-9);
cout << r <<" "; }
return 0;
}
```

There are 19 distinct integers between -9 and +9 (including both), that is why you need to get the modulous of 19)

- To increase the randomness of the generated numbers one shoul use **SEEDING**, is will increase the randomness of the gnerated numbers.

```
#include < iostream >
#include < cstdlib >
using namespace std;

int main()
{
int i = 0, seed=12345;
srand(seed);
while(i++ < 10) {
int r = (rand() % 100) + 1;
cout << r <<" "; }
return 0;
}
```

By changing the seed to a different value you can get different sequence of number.

- To increase the randomness further, people use the computer time as seed. For this one need to use the library < *ctime* >, Please see the code below:

```
#include < iostream >
#include < cmath >
#include < cstdlib >
#include < ctime >
using namespace std;

int main()
{
int i = 0;
srand(time(NULL));
while(i++ < 10) {
int r = (rand() % 100) + 1;
cout << r << " ";
}
return 0;
}
```

- **Monte Carlo Integration Method**

  - ⋆ **Brief history of the Monte Carlo method** The idea of the Monte Carlo (MC) method is a lot older than the computer. The name Monte Carlo is relatively new - it was suggested by Nicolas Metropolis in 1949 because of the similarity of statistical simulation to games of chance (Monaco was the center of gambling). Under the name statistical sampling, the MC method stretches back to times when numerical calculations were performed using pencil and paper. At first, Monte Carlo was a method for estimating integrals which could not be solved by other means. Integrals

over poorly-behaved functions and multidimensional integrals were profitable subjects of the MC method. The famous physicist Richard Feynman realized around the time of the Second World War that the time of electronic computing was just around the corner. He created what could be described as a highly pipelined human CPU, by employing a large number of people to use mechanical adding machines in an arithmetic assembly line. A number of crucial calculations to the design of the atomic bomb were performed in this way. It was in the last months of the Second World War when the new ENIAC electronic computer was used for the first time to perform numerical calculations. The technology that went into ENIAC had existed even before but the war had slowed down the construction of the machine. The idea of using randomness for calculations occurred to Stan Ulam while he was playing a game of cards. He realized that he could calculate the probability of a certain event simply by repeating the game over and over again. From there it was a simple step to realize that the computer could play the games for him. This seems obvious now, but it is actually a subtle question that a physical problem with an exact answer can be approximately solved by studying a suitable random process. Nowadays Monte Carlo has grown to become the most powerful method for solving problems in statistical physics - among many other applications. The name Monte Carlo is used as a general term for a wide class of stochastic methods. The common factor is that random numbers are used for sampling.

⋆ **Monte Carlo integration - simple sampling (Hit or Miss method):** One of the simplest but also effective uses of the Monte Carlo method is the evaluation of integrals which are intractable by analytic techniques. In the simplest case, we wish to obtain the one-dimensional integral of $f(x)$ within the interval [a,b], i. e $I = \int_a^b f(x)dx$ One-dimensional integrals can be effectively calculated using discrete approximations such as the trapezoidal rule or Simpsons rule. In order to illustrate the MC integration technique, we apply it first to the one-dimensional case and then extend the discussion to multidimensional integrals (where the other methods become computationally very expensive and thus less effective). In the so-called "hit-or-miss" Monte Carlo integration, the definite integral is estimated by drawing a box which bounds the function f(x) in the interval [a,b]; i.e. the box extends from **a** to **b** and from **0** to **fmax** where **fmax** $>$ **f(x)** throughout the interval. Then **N** points are dropped randomly into the box. An estimate for the integral can now be obtained by calculating the total number of points **N0** which fall under the curve of f(x); i.e.

$$I_{est} = \frac{N_0}{N} R$$

where $R = (b-a) * fmax$ be the total area of the box. Each of N random points is obtained by generating 2 uniformly distributed random numbers s_1 and s_2 and taking $x = a + s_1 * (b-a)$
$y = s_2 * fmax$
as the x and y coordinates of the point.

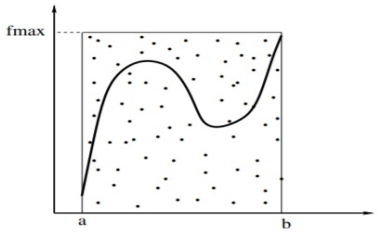


Figure 1: The hit and miss method, Points are generated randomly inside a rectangular area which bounds the function f(x) in the interval [a,b]. The number of points under the curve is calculated to obtain an estimate of the integral.

The estimate of the integral becomes increasingly precise as $N \to \infty$ and will eventually converge to the correct answer. Obviously, the quality of the answer depends on the quality of the random number generator sequence which is used. We can obtain independent estimates by repeating the calculation using different random number sequences. Comparing the values gives an idea of the precision of the calculation.

- **Calculation of PI:** A simple example of the use of random numbers is the calculation of π. If we put points within a square, with center at the origin and sides having length two times unity at random, then the number of point within a unit circle with center at the origin to that within the square, will be equal to the ratio of the areas of the unit circle to that of the square. i. e. $\pi/4$. We generate N random points in the square $x \in [-1, 1]$ and $y \in [-1, 1]$. Then we calculate how many of those points landed inside the circle $x^2 + y^2 = 1$ Denote this number by N_0. The ratio of the two areas is

$$\frac{A_{circle}}{A_{square}} = \frac{\pi R^2}{4R^2} = \frac{N_0}{N}$$

Thus

$$\pi = 4\frac{N_0}{N}$$

The following program calculates π using this algorithm.

```
#include < iostream >
#include < cmath >
#include < cstdlib >
#include< ctime >
using namespace std;


double getmePi(int N){ int i=0, seed, count=0;
double x=0.0, y=0.0;
seed= time(NULL);
srand(seed); for(i=1;i<=N;i++) {
x=(double) rand()/(double) RAND_MAX;
y=(double) rand()/(double) RAND_MAX;
RAND_MAX is the maximum random integer number created by rand ().
if( sqrt(x*x+y*y)<=1.0) count++;
}
return 4.0*(double) count/(double)N;
```

```
}
int main(){
int i=0; double val=0.0;
for(i=1000;i<=1000000;i*=10){
val=getmePi(i);
cout<<val<<endl;}
return 0;}
```

- Calculate the integral

$$I = \int_{-\pi}^{\pi} \frac{(1 + cosx)sin|2x|}{1 + |sin(2x)|} dx$$

```
#include < iostream >
#include < cmath >
#include < cstdlib >
#include< ctime >
using namespace std;


double func(double x){
return (1+cos(x))*sin(abs(2*x))/(1+abs(sin(2*x)));
}
double MC1D(double a, double b, double c, double d, int N){
int i;
double x, y, area;
int S=0;
for(i=1;i<=N;i++){
x = a + (b - a) * (rand()/(double)RAND_MAX);
y = c + (d - c) * (rand()/(double)RAND_MAX);
if(y <= func(x))S++;
area=((double)S/(double)N)*(b-a)*(d-c);}
return area;
}


int main() {
srand(time(NULL));
cout << MC1D(-M_PI, M_PI, -0.3, 0.9, 10000) << endl;
return 0;}
```

- Volume of a sphere of radius one. (centered at the origin)

```
#include <iostream>
#include <fstream>
#include <cstdlib>
#include <cmath>

using namespace std;
```

```
main () {
     double vol;
     double xin;
     double yin;
     double zin;
     double xout;
     double yout;
     double zout;
     double hit;
     srand48(time(NULL));

     ofstream out("hits.txt");
     ofstream out2("misses.txt");
     for (int i = 0; i < 1000; i++) {
         double x = 2*(drand48()-.5);
         double y = 2*(drand48()-.5);
         double z = 2*(drand48()-.5);
         double r = sqrt(x*x + y*y + z*z);
         if (r < 1.0) {
             xin = x;
             yin = y;
             zin = z;
             hit += 1;
         } else {
             xout = x;
             yout = y;
             zout = z;
         }
         out << xin << " " << yin << " " << zin << endl;
         out2 << xout << " " << yout << " " << zout << endl;
     }
     out.close();
     vol = 8 * (hit / 1000);
     cout<<hit<<endl;
     cout << "Volume: " << vol << endl;
     cout << "Actual volume: " << 4*M_PI / 3 << endl;
    cout << "Fractional error: " << fabs( (4*M_PI / 3) - vol) << endl;
}
```

- **Running various functions at the same time:** If you need to integrate various different functions at the same tieme with the same method, for example with mote carlo method, you can proceed as follows:

  ⋆ define all your functions with their return types one by one with a common name and some extension. For Example,
  double func_1(double x){ return .....}
  double func_2(double x){ return .....}
  and so on .....

- ⋆ Define the integration function (monte carlo function here) with the guest function(function to be integrated) as the first argument with the common name. Example:
  double MC_1D(**double func(double x),** double a, double b, double c, double d, int N).
- ⋆ Call the integration function in the main function, with the **full name** of the function to be integrated as the first argumnet. For example: Now this time if you want to get the func_1 to be integrated,
  MC_1D(func_1, a, b, c, d, N).

- See the example below: Here we are calculating

$$I = \int_{\pi}^{\pi} \frac{(1 + cosx)sin|2x|}{1 + |sin(2x)|} dx$$

and

$$\int_{0}^{\pi/2} \frac{1}{1 + sin^2(x)} dx$$

```
#include < iostream >
#include < cmath >
#include < cstdlib >
#include< ctime >
using namespace std;


double func_1(double x){
return (1+cos(x))*sin(abs(2*x))/(1+abs(sin(2*x)));
}
double func_2(double x){
return 1/(1+sin(x)*sin(x));
}
double MC1D(double func(double x), double a, double b, double c, double d, int N){
int i;
double x, y, area;
int S=0;
for(i=1;i¡=N;i++){
x = a + (b - a) * (rand()/(double)RAND_MAX);
y = c + (d - c) * (rand()/(double)RAND_MAX);
if(y <= func(x))S++;
area=((double)S/(double)N)*(b-a)*(d-c);}
return area;
}


int main() {
srand(time(NULL));
cout << MC1D(func_1, - M_PI, M_PI, -0.3, 0.9, 10000) << endl;
cout << MC1D(func_2,0, M_PI/2, -0.3, 0.9, 10000) << endl;
return 0;}
```