MEP 202

Numerical Methods

2nd semester SY 2003-2004

Compilations of Algorithms

Compiled By:
Jeune Prime M. Origines

Faculty-In-Charge:
Nicanor S. Buenconsejo Jr, PhD

List of Algorithms

Fixed-Point Iteration

Discussion

Fixed-point iteration is one of the known methods for solving the root of a non-linear equation.

Given an equation:

$$f(x) = 0 \tag{1}$$

one must derive (1) into a new form:

$$x = g(x) \tag{2}$$

which will be used for finding the Fixed-Point.

For example:

$$f(x) = x^2 - x - 2 \tag{3}$$

the $g(x)$ can be derived using any of the following formula:

(a) $g(x) = x^2 - 2$

(b) $g(x) = \sqrt{x + 2}$ (4)

(c) $g(x) = 1 + \frac{2}{x}$

(d) $g(x) = x - \frac{x^2 - x - 2}{m}$, where m is a non-zero constant.

Once the $g(x)$ is obtained from (1), one may now apply the Fixed-Point iteration using a starting value p_0 as stated in Theorem 2.1 on p. 46 of the textbook.

n	p_{n+1}	***G(x)***
0	p_1	$g(p_0)$
1	p_2	$g(p_1)$
.	.	.
.	.	.
.	.	.
∞	P	$g(P)$

Table 2.1.1 The sequence pattern for the Fixed-Point Iteration.

Not all iterative functions, *g(x),* will converge to a Fixed-Point P. There are some iterations which may diverge away from the root as $n \to \infty$. In order to check if the given *g(x)* may arrive to a Fixed-Point P, the given *g(x)* shall be tested based on Theorem 2.3 on p. 47 the textbook.

If $|g(P)| \le K < 1$ then the iteration is convergent. (5)

If $|g(P)| \le K < 1$ then the iteration is divergent. (6)

There are 2 types of errors that can be considered in this method: **Absolute** and **Relative** errors. Each can be obtained by:

$$\text{Absolute Error} = |P - p_n| \tag{7}$$

and

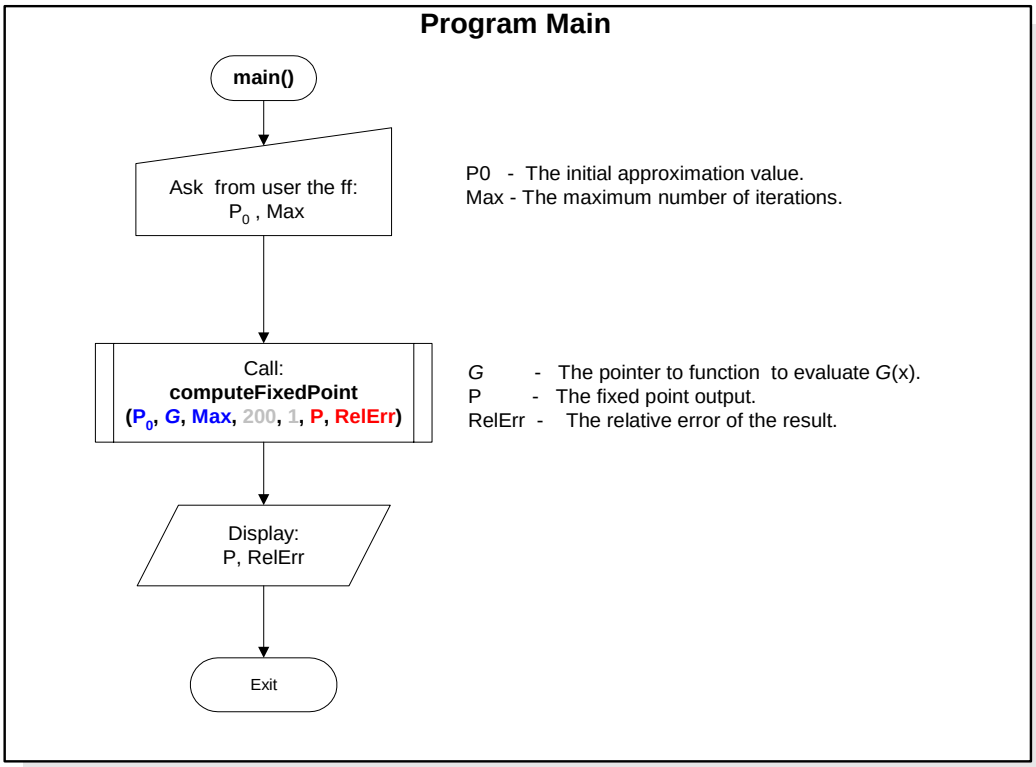
$$\text{Relative Error} = |P_n - p_{n-1}| \tag{8}$$

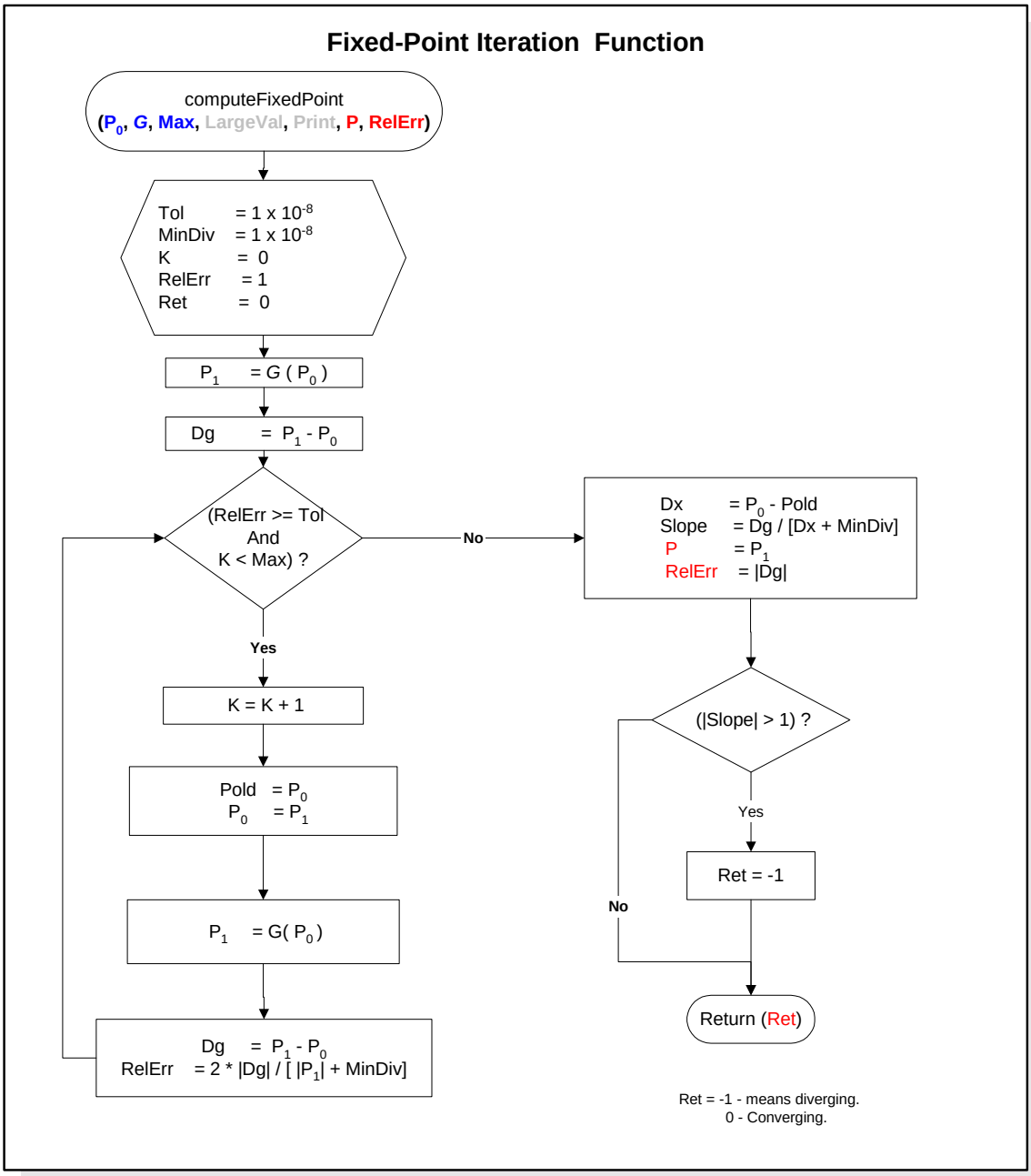
where P is the Fixed-Point value and p_n is the value obtained at ***n**th* iterations.

Ideally, the ***absolute*** error would give the exact deviation of the improved value at a certain number of iterations. But in reality, the Fixed-Point P is yet to be solved which makes the ***relative*** error the only criterion available for terminating the process.

Algorithm 2.1 (Fixed-Point Iteration). To find the solution of the equation $x = g(x)$ by starting with p_0 and iterating $p_{n+1} = g(p_n)$.

Program Flow




Fixed-Point Iteration Function
computeFixedPoint
(P0, G, Max, LargeVal, Print, P, RelErr)
Tol = 1 x 10-8
MinDiv = 1 x 10-8
K = 0
RelErr = 1
Ret = 0
P1 = G (P0)
Dg = P1 - P0
(RelErr >= Tol And K < Max) ?
No
Yes
Dx = P0 - Pold
Slope = Dg / [Dx + MinDiv]
P = P1
RelErr = |Dg|
(|Slope| > 1) ?
Yes
No
Ret = -1
Return (Ret)
K = K + 1
Pold = P0
P0 = P1
P1 = G(P0)
Dg = P1 - P0
RelErr = 2 * |Dg| / [|P1| + MinDiv]
Ret = -1 - means diverging.
0 - Converging.

Program Source Code

```
#include <stdio.h>
#include <conio.h>
#include "FixedPointFunc.h"

double  gFunc(double x);

int main(int argc, char* argv[])
{
    int nStatus = 0;
    double  nP0;
    double  nRelativeError;
    double  nP;
    int     nMaxIter;

    printf("\n     *** Fixed-Point Iteration *** ");
    printf("\n\n Using g(x) = -4 + 4x - x^2/2  ");

    /* Ask for input */
    printf("\n\n Enter the initial value P0 : ");
    scanf("%lf",&nP0);    fflush(stdin);
    printf(" Enter the maximum number of iterations: ");
    scanf("%d",&nMaxIter);
    fflush(stdin);

    /* Copmpute the the fixed-point iteration */
    nStatus = computeFixedPoint(nP0,gFunc,nMaxIter,200,1,&nP,&nRelativeError);

    /* Display output */
    if(nStatus == 0)
    {
        printf("\n The computed fixed point is : %2.8lf",nP);
        printf("\n The the relative error is   : %2.8lf",nRelativeError);
    }
    else
    {
        printf("\n The sequence appears tobe diverging.");
    }

    printf("\n\nPress any key to continue....");

    getch();
    return 0;
}


/* The g(x) function */
double  gFunc(double x)
{
    return -4 + 4 * x  - pow(x,2)/2;
}
```

```
#include "FixedPointFunc.h"
/* Function name      : computeFixedPoint
* Description        : Computes the x = g(x) using  the Fixed-Point iteration.
* Parameter(s)       :
*     nPterm           [in]  The initial value.
*     gFunction        [in]  The function that will evaluate the g(x)
*     nMaxIter         [in]  The maximum number of iterations.
*     largeVal         [in]  The absolute value used as boundary for the iteration.
*     nPrint           [in]  Flag for printing the output on screen,
*     *pOutput         [out]  The Fixed-Point output.
*     *pDelta          [out]  The relative error  |Pn+1 - Pn|
* Return             :
*    int       -  0  Converging,  -  -1  Diverging.
*/
int _cdecl
computeFixedPoint(double nPterm,double (*gFunction)(double),int nMaxIter,double
largeVal,int nPrint,double *pOutput,double *pDelta)
{
    int nRet = 0;
    static const double nTolerance  = 10e-8; /* The stopping point tolerance */
    static const double nMinDivisor = 10e-8; /* The minimum divisor */
    int      i;
    double   nRelError  = 1;  /* program relative error
                                 Note: the real relative error is g(Pn+1) - g(Pn) */
    double   nPnew;           /* the new value */
    double   nPold;           /* the current value */
    double   nDg;             /* The difference between g(Pn+1) - g(Pn)  */
    double   nDx;             /* The difference between Pn+1 - Pn        */
    double   nSlope;          /* The slope of the secant (Pn,g(pn)) (Pn+1,g(Pn+1))*/
       /* Init values */
    *pOutput = 0; *pDelta  = 0;
    nPnew    = gFunction(nPterm);
    nDg = nPnew - nPterm;
    i = 0;
    if(nPrint)
    {
        printf("\n N \t %-13s \t %-13s \t %-13s ","Pn","Pn+1","Relative Error");
        printf("\n-------------------------------------------------------");
        printf("\n %02d \t %5.8lf \t %5.8lf \t %2.8lf ",i,nPterm,nPnew,fabs(nDg));
    }
    /* This has been slightly modified to save some variables
       result is still the same.
    */
    while(nRelError >= nTolerance
                && i < nMaxIter
                && largeVal > fabs(nPnew))
    {
        i++;
        nPold  = nPterm;
        nPterm = nPnew;
        /* compute the Pn+1 */
        nPnew = gFunction(nPterm);
        nDg = nPnew - nPterm;
        nRelError = 2 * fabs(nDg) / (fabs(nPnew) + nMinDivisor);
        if(nPrint)
            printf("\n %02d \t %5.8lf \t %5.8lf \t %2.8lf ",i,nPterm,nPnew,fabs(nDg));
    }
    nDx    = nPterm - nPold;
    nSlope = nDg / (nDx + nMinDivisor);
    *pOutput = nPnew;
```

```
    /* This is the real relative error is g(Pn+1) - g(Pn) */
    *pDelta  = fabs(nDg);
    if(fabs(nSlope) > 1)
        nRet = -1;
    if(nPrint)
        printf("\n-----------------------------------------------------------");
    return nRet;
}
```

Program Output

```
     *** Fixed-Point Iteration ***

 Using g(x) = -4 + 4x - x^2/2

 Enter the initial value P0 : 2.5
 Enter the maximum number of iterations: 40

 N       Pn              Pn+1            Relative Error
-----------------------------------------------------------
 00      2.50000000      2.87500000      0.37500000
 01      2.87500000      3.36718750      0.49218750
 02      3.36718750      3.79977417      0.43258667
 03      3.79977417      3.97995481      0.18018064
 04      3.97995481      3.99979910      0.01984429
 05      3.99979910      3.99999998      0.00020088
 06      3.99999998      4.00000000      0.00000002
-----------------------------------------------------------
 The computed fixed point is : 4.00000000
 The the relative error is   : 0.00000002

Press any key to continue....
```

Bisection Method

Discussion

One of the bracketing methods used in finding the root of the equation *r*, for which *f(r) = 0*, is the Bisection method. Its goal is to enclose the root *r* in narrowing brackets until such interval, with arbitrarily small width that encloses the root, is obtained.

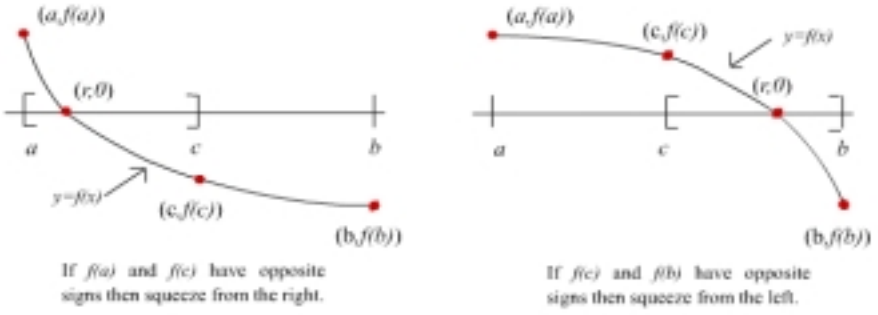


Fig. 2.2.1 The selection for the new interval.

It requires 2 initial intervals [*a*, *b*] with the assumption that *f(a)* and *f(b)* have opposite signs and that *f(x)* is continuous from *a* to *b* as stated.

As its name implies, the method first bisects the interval at point $c = (a + b)/2$, forming into 2 subintervals [*a*, *c*] and [*b*, *c*]. Then, it determines the location of the root *r* between the 2 subintervals based on the following criteria:

If *f(a)* and *f(c)* have opposite signs, the root *r* lies in [*a*, *c*] so the process must be repeated using [*a*, *c*] as the new interval.

If *f(c)* and *f(b)* have opposite signs, the root r lies in [*c*, *b*] so the process must be repeated using [*c*, *b*] as the new interval.

If *f(c) = 0*, the root is found and so the iteration must now stop.

The method proceeds by iteratively reducing the size of the range until the root *r* is found or until the brackets are considerably close enough to the root.

Current Interval	Left Subinterval	Right Subinterval
$[a_0, b_0]$	$[a_0 , c_0]$	$[c_0, b_0]$
$[a_1, b_1]$	$[a_1 , c_1]$	$[c_1, b_1]$

.	.	.
.	.	.
$[a_n, b_n]$	$[a_n , c_n]$	$[c_n, b_n]$

Table 2.2.1 The sequence of nested intervals where the current interval $[a_n, b_n]$ are the new values from either $[a_{n-1}, c_{n-1}]$ or $[c_{n-1}, b_{n-1}]$, for all $n > 0$.

The error bound for an *n* number of iterations is defined by:

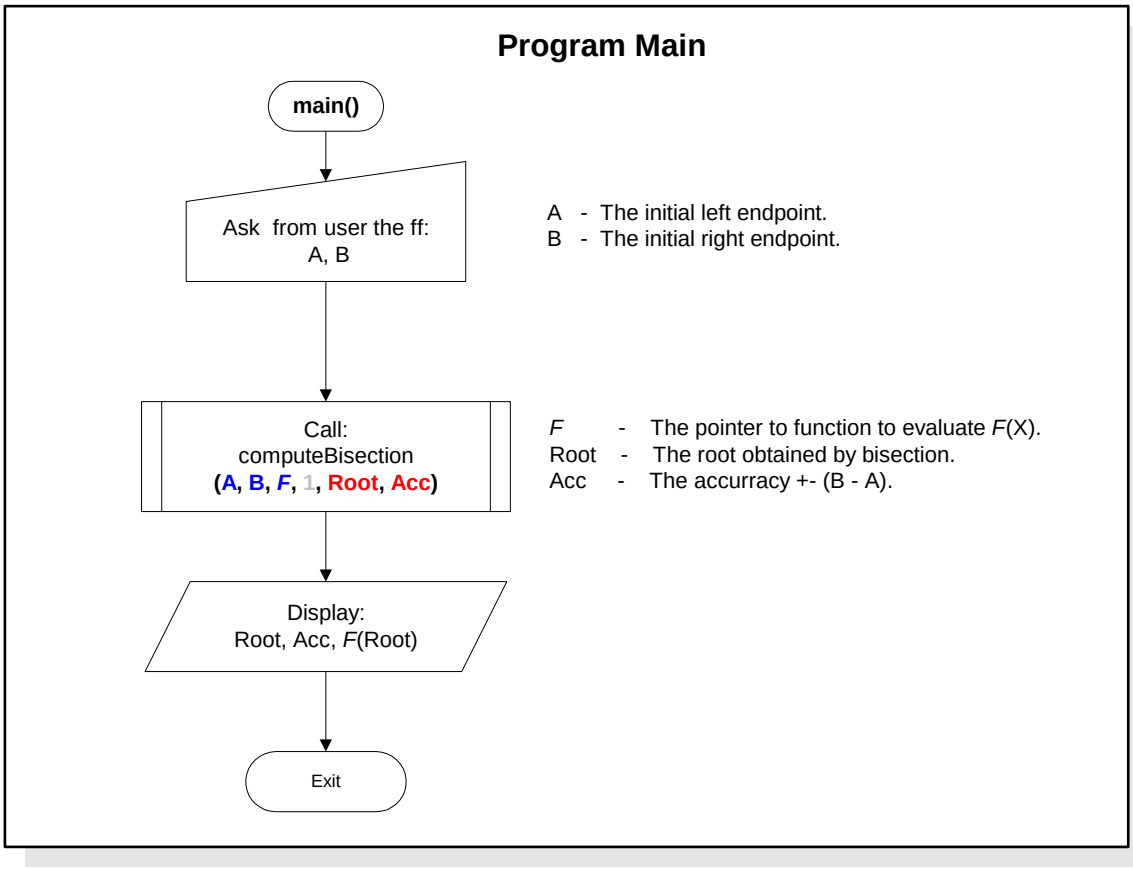
$$|e_n| = |r - c_n| \leq \left| \frac{b_0 - a_0}{2^{n+1}} \right| \tag{1}$$

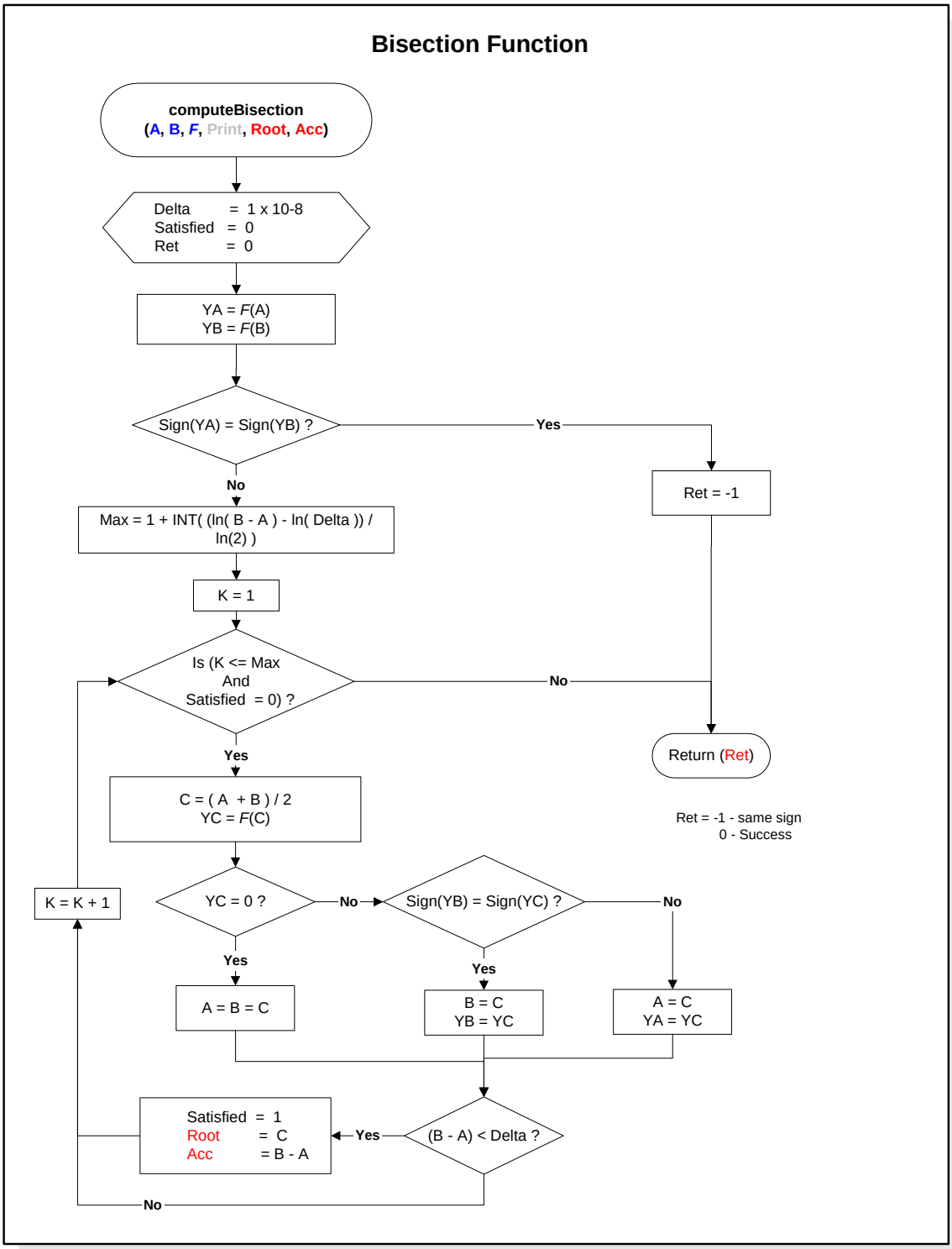
To estimate the number of repetitions required to guarantee that the midpoint C_N is an approximation to a root and has an error less than the preassigned value Delta is:

$$N = \text{int}\left(\frac{\ln(B-A) - \ln(Delta)}{\ln(2)} \right) \tag{2}$$

Algorithm 2.2 (Bisection Method). To find a root of the equation *f(x) = 0* in the interval [*a, b*]. Proceed with the method only if *f(x)* is continuous and *f(a)* and *f(b)* have opposite signs.

Program Flow




Bisection Function
computeBisection
(A, B, F, Print, Root, Acc)
Delta = 1 x 10-8
Satisfied = 0
Ret = 0
YA = F(A)
YB = F(B)
Sign(YA) = Sign(YB) ?
Yes
Ret = -1
No
Max = 1 + INT((ln(B - A) - ln(Delta)) / ln(2))
K = 1
Is (K <= Max And Satisfied = 0) ?
No
Return (Ret)
Yes
C = (A + B) / 2
YC = F(C)
Ret = -1 - same sign
0 - Success
K = K + 1
YC = 0 ?
No
Sign(YB) = Sign(YC) ?
No
Yes
Yes
A = B = C
B = C
YB = YC
A = C
YA = YC
Satisfied = 1
Root = C
Acc = B - A
Yes
(B - A) < Delta ?
No

Program Source Code

```
#include <stdio.h>
#include <conio.h>
#include "BisectionFunc.h"

double  fFunc(double x);

int main(int argc, char* argv[])
{
    int nStatus = 0;
    double  nA;
    double  nB;
    double  nAccuracy;
    double  nRoot;

    printf("\n     *** Bisection Method *** ");
    printf("\n\n Using f(x) = x sin (x) - 1  ");

    /* Ask for input */
    printf("\n\n Enter the value for point A : ");
    scanf("%lf",&nA);  fflush(stdin);
    printf(" Enter the value for point B : ");
    scanf("%lf",&nB);   fflush(stdin);

    /* Compute the bisection method */
    nStatus = computeBisection(nA,nB,fFunc,1,&nRoot,&nAccuracy);

    if(nStatus == 0)
    {
        printf("\n The computed root is : %2.8lf",nRoot);
        printf("\n The accuracy is  +-  : %2.8lf",nAccuracy);
        printf("\n The function value at root is : %2.8lf",fFunc(nRoot));
    }
    else
    {
        printf("\n Error: f(A) and f(B) have same sign:
[%2.8lf,%2.8lf]",fFunc(nA),fFunc(nB));
    }

    printf("\n\nPress any key to continue....");

    getch();
    return 0;
}

/* The f(x) function */
double  fFunc(double x)
{   return ((x  * sin(x)) - 1); }
```

```
#include "BisectionFunc.h"

/* Function name      : computeBisection
* Description        : Computes the root using the Bisection Method.
* Parameter(s)       :
*     nA            [in]  Point A.
*     nB            [in]  Point B.
*     fFunction     [in]  The function to evaluate f(x).
*     nPrint        [in]  Flag for printing the output on screen,
*     pOutput       [out]  The root.
*     pAccuracy     [out]  The accuracy of the root.
* Return             :
*    int        0  - Success, -1 - Same sign.
*/
int _cdecl
computeBisection(double nA,double nB,double (*fFunction)(double),int nPrint,double
*pOutput,double* pAccuracy)
{
    int nRet = 0;
    static const double nDelta  = 10e-8;         /* Tolerance for with of interval */
    int    nSatisfied           = 0;             /* If 1, the calculation is satisfied */
    int      i;
    int      nMaxIter;
    double   nYA,nYB,nYC;
    double   nC;
    nYA = fFunction(nA); nYB = fFunction(nB);

    if(GETSIGN(nYA) == GETSIGN(nYB))  nRet = -1;  /* Check for the signs */

    if(nRet == 0)
    {
        /* Compute the max number of iterations */
        nMaxIter = 1 + (int)((log(nB - nA) - log(nDelta))/log(2));
        if(nPrint)
        {
            printf("\n The computed max iteration is: %d",nMaxIter);
            printf("\n\n N  %-10s \t %-10s  %-10s  %-8s  %-8s  %-8s  "
                  ,"A","B","C","f(A)","f(B)","f(C)");
            printf("\n-------------------------------------------------------------");
        }
        for(i = 1; i <= nMaxIter && nSatisfied == 0 ; i++)
        {
            /* Get the mid point */
            nC = (nA + nB) / 2;
            nYC = fFunction(nC);

            if(nPrint)
                printf("\n %2d  %2.8lf  %2.8lf  %2.8lf  %2.6lf  %2.6lf  %2.6lf "
                      ,i,nA,nB,nC,nYA,nYB,nYC);

            if(nYC == 0)  /* Root is found */
            {
                nA = nC;  nB = nC;
            }
            else if(GETSIGN(nYB) == GETSIGN(nYC))
            {
                nB  = nC; nYB = nYC;
            }
            else
            {
                nA  = nC; nYA = nYC;
            }
```

```
            if((nB - nA) < nDelta)
            {
                nSatisfied = 1;
                *pOutput = nC;
                *pAccuracy = (nB - nA);
            }
        }
        if(nPrint)
            printf("\n----------------------------------------------------------");
    }
    return nRet;
}
```

Program Output

```
     *** Bisection Method ***

 Using f(x) = x sin (x) - 1

 Enter the value for point A : 1
 Enter the value for point B : 2

 The computed max iteration is: 24

 N  A           B           C           f(A)      f(B)      f(C)
-------------------------------------------------------------
  1  1.00000000  2.00000000  1.50000000  -0.158529  0.818595  0.496242
  2  1.00000000  1.50000000  1.25000000  -0.158529  0.496242  0.186231
  3  1.00000000  1.25000000  1.12500000  -0.158529  0.186231  0.015051
  4  1.00000000  1.12500000  1.06250000  -0.158529  0.015051  -0.071827
  5  1.06250000  1.12500000  1.09375000  -0.071827  0.015051  -0.028362
  6  1.09375000  1.12500000  1.10937500  -0.028362  0.015051  -0.006643
  7  1.10937500  1.12500000  1.11718750  -0.006643  0.015051  0.004208
  8  1.10937500  1.11718750  1.11328125  -0.006643  0.004208  -0.001216
  9  1.11328125  1.11718750  1.11523438  -0.001216  0.004208  0.001496
 10  1.11328125  1.11523438  1.11425781  -0.001216  0.001496  0.000140
 11  1.11328125  1.11425781  1.11376953  -0.001216  0.000140  -0.000538
 12  1.11376953  1.11425781  1.11401367  -0.000538  0.000140  -0.000199
 13  1.11401367  1.11425781  1.11413574  -0.000199  0.000140  -0.000030
 14  1.11413574  1.11425781  1.11419678  -0.000030  0.000140  0.000055
 15  1.11413574  1.11419678  1.11416626  -0.000030  0.000055  0.000013
 16  1.11413574  1.11416626  1.11415100  -0.000030  0.000013  -0.000009
 17  1.11415100  1.11416626  1.11415863  -0.000009  0.000013  0.000002
 18  1.11415100  1.11415863  1.11415482  -0.000009  0.000002  -0.000003
 19  1.11415482  1.11415863  1.11415672  -0.000003  0.000002  -0.000001
 20  1.11415672  1.11415863  1.11415768  -0.000001  0.000002  0.000001
 21  1.11415672  1.11415768  1.11415720  -0.000001  0.000001  0.000000
 22  1.11415672  1.11415720  1.11415696  -0.000001  0.000000  -0.000000
 23  1.11415696  1.11415720  1.11415708  -0.000000  0.000000  -0.000000
 24  1.11415708  1.11415720  1.11415714  -0.000000  0.000000  -0.000000
-------------------------------------------------------------
 The computed root is : 1.11415714
 The accuracy is  +-  : 0.00000006
 The function value at root is : -0.00000000

Press any key to continue....
```

False Position or Regula Falsi Method

Discussion

Another popular method for finding the root *r* of the equation, where *f(r) = 0*, is the **False Position or Regula Falsi** method. It is similar to the bisection method since it operates by enclosing the root *r* in a narrowing bracket. But the method of subdividing the bracket is somewhat different. It converges faster to the root because it is an algorithm which uses appropriate weighting of the end points *a* and *b* using the information about the function. In other words, finding *c* is a *static* procedure in the case of the bisection method since for a given *a* and *b*, it gives identical *c*, no matter what function one wishes to solve for. On the other hand, the false position method uses the information about the function to arrive at *c*.

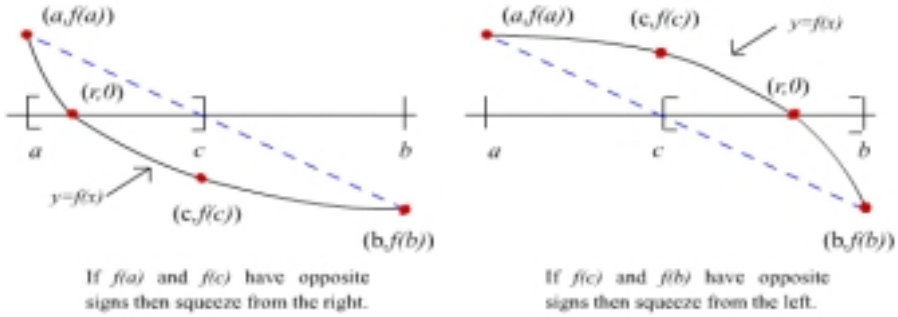


Fig. 2.3.1 The selection for the new interval.

As shown in the figure above, *c* is obtained from the intersection of the horizontal axis of the straight line joining (*a, f(a)*) and (*b, f(b)*). By equating the slopes of the lines: (*a, f(a)*), (*b, f(b)*) which is $m = \frac{f(b) - f(a)}{b - a}$ and (*c, 0*) , (*b, f(b)*) which is $m = \frac{0 - f(b)}{c - b}$; the formula for solving *c* will be obtained.

$$c = b - \frac{f(b)(b-a)}{f(b) - f(a)} \tag{1}$$

After obtaining the value for *c*, the same criteria used in the bisection method are applied to determine the location of the root between the 2 subintervals [*a, c*] and [*c, b*].

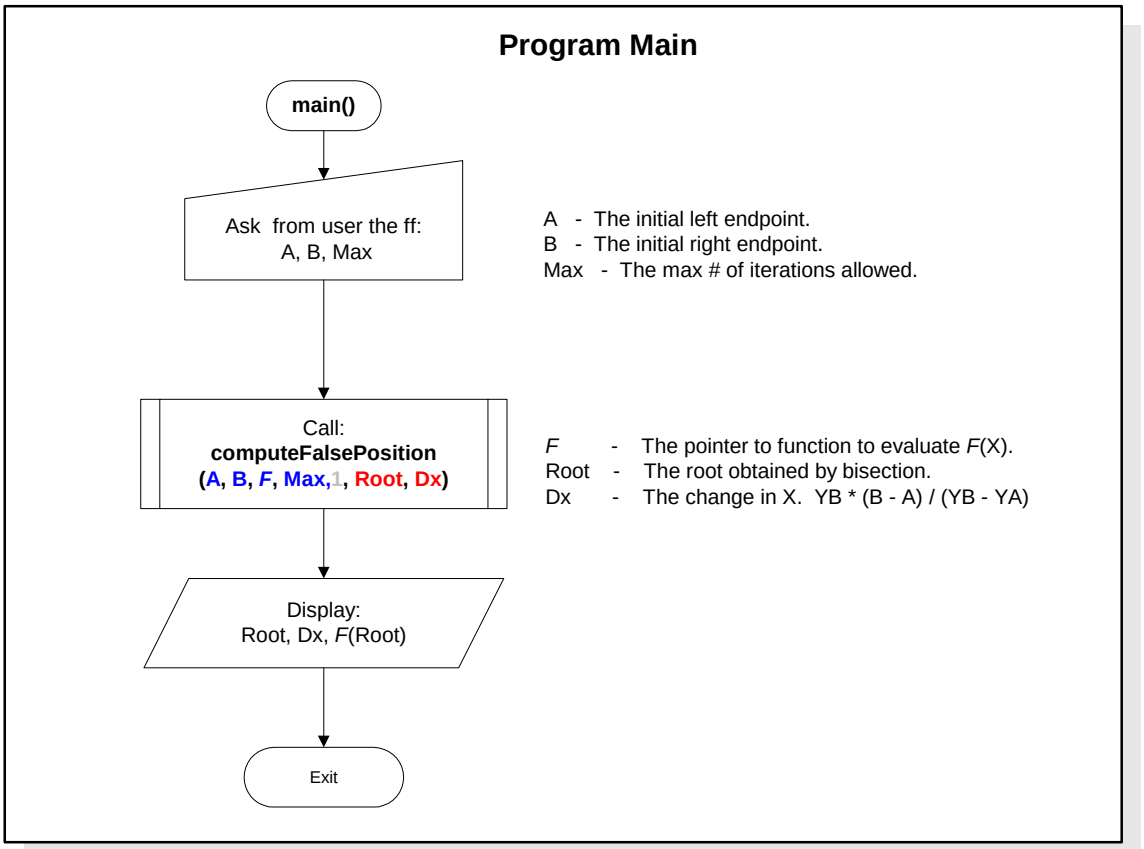
If *f(a)* and *f(c)* have opposite signs, the root *r* lies in [*a*, *c*] so the process must be repeated using [*a*, *c*] as the new interval.

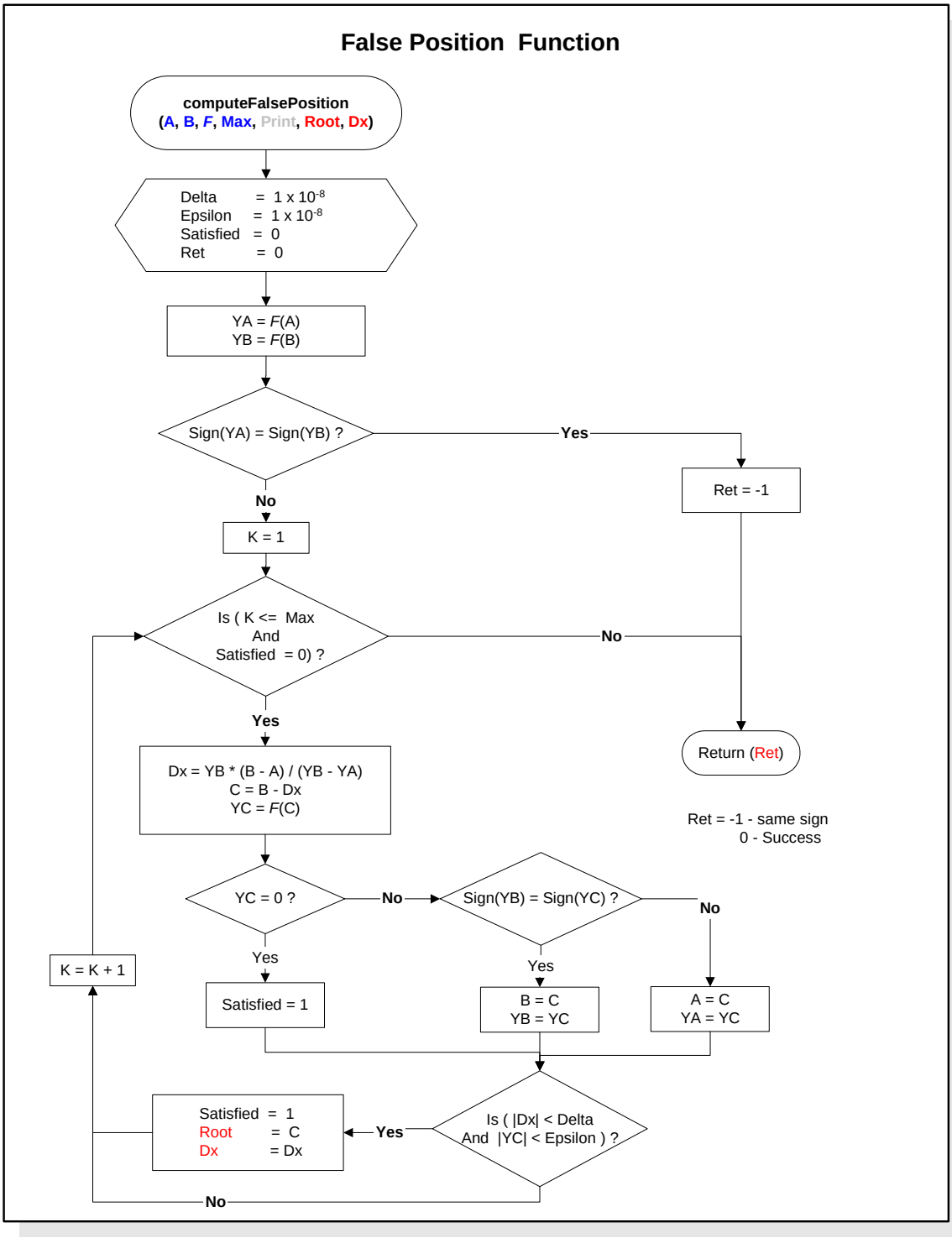
If *f(c)* and *f(b)* have opposite signs, the root r lies in [*c*, *b*] so the process must be repeated using [*c*, *b*] as the new interval.

If *f(c) = 0*, the root is found and so the iteration must now stop.

Algorithm 2.3 (False Position or Regula Falsi Method). To find a root of the equation *f(x) = 0* in the interval [*a*, *b*]. Proceed with the method only if *f(x)* is continuous and *f(a)* and *f(b)* have opposite signs.

Program Flow




False Position Function
computeFalsePosition
(A, B, F, Max, Print, Root, Dx)
Delta = 1 x 10-8
Epsilon = 1 x 10-8
Satisfied = 0
Ret = 0
YA = F(A)
YB = F(B)
Sign(YA) = Sign(YB) ?
Yes
Ret = -1
No
K = 1
Is (K <= Max
And
Satisfied = 0) ?
No
Yes
Return (Ret)
Dx = YB * (B - A) / (YB - YA)
C = B - Dx
YC = F(C)
Ret = -1 - same sign
0 - Success
YC = 0 ?
No
Sign(YB) = Sign(YC) ?
No
Yes
Yes
K = K + 1
Satisfied = 1
B = C
YB = YC
A = C
YA = YC
Satisfied = 1
Root = C
Dx = Dx
Yes
Is (|Dx| < Delta
And |YC| < Epsilon) ?
No

Program Source Code

```
#include <stdio.h>
#include <conio.h>
#include "FalsePositionFunc.h"

double  fFunc(double x);

int main(int argc, char* argv[])
{
    int nStatus = 0;
    double  nA;
    double  nB;
    double  nDx;
    double  nRoot;
    int     nMaxIter = 0;

       /* Ask for input */
    printf("\n     *** False Position or Regula Falsi Method *** ");
    printf("\n\n Using f(x) = x sin (x) - 1  ");
    printf("\n\n Enter the value for point A : ");
    scanf("%lf",&nA);  fflush(stdin);
    printf(" Enter the value for point B : ");
    scanf("%lf",&nB);  fflush(stdin);
    printf(" Enter the maximum number of iteration : ");
    scanf("%d",&nMaxIter);     fflush(stdin);

       /* Compute the the root */
    nStatus = computeFalsePosition(nA,nB,fFunc,nMaxIter,1,&nRoot,&nDx);

    if(nStatus == 0)
    {
        printf("\n The computed root is : %2.8lf",nRoot);
        printf("\n Consecutive iterates differ by : %2.8lf",nDx);
        printf("\n The function value at root is  : %2.8lf",fFunc(nRoot));
    }
    else
    {
        printf("\n Error: f(A) and f(B) have same sign:
[%2.8lf,%2.8lf]",fFunc(nA),fFunc(nB));
    }

    printf("\n\nPress any key to continue....");

    getch();
    return 0;
}


/* The f(x) function */
double  fFunc(double x)
{
    return ((x  * sin(x)) - 1);
}
```

```
#include "FalsePositionFunc.h"
/*
* Function name       : computeFalsePosition
* Description         : Computes the root using the false position method.
* Parameter(s)        :
*     nA            [in]  Point A.
*     nB            [in]  Point B.
*     fFunction     [in]  The function to evaluate f(x).
*     nMaxIter      [in]  The maximum number of iteration.
*     nPrint        [in]  Flag for printing the output on screen,
*     pOutput       [out] The root.
*     pDx           [out] The difference between the iteration.
* Return              :
*    int        0  - Success, -1 - Same sign.
*/
int _cdecl
computeFalsePosition(double nA,double nB,double (*fFunction)(double),int nMaxIter,int
nPrint,double *pOutput,double* pDx)
{
    int nRet = 0;
    static const double nDelta    = 10e-8;         /* Tolerance for change in X */
    static const double nEpsilon  = 10e-8;         /* Tolerance for f(C) */
    double nDx  = 0;                               /* Change in X */
    int    nSatisfied          = 0;                /* If 1, the calculation is satisfied
*/
    int      i;
    double   nYA,nYB,nYC,nC;

    nYA = fFunction(nA);  nYB = fFunction(nB);
    if(GETSIGN(nYA) == GETSIGN(nYB)) nRet = -1; /* Check for the signs */

    if(nRet == 0)
    {
        if(nPrint)
        {
            printf("\n\n N  %-10s \t %-10s  %-10s  %-8s  %-8s  %-8s  "
                   ,"A","B","C","f(A)","f(B)","f(C)");
            printf("\n-----------------------------------------------------------------");
        }
        for(i = 1; i <= nMaxIter && nSatisfied == 0 ; i++)
        {
            /* Get the change in X */
            nDx = nYB * (nB-nA) /(nYB- nYA);
            nC  = nB - nDx;

            nYC = fFunction(nC);
            if(nPrint)
                printf("\n %2d  %2.8lf  %2.8lf  %2.8lf  %2.6lf  %2.6lf  %2.6lf "
                       ,i,nA,nB,nC,nYA,nYB,nYC);
            if(nYC == 0)
            {
                nSatisfied = 1; /* Root is found */
            }
            else if(GETSIGN(nYB) == GETSIGN(nYC))
            {
                nB  = nC;  nYB = nYC;
            }
            else
            {
                nA  = nC;  nYA = nYC;
            }
```

```
            if(fabs(nDx) < nDelta && fabs(nYC) < nEpsilon)
            {
                nSatisfied = 1;
                *pOutput = nC;
                *pDx     = nDx;
            }
        }

        if(nPrint)
             printf("\n----------------------------------------------------------------");
    }

    return nRet;
}
```

Program Output

```
    *** False Position or Regula Falsi Method ***

Using f(x) = x sin (x) - 1

Enter the value for point A : 1
Enter the value for point B : 2
Enter the maximum number of iteration : 20


N  A           B           C           f(A)      f(B)      f(C)
----------------------------------------------------------------
 1  1.00000000  2.00000000  1.16224045  -0.158529  0.818595  0.066583
 2  1.00000000  1.16224045  1.11425351  -0.158529  0.066583  0.000134
 3  1.00000000  1.11425351  1.11415713  -0.158529  0.000134  -0.000000
 4  1.11415713  1.11425351  1.11415714  -0.000000  0.000134  0.000000
 5  1.11415713  1.11415714  1.11415714  -0.000000  0.000000  0.000000
----------------------------------------------------------------
The computed root is : 1.11415714
Consecutive iterates differ by : 0.00000000
The function value at root is  : 0.00000000

Press any key to continue....
```

Newton-Raphson Method

Discussion

A powerful locally convergent method used in solving the root p of the equation is the **Newton-Raphson** method. It converges faster to p than either of the Bisection or False Position methods, provided that *f(x), f '(x) and f ''(x)* are continuous near the root p.

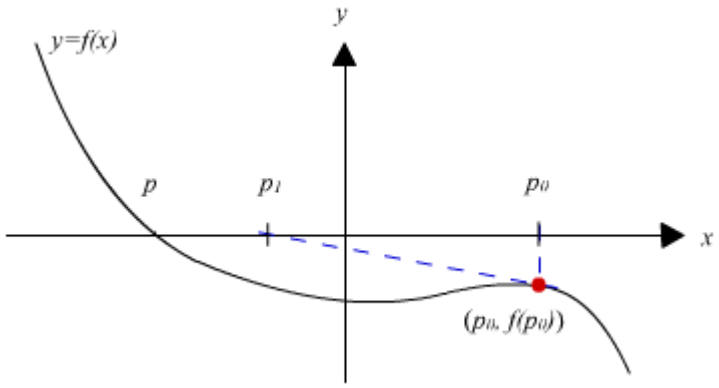


Fig. 2.5.1 Obtaining the improved value graphically.

As shown in Figure 2.5.1, the improved value p_1 is obtained from the line tangent to the point $(p_0, f(p_0))$ which intersects the horizontal axis. Using the slope of the line $(p_1, 0)$, $(p_0, f(p_0))$; which is $m = \dfrac{0 - f(p_0)}{p_1 - p_0}$, equated against the slope of the line tangent to the curve $y = f(x)$ at $(p_0, f(p_0))$; which is $m = f'(p_0)$, the resulting formula would be:

$$p_1 = g(p_0) = p_0 - \frac{f(p_0)}{f'(p_0)} \tag{1}$$

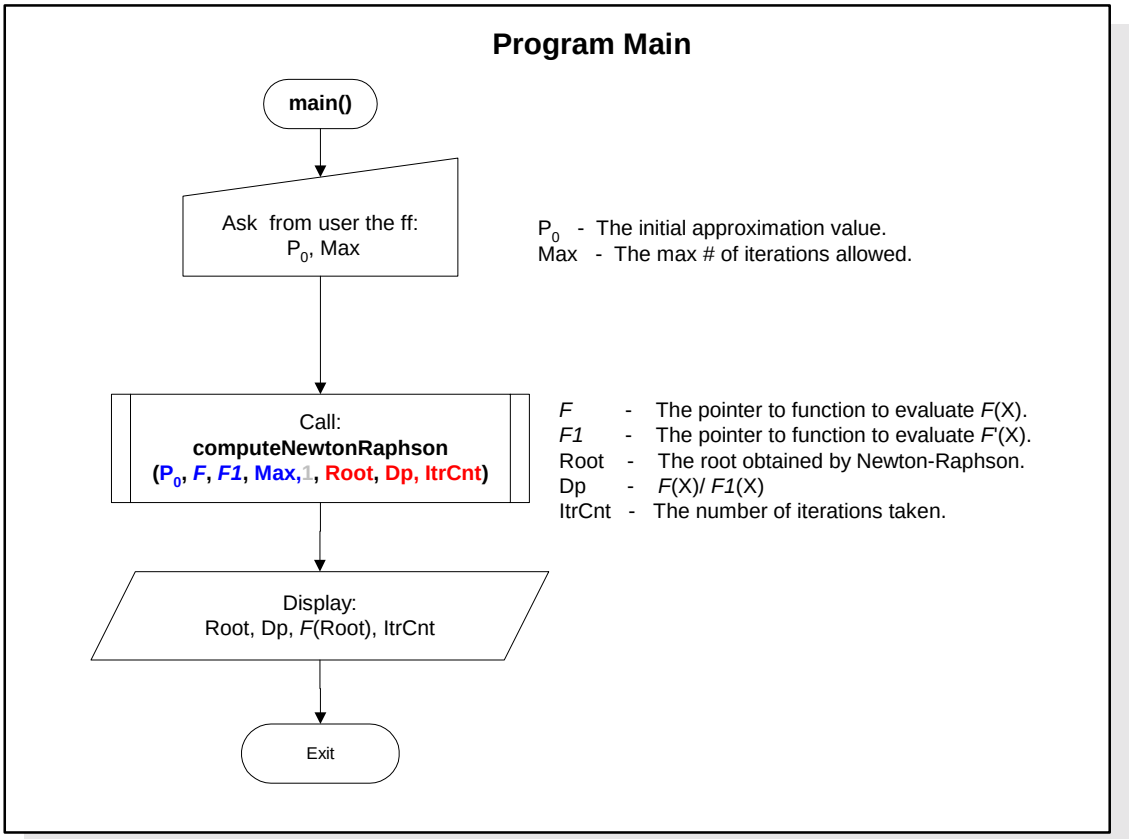
As a locally convergent method, its initial point p_0 must be close enough to the root p to obtain the desired convergence. Its proof, as shown on pp. 72-73 of the textbook, uses the Taylor polynomial analysis of degree n=1. To check if the value p_0 is close enough to p so that the desired convergence can be achieved, Theorem 2.2 from the Fixed-Point Iteration on p. 46 of the textbook is applied by obtaining the *g'(x)* in (1) in which $p_0 \in [p - \delta, p + \delta]$ and that δ would be chosen so that

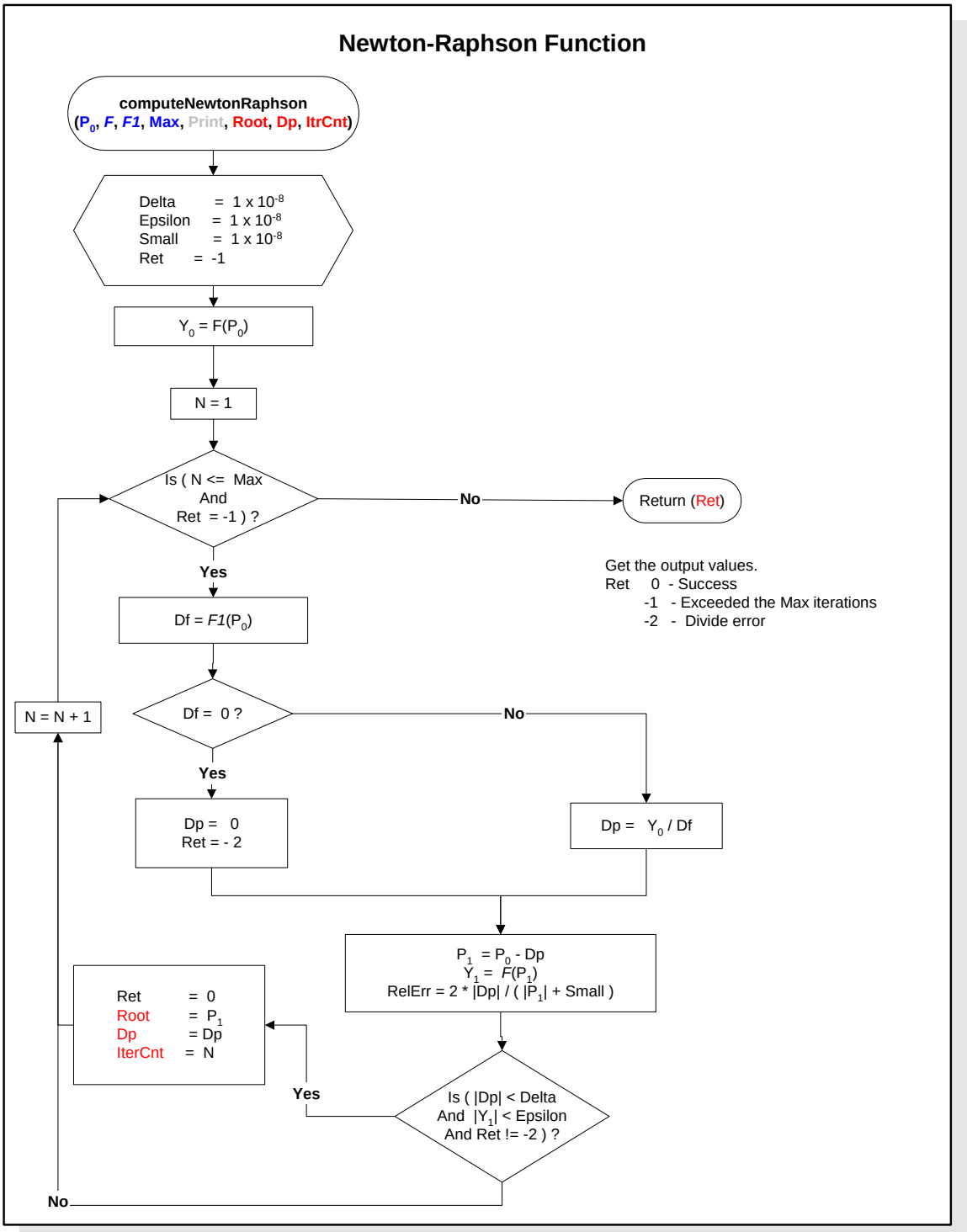
$$g'(x) = \frac{|f(x)\,f''(x)|}{|f'(x)|^2} < 1 \qquad \text{for all } x \in |p - \delta, p + \delta|. \tag{2}$$

Algorithm 2.5 (Newton-Raphson Iteration). To find a root of $f(x) = 0$ given the initial approximation p_0 and using the iteration

$$p_k = p_{k-1} - \frac{f(p_{k-1})}{f'(p_{k-1})} \text{ for k= 1,2 . . . .}$$

Program Flow




Newton-Raphson Function
computeNewtonRaphson
(P0, F, F1, Max, Print, Root, Dp, ItrCnt)
Delta = 1 x 10-8
Epsilon = 1 x 10-8
Small = 1 x 10-8
Ret = -1
Y0 = F(P0)
N = 1
Is (N <= Max And Ret = -1) ?
No
Return (Ret)
Get the output values.
Ret 0 - Success
-1 - Exceeded the Max iterations
-2 - Divide error
Yes
Df = F1(P0)
Df = 0 ?
No
Yes
N = N + 1
Dp = 0
Ret = - 2
Dp = Y0 / Df
P1 = P0 - Dp
Y1 = F(P1)
RelErr = 2 * |Dp| / (|P1| + Small)
Ret = 0
Root = P1
Dp = Dp
IterCnt = N
Yes
Is (|Dp| < Delta And |Y1| < Epsilon And Ret != -2) ?
No

Program Source Code

```
#include <stdio.h>
#include <conio.h>
#include "NewtonRaphsonFunc.h"

double  fFunc(double x);   /* The function of x */
double  fFunc1(double x);  /* The first derivative of a function */

int main(int argc, char* argv[])
{
    int nStatus = 0;
    double  nPn;
    double  nRoot;
    double  nDp;
    int     nMaxIter = 0;
    int     nIterCnt = 0;

       /* Input */
    printf("\n     *** Newton-Raphson Method *** ");
    printf("\n\n Using f(x)  = x^3 - 3x + 2  ");
    printf("\n\n       f'(x) = 2*x^2 - 3 ");
    printf("\n\n Enter the initial value P0 : ");
    scanf("%lf",&nPn);   fflush(stdin);

    printf(" Enter the maximum number of iteration : ");
    scanf("%d",&nMaxIter);     fflush(stdin);

       /* Compute the root */
    nStatus = computeNewtonRaphson(nPn,fFunc,fFunc1,nMaxIter,1,&nRoot,&nDp,&nIterCnt);

    switch(nStatus)
    {
    case 0:
        /* Sucess */
        printf("\n The root was found with the desired tolerance.");
        printf("\n The computed root is           : %2.8lf",nRoot);
        printf("\n Consecutive iterates differ by : %2.8lf",nDp);
        printf("\n The function value at root is  : %2.8lf",fFunc(nRoot));
        printf("\n The number of iteration is     : %d",nIterCnt);
        break;
    case -1:
        printf("\n Failed: The maximum number of iteration was exceeded.");
        break;
    case -2:
        printf("\n Failed: Divide by zero error.");
        break;

    }

    printf("\n\nPress any key to continue....");

    getch();
    return 0;
}
/* The f(x) function */
double  fFunc(double x)
{
    return (pow(x,3) - 3*x + 2);
}
/* The first derivative of a function */
double  fFunc1(double x)
{
    return (3*pow(x,2) - 3);
}
```

```
#include "NewtonRaphsonFunc.h"

/* Function name      : computeNewtonRaphson
* Description        : Computes a root of an equation using the newton raphson
algorithm.
* Parameter(s)       :
*     nPn              [in]  The initial value P0.
*     fFunction        [in]  The function to evaluate f(Pn).
*     fFunction1       [in]  The function to evaluate f'(Pn).
*     nMaxIter         [in]  The allowable maximum number of iteration.
*     nPrint           [in]  Flag for printing the output on screen,
*     pOutput          [out]  The root.
*     pDp              [out]  The difference on iteration.
*     pIterCnt         [out]  The actual number of iteration performed.
* Return             :
*    int     0 -  Success, -1 -  Iteration runout , -2 -  Divide error.
*/
int _cdecl
computeNewtonRaphson(double nPn,double (*fFunction)(double),double
(*fFunction1)(double),int nMaxIter,int nPrint,double *pOutput,double *pDp,int* pIterCnt)
{
    int nRet = -1;                    /* Initialize as iteration runout */
    static const double nDelta    = 10e-8;  /* Tolerances*/
    static const double nEpsilon  = 10e-8;
    static const double nSmall    = 10e-8;
    double nDf;     /* derivative of a function */
    double nDp;     /* f(Pn)/f'(Pn) */
    double nPn1;    /* Pn + 1  */
    double nYn;     /* f(Pn)  */
    double nYn1;    /* f(Pn+1)  */
    int    i;
    double nRelErr;

    nYn = fFunction(nPn);    /* Compute the function value */
    if(nPrint)
    {
        printf("\n\n N    %-10s  %-10s  %-10s  ","Pn","Pn+1","Ek+1");
        printf("\n-------------------------------------------------------------------------");
    }
    for(i = 1; i <= nMaxIter && nRet == -1 ; i++)
    {
        nDf = fFunction1(nPn); /* Get the derivative of Pn */
        if(nDf == 0)
        {
            /* Divide error */
            nRet = -2;
            nDp  = 0;
        }
        else
            nDp = nYn/nDf;

        nPn1  = nPn - nDp;    nYn1  = fFunction(nPn1);

        if(nPrint)
            printf("\n %2d  %2.8lf  %2.8lf  %2.8lf  ",i-1,nPn,nPn1,nPn1 - nPn);

        nRelErr = 2 * fabs(nDp)/(fabs(nPn1) + nSmall);

        if(nRelErr < nDelta && fabs(nYn1) < nEpsilon)
        {
```

```
            /* Check for convergence */
            if(nRet != -2)
            {
                nRet = 0;
                *pOutput = nPn1;
                *pDp     = nDp;
                *pIterCnt = i;
            }
        }

        nPn = nPn1;  nYn = nYn1;
    }
    if(nPrint)
        printf("\n----------------------------------------------------------------------");
    return nRet;
}
```

Program Output

```
    *** Newton-Raphson Method ***

Using f(x)  = x^3 - 3x + 2

      f'(x) = 2*x^2 - 3

Enter the initial value P0 : -3
Enter the maximum number of iteration : 10


N    Pn           Pn+1         Ek+1
----------------------------------------------------------------------
 0  -3.00000000  -2.33333333  0.66666667
 1  -2.33333333  -2.05555556  0.27777778
 2  -2.05555556  -2.00194932  0.05360624
 3  -2.00194932  -2.00000253  0.00194679
 4  -2.00000253  -2.00000000  0.00000253
 5  -2.00000000  -2.00000000  0.00000000
----------------------------------------------------------------------
The root was found with the desired tolerance.
The computed root is           : -2.00000000
Consecutive iterates differ by : -0.00000000
The function value at root is  : 0.00000000
The number of iteration is     : 6

Press any key to continue....
```

Secant Method

Discussion

As previously discussed, the Newton-Raphson is one of the powerful methods in solving the root for the equation $f(x) = 0$. One can easily evaluate $f'(x)$ when $f(x)$ is reasonably simple. However, when the expression is too complicated, obtaining the $f'(x)$ from $f(x)$ is no longer an easy task. Thus, it is useful to have another method which does not require evaluation of $f'(x)$. Such a method is the **Secant** method.

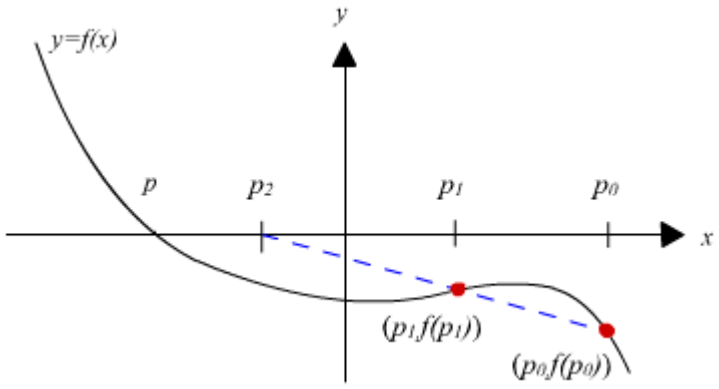


Fig. 2.6.1 Obtaining the improved value graphically.

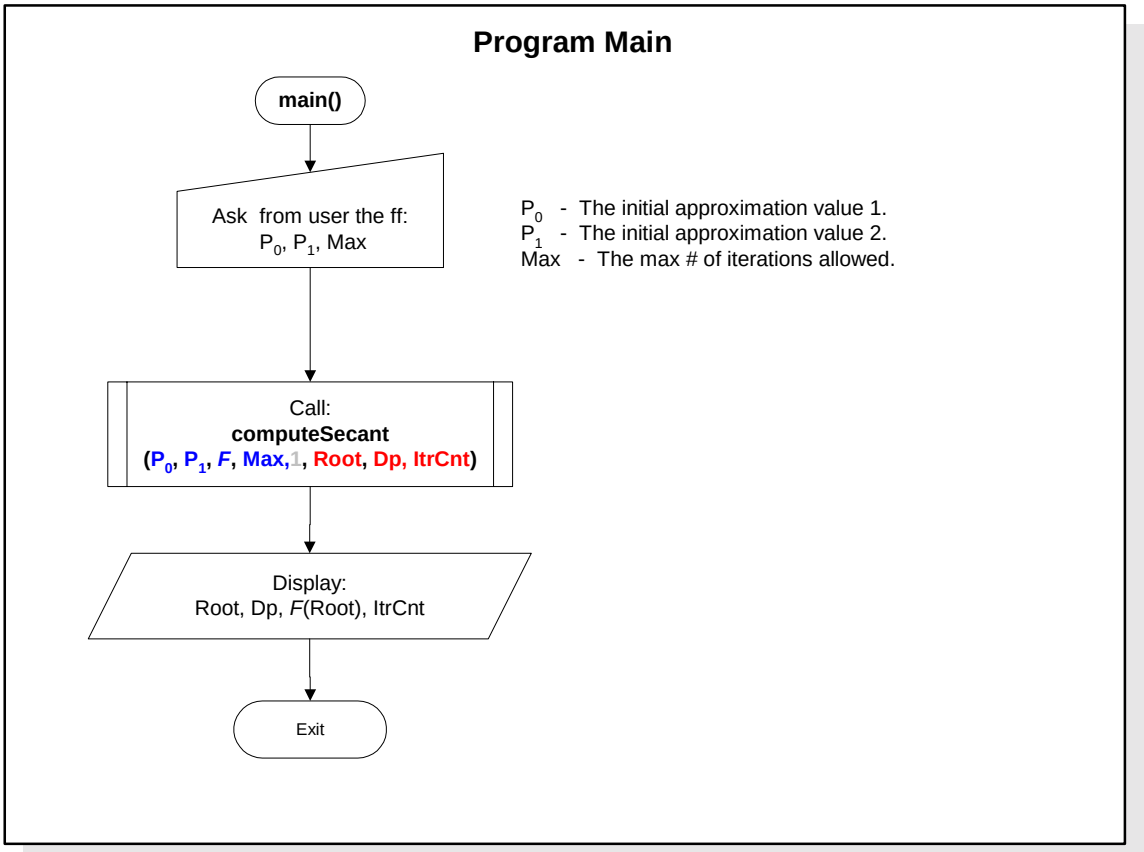
As shown in Figure 2.6.1, it requires 2 initial points $(p_0, f(p_0))$ and $(p_1, f(p_1))$ near the point $(p, 0)$. The improved value p_2 is obtained from the line secant to the points $(p_0, f(p_0))$ and $(p_1, f(p_1))$ which intersects the horizontal axis. Using the slope of the line $(p_2, 0)$, $(p_1, f(p_1))$; which is $m = \dfrac{0 - f(p_1)}{p_2 - p_1}$, equated against the slope of the line secant to the curve $y = f(x)$ at $(p_0, f(p_0))$ and $(p_1, f(p_1))$; which is $m = \dfrac{f(p_1) - f(p_0)}{p_1 - p_0}$, the resulting formula would be:

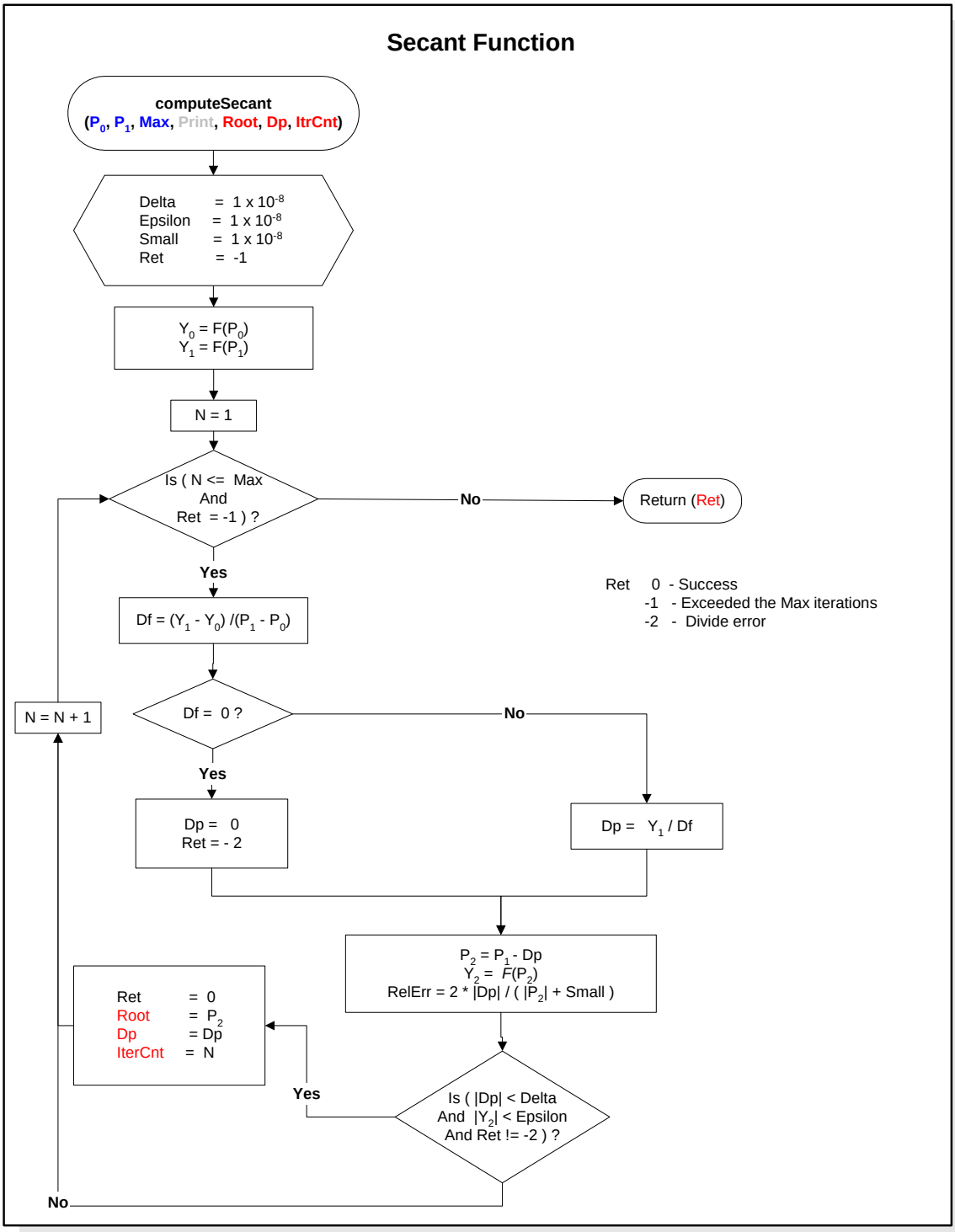
$$p_2 = g(p_1, p_0) = p_1 - \frac{f(p_1)(p_1 - p_0)}{f(p_1) - f(p_0)} \qquad (1)$$

Algorithm 2.6 (Secant Method). To find a root of $f(x) = 0$ given the initial approximations p_0 and p_1 and using the iteration

$$p_{k+1} = p_k - \frac{f(p_k)(p_k - p_{k-1})}{f(p_k) - f(p_{k-1})} \text{ for k= 1,2 . . . .}$$

Program Flow




Secant Function
computeSecant
(P0, P1, Max, Print, Root, Dp, ItrCnt)
Delta = 1 x 10-8
Epsilon = 1 x 10-8
Small = 1 x 10-8
Ret = -1
Y0 = F(P0)
Y1 = F(P1)
N = 1
Is (N <= Max
And
Ret = -1) ?
No
Return (Ret)
Yes
Ret 0 - Success
-1 - Exceeded the Max iterations
-2 - Divide error
Df = (Y1 - Y0) /(P1 - P0)
Df = 0 ?
No
Yes
N = N + 1
Dp = 0
Ret = - 2
Dp = Y1 / Df
P2 = P1 - Dp
Y2 = F(P2)
RelErr = 2 * |Dp| / (|P2| + Small)
Ret = 0
Root = P2
Dp = Dp
IterCnt = N
Yes
Is (|Dp| < Delta
And |Y2| < Epsilon
And Ret != -2) ?
No

Program Source Code

```
#include <stdio.h>
#include <conio.h>
#include "SecantFunc.h"

double  fFunc(double x);   /* The function of x */

int main(int argc, char* argv[])
{
    int nStatus = 0;
    double  nPn;
    double  nPn1;
    double  nRoot;
    double  nDp;
    int     nMaxIter = 0;
    int     nIterCnt = 0;

    /* Input */
    printf("\n     *** Secant Method *** ");
    printf("\n\n Using f(x)  = x^3 - 3x + 2  ");
    printf("\n\n Enter the initial value P0 : ");
    scanf("%lf",&nPn);   fflush(stdin);
    printf(" Enter the initial value P1 : ");
    scanf("%lf",&nPn1);   fflush(stdin);

    printf(" Enter the maximum number of iteration : ");
    scanf("%d",&nMaxIter);    fflush(stdin);

       /* Compute the root */
    nStatus = computeSecant(nPn,nPn1,fFunc,nMaxIter,1,&nRoot,&nDp,&nIterCnt);

    switch(nStatus)
    {
    case 0:
        /* Sucess */
        printf("\n The root was found with the desired tolerance.");
        printf("\n The computed root is          : %2.8lf",nRoot);
        printf("\n Consecutive iterates differ by : %2.8lf",nDp);
        printf("\n The function value at root is  : %2.8lf",fFunc(nRoot));
        break;
    case -1:
        printf("\n Failed: The maximum number of iteration was exceeded.");
        break;
    case -2:
        printf("\n Failed: Divide by zero error.");
        break;
    }

    printf("\n\nPress any key to continue....");

    getch();
    return 0;
}

/* The f(x) function */
double  fFunc(double x)
{
    return (pow(x,3) - 3*x + 2);
}
```

```
#include "SecantFunc.h"
/* Function name      : computeSecant
* Description        : Computes a root of an equation using the secant method.
* Parameter(s)       :
*     nPn             [in]  The initial value P0.
*     nPn1            [in]  The initial value P1.
*     *fFunction      [in]  The function to evaluate f(Pn).
*     nMaxIter        [in]  The maximum number of iteration.
*     nPrint          [in]  Flag for printing the output on screen,
*     pOutput         [out]  The root.
*     pDp             [out]  The difference on iteration.
*     pIterCnt        [out]  The actual number of iteration performed.
* Return             :
*    int     0 -  Success, -1 -  Iteration has runout, -2 -  Divide error.
*/
int _cdecl
computeSecant(double nPn,double nPn1,double (*fFunction)(double),int nMaxIter,int
nPrint,double *pOutput,double *pDp,int* pIterCnt)
{
    int nRet = -1;                                  /* Initialize as iteration runout */
    static const double nDelta    = 10e-8;          /* Tolerances*/
    static const double nEpsilon  = 10e-8;
    static const double nSmall    = 10e-8;
    double nDf;                                     /* Slope    */
    double nDp;                                     /* Y1/slope */
    double nPn2;                                    /* Pn+2   */
    double nYn;                                     /* f(Pn)  */
    double nYn1;                                    /* f(Pn+1)  */
    double nYn2;                                    /* f(Pn+2)  */
    int    i;
    double nRelErr;
    nYn  = fFunction(nPn);                           /* Compute the function value */
    nYn1 = fFunction(nPn1);                          /* Compute the function value */
    if(nPrint)
    {
        printf("\n\n    N    %-10s   %-10s   %-10s   %-10s   %-10s  "
               ,"Pn","Pn+1","Pn+2","Ek+1","Ek+2");
        printf("\n-------------------------------------------------------------------------");
    }
    for(i = 1; i <= nMaxIter && nRet == -1 ; i++)
    {
        /* Compute the slope */
        nDf = (nYn1 - nYn)/(nPn1 - nPn);
        if(nDf == 0)
        {
            nRet = -2;   nDp  = 0; /* Divide error */
        }
        else
        {
            nDp = nYn1/nDf;
        }

        nPn2  = nPn1 - nDp;   nYn2  = fFunction(nPn2);
        if(nPrint)
            printf("\n   %2d  %2.8lf  %2.8lf  %2.8lf  %2.8lf  %2.8lf  "
                   ,i-1,nPn,nPn1,nPn2,nPn1 - nPn,nPn2 - nPn1);
         nRelErr = 2 * fabs(nDp)/(fabs(nPn2) + nSmall);
        if(nRelErr < nDelta && fabs(nYn2) < nEpsilon)
        {
            if(nRet != -2)
            {
```

```
                /* Convergence has been found on the given tolerance */
                nRet = 0;
                *pOutput = nPn2;  *pDp     = nDp;
            }
        }
        nPn  = nPn1;  nPn1 = nPn2;   nYn  = nYn1; nYn1 = nYn2;
    }
    if(nPrint)
        printf("\n----------------------------------------------------------------------");
    return nRet;
}
```

Program Output

```
     *** Secant Method ***

 Using f(x)  = x^3 - 3x + 2

 Enter the initial value P0 : -3
 Enter the initial value P1 : -2.5
 Enter the maximum number of iteration : 20


    N    Pn           Pn+1         Pn+2         Ek+1         Ek+2
----------------------------------------------------------------------
    0  -3.00000000  -2.50000000  -2.18987342  0.50000000  0.31012658
    1  -2.50000000  -2.18987342  -2.04697513  0.31012658  0.14289828
    2  -2.18987342  -2.04697513  -2.00531401  0.14289828  0.04166112
    3  -2.04697513  -2.00531401  -2.00016217  0.04166112  0.00515184
    4  -2.00531401  -2.00016217  -2.00000057  0.00515184  0.00016160
    5  -2.00016217  -2.00000057  -2.00000000  0.00016160  0.00000057
    6  -2.00000057  -2.00000000  -2.00000000  0.00000057  0.00000000
----------------------------------------------------------------------
 The root was found with the desired tolerance.
 The computed root is           : -2.00000000
 Consecutive iterates differ by : -0.00000000
 The function value at root is  : 0.00000000

Press any key to continue....
```

Seidel Iteration

Discussion

Seidel Iteration is an enhanced version of the Fixed-Point Iteration for nonlinear systems. Considering the given functions:

$$\begin{aligned} f_1(x,y) &= 0, \\ f_2(x,y) &= 0 \end{aligned} \tag{1}$$

one must transform (1) into a new form (same principle as the nonlinear equation)

$$\begin{aligned} x &= g_1(x,y), \\ y &= g_2(x,y) \end{aligned} \tag{2}$$

For example:

$$\begin{aligned} f_1(x,y) &= x^2 - 2x - y + 0.5, \\ f_2(x,y) &= x^2 + 4y^2 - 4 \end{aligned} \tag{3}$$

the *g(x)* can be transformed into its iterative system of equation:

$$\begin{aligned} g_1(x,y) &= \frac{x^2 - y + 0.5}{2}, \\ g_2(x,y) &= \frac{-x^2 - 4y^2 + 8y + 4}{8} \end{aligned} \tag{4}$$

Once the *g(x)* is obtained from (1), one may now apply the Seidel iteration using a starting value $p_0, q_0,$ as what is done in the nonlinear equations.

n	$p_{n,Qn}$	$g(x,y)$
1	p_1	$g_1(p_0, q_0)$
	q_1	$g_2(p_1, q_0)$
2	p_2	$g_1(p_1, q_1)$
	q_2	$g_2(p_2, q_1)$
.	.	.
	.	.
.	.	.
	.	.
∞	p	$g_1(p, q)$
	q	$g_2(p, q)$

Table 2.9.1 The sequence pattern for the Nonlinear Seidel Iteration.

As shown in the Table 2.9.1, Seidel iteration is just the same as the Fixed-Point iteration for nonlinear systems except that the value of p_{n+1}, instead of p_n, is used in solving for q_{n+1} which improves the performance of the iteration.

In order to insure that the solution converge as $n \to \infty$, the functions $g_1(x,y)$ *and* $g_2(x,y)$ must be tested based on Theorem 2.9 on pp. 106-107 of the textbook which is the starting values (p_0, q_0) must be sufficiently close to (p,q) and that

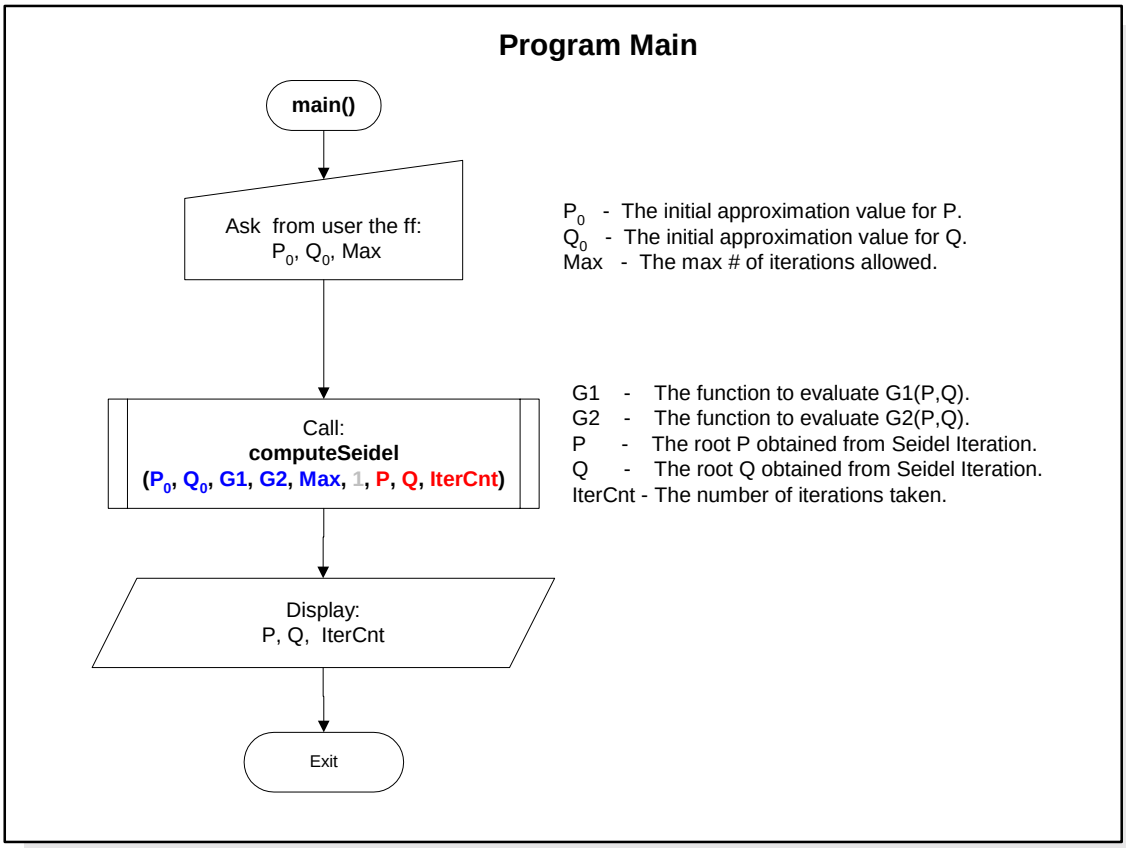
$$\begin{aligned} \left|\frac{\partial g_1}{\partial x}(p,q)\right| + \left|\frac{\partial g_1}{\partial y}(p,q)\right| < 1 \\ \left|\frac{\partial g_2}{\partial x}(p,q)\right| + \left|\frac{\partial g_2}{\partial y}(p,q)\right| < 1 \end{aligned} \tag{5}$$

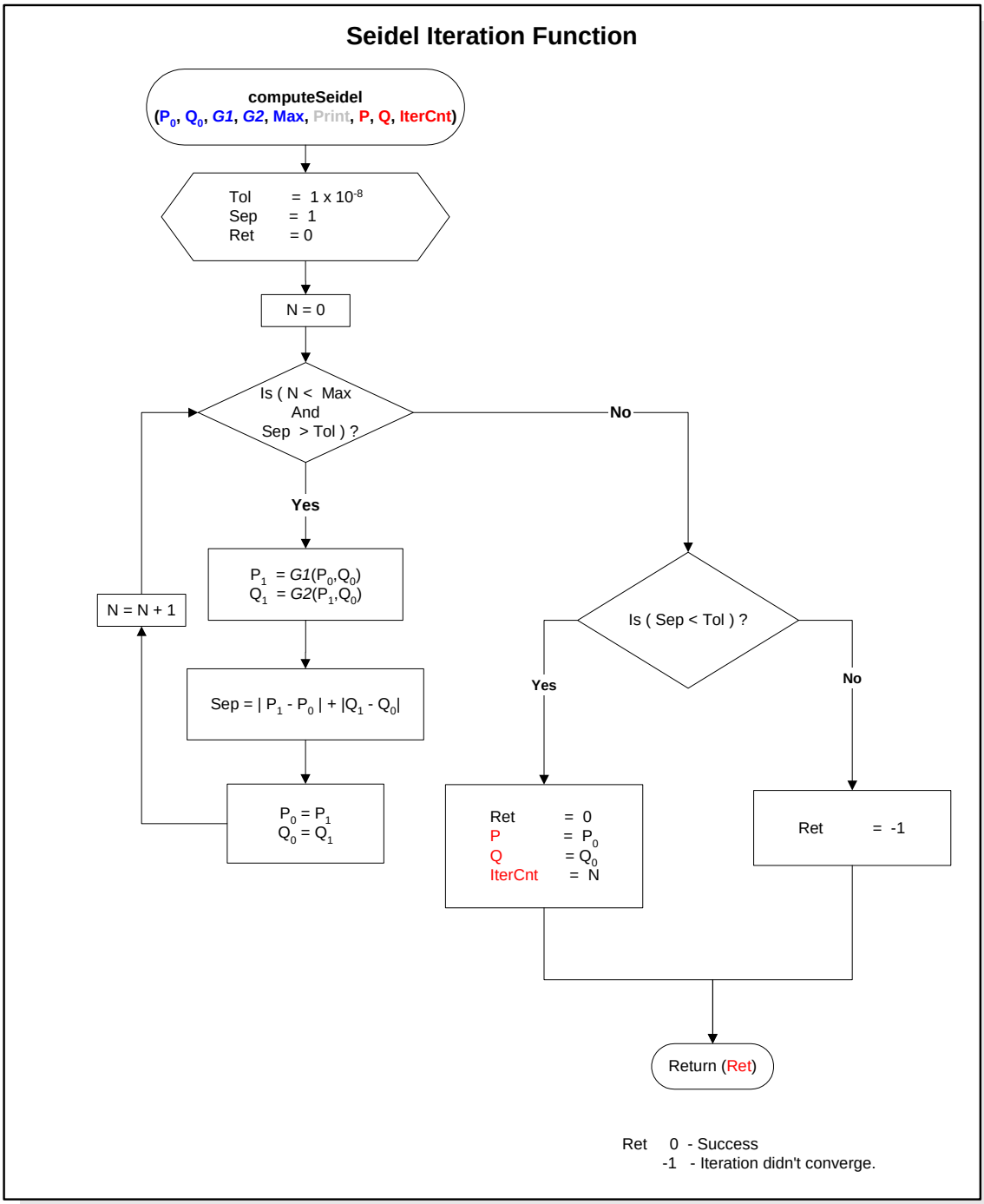
Algorithm 2.9 (Nonlinear Seidel Iteration). To solve the nonlinear fixed point system

$$x = g_1(x,y)$$
$$y = g_2(x,y)$$

given one initial approximation $P_0 = (p_0,q_0)$, and generating the sequence $\{P_K\} = \{(p_K, q_K)\}$ that converges to the solution $P = (p, q)$ [i.e., $p = g_1(p, q)$ and $q=g_2(p, q)$].

Program Flow




Seidel Iteration Function
computeSeidel
(P0, Q0, G1, G2, Max, Print, P, Q, IterCnt)
Tol = 1 x 10-8
Sep = 1
Ret = 0
N = 0
Is (N < Max And Sep > Tol) ?
No
Yes
P1 = G1(P0,Q0)
Q1 = G2(P1,Q0)
N = N + 1
Sep = | P1 - P0 | + |Q1 - Q0|
P0 = P1
Q0 = Q1
Is (Sep < Tol) ?
Yes
No
Ret = 0
P = P0
Q = Q0
IterCnt = N
Ret = -1
Return (Ret)
Ret 0 - Success
-1 - Iteration didn't converge.

Program Source Code

```
#include <stdio.h>
#include <conio.h>
#include "SeidelFunc.h"

double  gFunc1(double x,double y);   /* The g1 of x,y*/
double  gFunc2(double x,double y);   /* The g2 of x,y*/

int main(int argc, char* argv[])
{
    int nStatus = 0;
    double  nPn;
    double  nQn;
    double  nX;
    double  nY;
    int     nMaxIter = 0;
    int     nIterCnt = 0;

       /* Input */
    printf("\n   *** Nonlinear Seidel Iteration *** ");
    printf("\n\n Using f1(x,y)  = y - x^2 + 3 ");
    printf("\n\n Using f2(x,y)  = 3 - xy ");

    printf("\n\n Enter the initial value P0 : ");
    scanf("%lf",&nPn);  fflush(stdin);

    printf(" Enter the initial value Q0 : ");
    scanf("%lf",&nQn);  fflush(stdin);

    printf(" Enter the maximum number of iteration : ");
    scanf("%d",&nMaxIter);  fflush(stdin);

       /* Compute the roots */
    nStatus = computeSeidel(nPn,nQn,gFunc1,gFunc2,nMaxIter,1,&nX,&nY,&nIterCnt);

    switch(nStatus)
    {
    case 0:
        /* Sucess */
        printf("\n Solution was found after %d iterations. ",nIterCnt);
        printf("\n x  : %2.8lf",nX);
        printf("\n y  : %2.8lf",nY);
        break;
    case -1:
        printf("\n Failed: Seidel iteration did not converge.");
        break;
    default:
        break;
    }

    printf("\n\nPress any key to continue....");

    getch();
    return 0;
}
double  gFunc1(double x,double y)   /* The g1 of x,y */
{
    return ((4*x - pow(x,2) + y + 3)/4);
}

double  gFunc2(double x,double y)   /* The g2 of x,y */
{
    return ((3 - x*y + 2 * y)/2);
}
```

```
#include "SeidelFunc.h"

/* Function name       : computeSeidel
* Description         : Computes the roots of the system using seidel iteration.
* Parameter(s)        :
*     nPn         [in]  Initial value P0.
*     nQn         [in]  Initial value Q0.
*     gFunc1      [in]  g1(x,y)
*     gFunc2      [in]  g2(x,y)
*     nMaxIter    [in]  The maximum number of iteration.
*    nPrint       [in]  Flag for printing the output on screen,
*     *pX         [out]  The X root.
*     *pY         [out]  The Y root.
*     pIterCnt    [out]  The number of iterations taken.
* Return            :
*    int   0 - Success, -1 - The iteration didn't converge.
*/
int _cdecl
computeSeidel(double nPn,double nQn,FuncXY gFunc1,FuncXY gFunc2,int nMaxIter,int
nPrint,double *pX,double *pY,int* pIterCnt)
{
    int nRet = 0;

    static const double nTol      = 10e-8;        /* Tolerance*/
    double nSep = 1;                              /* Total relative errors */
    double nPn1;                                  /* Pn+1  */
    double nQn1;                                  /* Qn+1  */
    int    i;
    if(nPrint)
    {
        printf("\n\n    N    %-10s   %-10s  ","Pn","Qn");
        printf("\n-------------------------------------------------------------------------");
        printf("\n   %2d  %2.8lf  %2.8lf  ",0,nPn,nQn);
    }
    i = 0;
    while(i < nMaxIter && nSep > nTol)
    {
        i++;
        nPn1 = gFunc1(nPn,nQn); nQn1 = gFunc2(nPn1,nQn);
        if(nPrint)
            printf("\n   %2d  %2.8lf  %2.8lf  ",i,nPn1,nQn1);

        nSep = fabs(nPn1 - nPn) + fabs(nQn1 - nQn) ;

        nPn = nPn1;    nQn = nQn1;
    }
    if(nSep < nTol)
    {
        nRet = 0;

        *pX = nPn;
        *pY = nQn;
        *pIterCnt = i;
    }
    else
    {
        nRet = -1;
    }
    if(nPrint)
        printf("\n-------------------------------------------------------------------");

    return nRet;
}
```

Program Output

```
   *** Nonlinear Seidel Iteration ***

 Using f1(x,y)  = y - x^2 + 3

 Using f2(x,y)  = 3 - xy

 Enter the initial value P0 : 1
 Enter the initial value Q0 : 1
 Enter the maximum number of iteration : 20


    N     Pn           Qn
-------------------------------------------------------------------------
    0  1.00000000  1.00000000
    1  1.75000000  1.62500000
    2  2.14062500  1.38574219
    3  2.09149170  1.43660805
    4  2.10705933  1.42309885
    5  2.10290929  1.42677495
    6  2.10404616  1.42577477
    7  2.10373729  1.42604699
    8  2.10382139  1.42597291
    9  2.10379851  1.42599307
   10  2.10380474  1.42598758
   11  2.10380304  1.42598908
   12  2.10380350  1.42598867
   13  2.10380338  1.42598878
   14  2.10380341  1.42598875
-------------------------------------------------------------------------
 Solution was found after 14 iterations.
 x   : 2.10380341
 y   : 1.42598875

Press any key to continue....
```

Newton's Method for Systems

Discussion

The principle of the Newton-Raphson iteration for nonlinear equation can also be applied to nonlinear systems in obtaining the roots. In the nonlinear equation,

$$f\ (x) = 0 \tag{1}$$

is transformed into a new form

$$p_1 = p_0 - \frac{f(p_0)}{f'(p_0)} \tag{2}$$

in order to obtain the improved value p_{k+1} from p_k.. Applying the principle to nonlinear systems, the system

$$F\ (X) = 0 \tag{3}$$

will be transformed into a new form similar to (2)

$$P_1 = P_0 + \Delta P = P_0 - J(P_0)^{-1} F(P_0). \tag{4}$$

where X = (w, x, y, z), F(X) = $f_1(w, x, y, z)$, $f_2(w, x, y, z)$, $f_3(w, x, y, z)$, $f_4(w, x, y, z)$, P = (p, q, r, s) and $J(P_0)^{-1}$ which is the equivalent to $\frac{1}{f'(p_0)}$ in the nonlinear equation, is the inverse of the Jacobian matrix.

$$J(P_0) = \begin{bmatrix} \frac{\partial}{\partial w} f_1(p_0,q_0,r_0,s_0) & \frac{\partial}{\partial x} f_1(p_0,q_0,r_0,s_0) & \frac{\partial}{\partial y} f_1(p_0,q_0,r_0,s_0) & \frac{\partial}{\partial z} f_1(p_0,q_0,r_0,s_0) \\ \frac{\partial}{\partial w} f_2(p_0,q_0,r_0,s_0) & \frac{\partial}{\partial x} f_2(p_0,q_0,r_0,s_0) & \frac{\partial}{\partial y} f_2(p_0,q_0,r_0,s_0) & \frac{\partial}{\partial z} f_2(p_0,q_0,r_0,s_0) \\ \frac{\partial}{\partial w} f_3(p_0,q_0,r_0,s_0) & \frac{\partial}{\partial x} f_3(p_0,q_0,r_0,s_0) & \frac{\partial}{\partial y} f_3(p_0,q_0,r_0,s_0) & \frac{\partial}{\partial z} f_3(p_0,q_0,r_0,s_0) \\ \frac{\partial}{\partial w} f_4(p_0,q_0,r_0,s_0) & \frac{\partial}{\partial x} f_4(p_0,q_0,r_0,s_0) & \frac{\partial}{\partial y} f_4(p_0,q_0,r_0,s_0) & \frac{\partial}{\partial z} f_4(p_0,q_0,r_0,s_0) \end{bmatrix} \tag{5}$$

For the convenience of solving (4), one may use the solutions to linear systems provided that the value obtained from the Jacobian matrix is nonsingular. The derivation

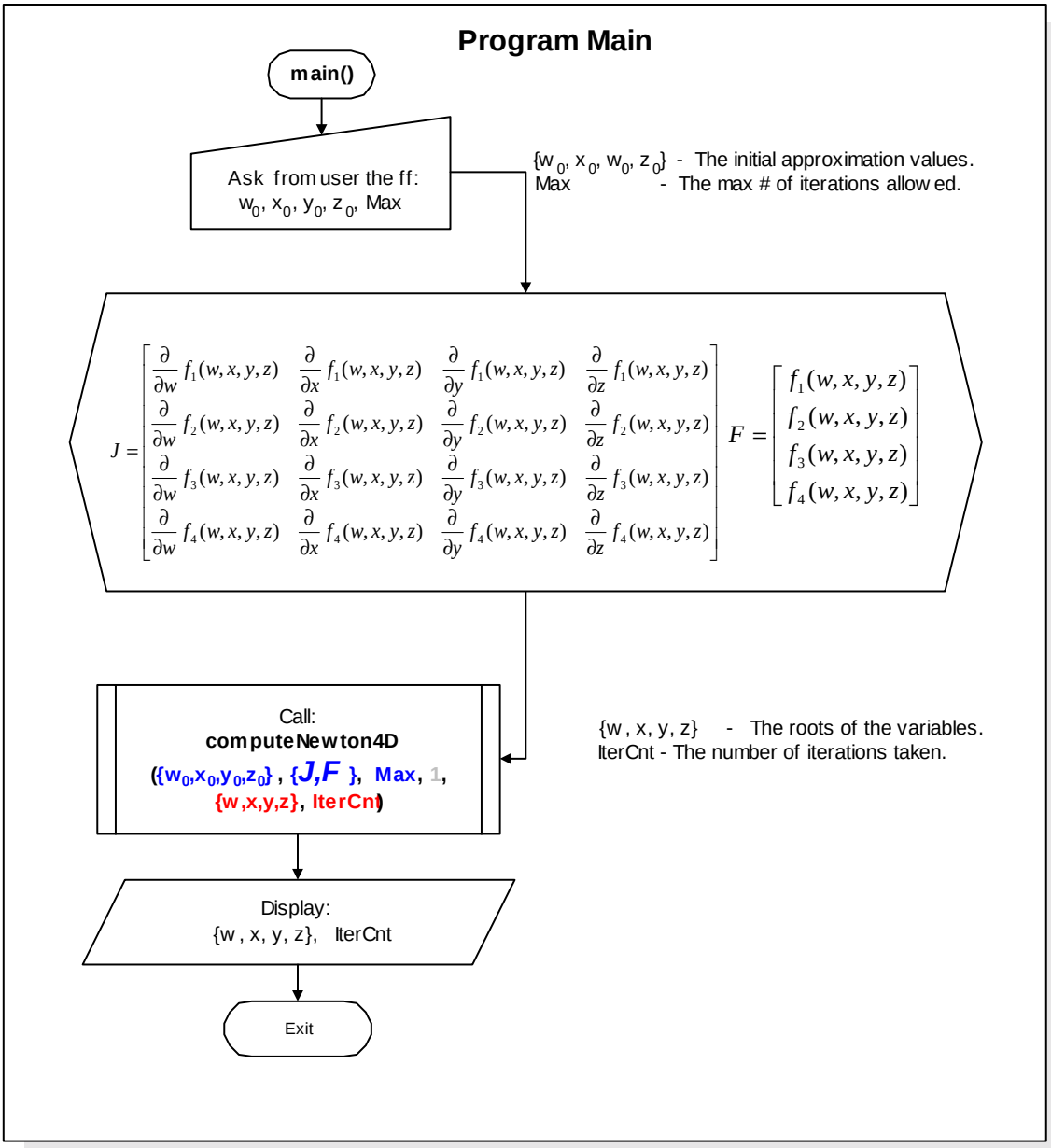
of (4) is illustrated in **Newton's Method in Two Dimensions** on pp. 113-114 of the textbook.

Algorithm 2.10 (Newton-Raphson Method in Four Dimensions). To solve the nonlinear system

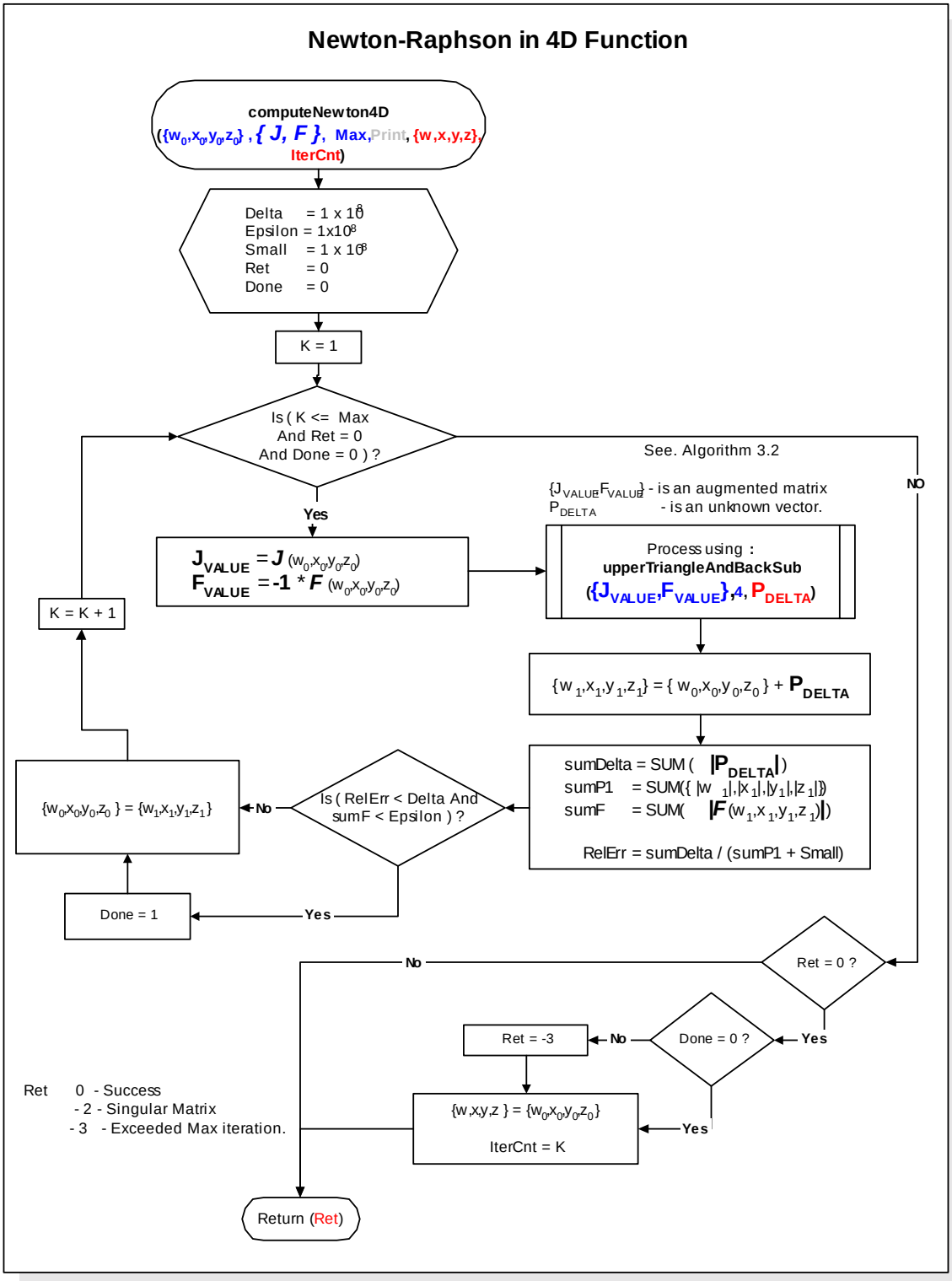
$$0 = f_1(w, x, y, z)$$
$$0 = f_2(w, x, y, z)$$
$$0 = f_3(w, x, y, z)$$
$$0 = f_4(w, x, y, z)$$

given one initial approximation (p_0, q_0, r_0, s_0), and using Newton-Raphson iteration.

Program Flow



Newton-Raphson in 4D Function



Program Source Code

```
#include <stdio.h>
#include <conio.h>
#include "Newton4dFunc.h"

/* The fn(x0,x1,x2,x3) */
double  fFunc0(double x0,double x1,double x2,double x3);
double  fFunc1(double x0,double x1,double x2,double x3);
double  fFunc2(double x0,double x1,double x2,double x3);
double  fFunc3(double x0,double x1,double x2,double x3);
/* The fn/dxn(x0,x1,x2,x3)*/
double  f0FuncDx0(double x0,double x1,double x2,double x3);
double  f0FuncDx1(double x0,double x1,double x2,double x3);
double  f0FuncDx2(double x0,double x1,double x2,double x3);
double  f0FuncDx3(double x0,double x1,double x2,double x3);
double  f1FuncDx0(double x0,double x1,double x2,double x3);
double  f1FuncDx1(double x0,double x1,double x2,double x3);
double  f1FuncDx2(double x0,double x1,double x2,double x3);
double  f1FuncDx3(double x0,double x1,double x2,double x3);
double  f2FuncDx0(double x0,double x1,double x2,double x3);
double  f2FuncDx1(double x0,double x1,double x2,double x3);
double  f2FuncDx2(double x0,double x1,double x2,double x3);
double  f2FuncDx3(double x0,double x1,double x2,double x3);
double  f3FuncDx0(double x0,double x1,double x2,double x3);
double  f3FuncDx1(double x0,double x1,double x2,double x3);
double  f3FuncDx2(double x0,double x1,double x2,double x3);
double  f3FuncDx3(double x0,double x1,double x2,double x3);


int main(int argc, char* argv[])
{
    int nStatus = 0;
    double  nPn[4];
    double  nPnOut[4];
    int     i;
    int     nMaxIter = 0;
    int     nIterCnt = 0;
    Func4   augFuncMtrx[4][5];

    /* Assign the function  matrix  of derivatives*/
    augFuncMtrx[0][0] = f0FuncDx0;  augFuncMtrx[0][1] = f0FuncDx1;
       augFuncMtrx[0][2] = f0FuncDx2;  augFuncMtrx[0][3] = f0FuncDx3;
    augFuncMtrx[1][0] = f1FuncDx0;  augFuncMtrx[1][1] = f1FuncDx1;
    augFuncMtrx[1][2] = f1FuncDx2;  augFuncMtrx[1][3] = f1FuncDx3;
    augFuncMtrx[2][0] = f2FuncDx0;  augFuncMtrx[2][1] = f2FuncDx1;
    augFuncMtrx[2][2] = f2FuncDx2;  augFuncMtrx[2][3] = f2FuncDx3;
    augFuncMtrx[3][0] = f3FuncDx0;  augFuncMtrx[3][1] = f3FuncDx1;
    augFuncMtrx[3][2] = f3FuncDx2; augFuncMtrx[3][3] = f3FuncDx3;

    /* Augment the function  */
    augFuncMtrx[0][4] = fFunc0;  augFuncMtrx[1][4] = fFunc1;
    augFuncMtrx[2][4] = fFunc2;  augFuncMtrx[3][4] = fFunc3;
```

```
    printf("\n   *** Newton-Raphson for 4D *** ");
    printf("\n\n-------------------------------------------");
    printf("\n\n Using: f0(x0,x1,x2,x3)  = x0 - 7.2 * x3^2 - 392.28 ");
    printf("\n\n        f1(x0,x1,x2,x3)  = x0 - 810 + 25 * x1 + 3.75 * x1^2 ");
    printf("\n\n        f2(x0,x1,x2,x3)  = x0 - 900 + 65 * x2 + 30 * x2^2 ");
    printf("\n\n        f3(x0,x1,x2,x3)  = x3 - x1 - x2 ");
    printf("\n\n-------------------------------------------");
    printf("\n\n");

       /* Input stage */
    for(i =0;i <4; i++)
    {
        printf(" Enter the initial value P%d : ",i);
        scanf("%lf",&nPn[i]);
        /* this will clean up the input buffer */
        fflush(stdin);
    }

    printf(" Enter the maximum number of iteration : ");
    scanf("%d",&nMaxIter);   fflush(stdin);

       /* Compute NR4D */
    nStatus = computeNewton4D(nPn,&augFuncMtrx[0][0],nMaxIter,1,nPnOut,&nIterCnt);

       /* Dispay output */
    switch(nStatus)
    {
    case 0:
        printf("\n Solution was found with the desired tolerance.\n");
        printf("\n x0 = %2.8lf",nPnOut[0]);
        printf("\n x1 = %2.8lf",nPnOut[1]);
        printf("\n x2 = %2.8lf",nPnOut[2]);
        printf("\n x3 = %2.8lf\n",nPnOut[3]);
        printf("\n Number of iterations: %d",nIterCnt);
        break;
    case -1:
        printf("\n Failed: Memory allocation in error.");
        break;
    case -2:
        printf("\n Failed: Solution cannot be solve as "
                        " it yeild to a singular matrix.");
        break;
    case -3:
        printf("\n Failed: Solution exceeded the maximum number of iterations.");
        break;
    default:
        break;
    }

    printf("\n\nPress any key to continue....");

    getch();
    return 0;
}
```

```
double  fFunc0(double x0,double x1,double x2,double x3)
{    return (x0 - 7.2 * pow(x3,2) - 392.28); }

double  fFunc1(double x0,double x1,double x2,double x3)
{    return (x0 - 810 + 25 * x1 + 3.75 * pow(x1,2)); }

double  fFunc2(double x0,double x1,double x2,double x3)
{    return (x0 - 900 + 65 * x2 + 30 * pow(x2,2)); }

double  fFunc3(double x0,double x1,double x2,double x3)
{    return (x3 - x1 - x2); }

double  f0FuncDx0(double x0,double x1,double x2,double x3)
{    return 1; }

double  f0FuncDx1(double x0,double x1,double x2,double x3)
{    return 0; }
double  f0FuncDx2(double x0,double x1,double x2,double x3)
{    return 0; }
double  f0FuncDx3(double x0,double x1,double x2,double x3)
{    return (-1 * 14.4 * x3); }
double  f1FuncDx0(double x0,double x1,double x2,double x3)
{    return 1;}

double  f1FuncDx1(double x0,double x1,double x2,double x3)
{    return (25 + 7.5 * x1);}
double  f1FuncDx2(double x0,double x1,double x2,double x3)
{    return 0; }
double  f1FuncDx3(double x0,double x1,double x2,double x3)
{    return 0; }
double  f2FuncDx0(double x0,double x1,double x2,double x3)
{    return 1; }
double  f2FuncDx1(double x0,double x1,double x2,double x3)
{    return 0; }
double  f2FuncDx2(double x0,double x1,double x2,double x3)
{    return (65 + 60 * x2); }
double  f2FuncDx3(double x0,double x1,double x2,double x3)
{    return 0; }
double  f3FuncDx0(double x0,double x1,double x2,double x3)
{    return 0; }
double  f3FuncDx1(double x0,double x1,double x2,double x3)
{    return  -1; }
double  f3FuncDx2(double x0,double x1,double x2,double x3)
{    return -1;  }
double  f3FuncDx3(double x0,double x1,double x2,double x3)
{    return  1; }
```

```
#include "Newton4dFunc.h"
#include "UpperTriangle.h"


/*Function name       : computeNewton4D
* Description         : Computes a 4 dimesional non-linear system using Newton-Raphson
* Parameter(s)        :
*     nPnVect          [in]  The input vector nPnVect[0],nPnVect[1],nPnVect[2],nPnVect[3]
*     *pAugFuncMatrix  [in]  The Augmented matrix of function pointers.
*     nMaxIter         [in]  The maximum number of iterations.
*     nPrint           [in]  Flag for printing the output on screen,
*     nPnOutVect       [out] Output vector.
*     pIterCnt         [out] Number of iterations taken.
* Return              :
*    int   0 - Sucess, -1 - Mem error, -2 - Singular matrix, -3 - Iteration exceeded.
*/
int _cdecl computeNewton4D(double* nPnVect,Func4 *pAugFuncMatrix,int nMaxIter,int
nPrint,double* nPnOutVect,int* pIterCnt)
{
    int nRet = 0;
    int nDone = 0;
    static const  double  nDelta   = 10e-8;
    static const  double  nEpsilon = 10e-8;
    static const  double  nSmall   = 10e-8;
    double nRelError = 0;
    double  nPnOp[4];                /* The duplicate value of P */
    double  nPn1Op[4];               /* Pn+1 */
    double  deltaPVector[4];         /* The Change in P vetor */
    double  augOprMtrx[4][5];        /* Matrix used for the computation */
    double  *pAugOpMatrix = NULL;
    Func4   pFuncCurrent  = NULL;     /* Pointer to the current function */
    int   k;
    int   i,j;
    /* init the operation */
    nPnOp[0] = nPnVect[0];
    nPnOp[1] = nPnVect[1];
    nPnOp[2] = nPnVect[2];
    nPnOp[3] = nPnVect[3];
    if(nPrint)
    {
        printf("\n\n    N    %-10s   %-10s  %-10s   %-10s   ","P0n","P1n","P2n","P3n");
        printf("\n-------------------------------------------------------------------");
        printf("\n   %2d  %2.8lf  %2.8lf  %2.8lf  %2.8lf  "
              ,0,nPnOp[0],nPnOp[1],nPnOp[2],nPnOp[3]);
    }
    /* use the pointer version */
    pAugOpMatrix = &augOprMtrx[0][0];
    for(k = 1; k <= nMaxIter && nRet == 0 && nDone == 0; k++)
    {
        /* Assuming that pAugFuncMatrix is a 4x5 Augmented matrix */
        /* Get the results of the partial function derivatives */
        /* a[i,j] = df[i,j] (x0,x1,x2,x3); */
        for(i = 0; i < 4; i++)
        {
            for(j = 0; j < 4; j++)
            {
                pFuncCurrent = MATRIXE(pAugFuncMatrix,i,j,5);
                MATRIXE(pAugOpMatrix,i,j,5) =
                    pFuncCurrent(nPnOp[0],nPnOp[1],nPnOp[2],nPnOp[3]);
            }
        }
```

```
        /* Compute the Augmented vector F(x0,x1,x2,x3) */
        for(i = 0; i < 4; i++)
        {
            pFuncCurrent = MATRIXE(pAugFuncMatrix,i,4,5);
            MATRIXE(pAugOpMatrix,i,4,5)
                = -1 * pFuncCurrent(nPnOp[0],nPnOp[1],nPnOp[2],nPnOp[3]);
        }
        /* Use Linear systems operations */
        nRet = upperTriangleAndBackSub(pAugOpMatrix,4,deltaPVector);
        if(nRet == 0)
        {
            double nSumDelta = 0;
            double nSumPn1   = 0;
            double nSumFunc  = 0;
            for(j=0; j < 4; j++)
            {
                /* Get the new value */
                nPn1Op[j] = nPnOp[j] + deltaPVector[j];
                nSumDelta += fabs(deltaPVector[j]);
                nSumPn1   += fabs(nPn1Op[j]);
            }
            for(i = 0; i < 4; i++)
            {
                /* Get the  - fj(x0,x1,x2,x3) */
                pFuncCurrent = MATRIXE(pAugFuncMatrix,i,4,5);
                nSumFunc +=  fabs(pFuncCurrent(nPn1Op[0],nPn1Op[1],nPn1Op[2],nPn1Op[3]));
            }

            nSumPn1   += nSmall;
            /* Compute the relative error */
            nRelError = nSumDelta/nSumPn1;
            if(nRelError < nDelta && nSumFunc < nEpsilon)
                nDone = 1;

            /* Transfer the Values */
            for(j=0; j < 4; j++)
                nPnOp[j] = nPn1Op[j];
            if(nPrint)
                printf("\n   %2d  %2.8lf  %2.8lf  %2.8lf  %2.8lf
",k,nPnOp[0],nPnOp[1],nPnOp[2],nPnOp[3]);
        }
    }

    if(nRet == 0)
    {
        /* iteration exceeded */
        if(nDone == 0)
            nRet = -3;

        for(j=0; j < 4; j++)
            /* Copy it to the output */
            nPnOutVect[j] = nPnOp[j];

        *pIterCnt = k;
    }

    if(nPrint)
        printf("\n----------------------------------------------------------------------");

    return nRet;
}
```

Program Output

```
   *** Newton-Raphson for 4D ***
-----------------------------------------------

Using: f0(x0,x1,x2,x3)  = x0 - 7.2 * x3^2 - 392.28

       f1(x0,x1,x2,x3)  = x0 - 810 + 25 * x1 + 3.75 * x1^2

       f2(x0,x1,x2,x3)  = x0 - 900 + 65 * x2 + 30 * x2^2

       f3(x0,x1,x2,x3)  = x3 - x1 - x2

-----------------------------------------------

Enter the initial value P0 : 750
Enter the initial value P1 : 3
Enter the initial value P2 : 1.5
Enter the initial value P3 : 5
Enter the maximum number of iteration : 20

   N    P0n           P1n          P2n          P3n
-------------------------------------------------------------------------
   0  750.00000000  3.00000000  1.50000000  5.00000000
   1  651.15695568  4.05459041  2.04092287  6.09551327
   2  650.47685694  3.99159563  1.99772417  5.98931980
   3  650.48730130  3.99113463  1.99736483  5.98849947
   4  650.48730189  3.99113461  1.99736481  5.98849942
-------------------------------------------------------------------------
Solution was found with the desired tolerance.

x0 = 650.48730189
x1 = 3.99113461
x2 = 1.99736481
x3 = 5.98849942

Number of iterations: 5

Press any key to continue....
```

Upper Triangularization Followed by Back Substitution

Discussion

Solving linear systems with *n* equations and *n* unknowns is not as hard as the nonlinear solution especially if one has knowledge in manipulating them using matrices. Given a linear system

$$AX = B \tag{1}$$

where A is the *n x n* coefficient matrix, **X** is the unknown vector, and **B** is the constant vector, one can construct an equivalent system

$$UX = Y \tag{2}$$

where **U** is the equivalent upper triangular matrix. Once (2) is constructed, it can now be used to solve for the unknown vector **X** by back substitution, as shown below.

$$\begin{aligned} X_n &= \frac{Y_n}{U_{n,n}} \\ X_r &= \frac{B_r - \sum_{j=r+1}^{n} U_{r,j} X_j}{U_{r,r}} \qquad for\ r = n\text{-}1,\ldots 1. \end{aligned} \tag{3}$$

The construction of (2) from (1) requires the Gaussian elimination technique by:

1) Creating an *n x n*+1 augmented matrix [A, B].
2) Selecting the pivot $A_{r,r}$, for r=0... n-1.
3) Eliminating the column below the pivot using the row operations.

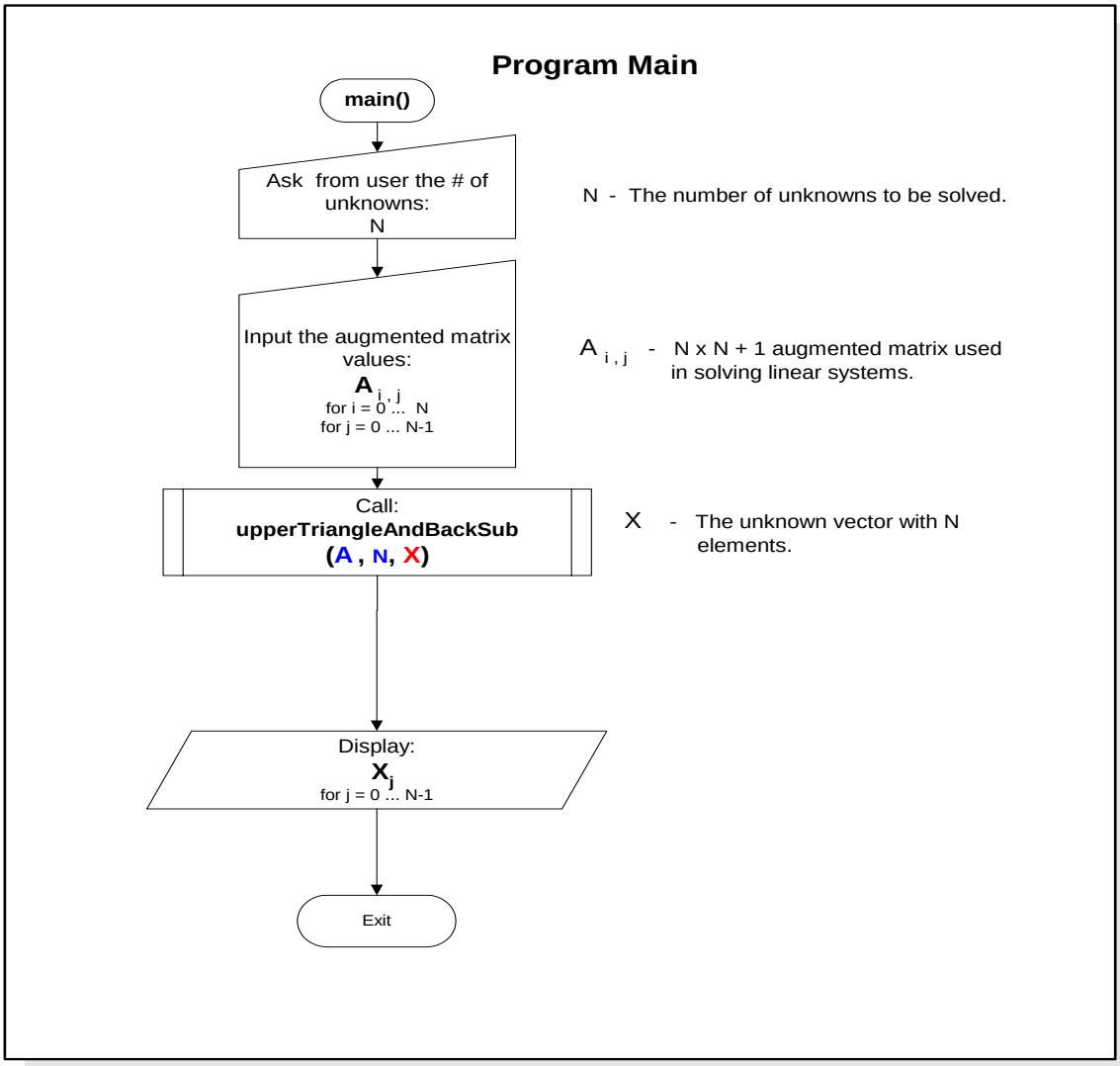
More details are illustrated in **Gaussian Elimination and Pivoting** on pp. 148-151 of the textbook.

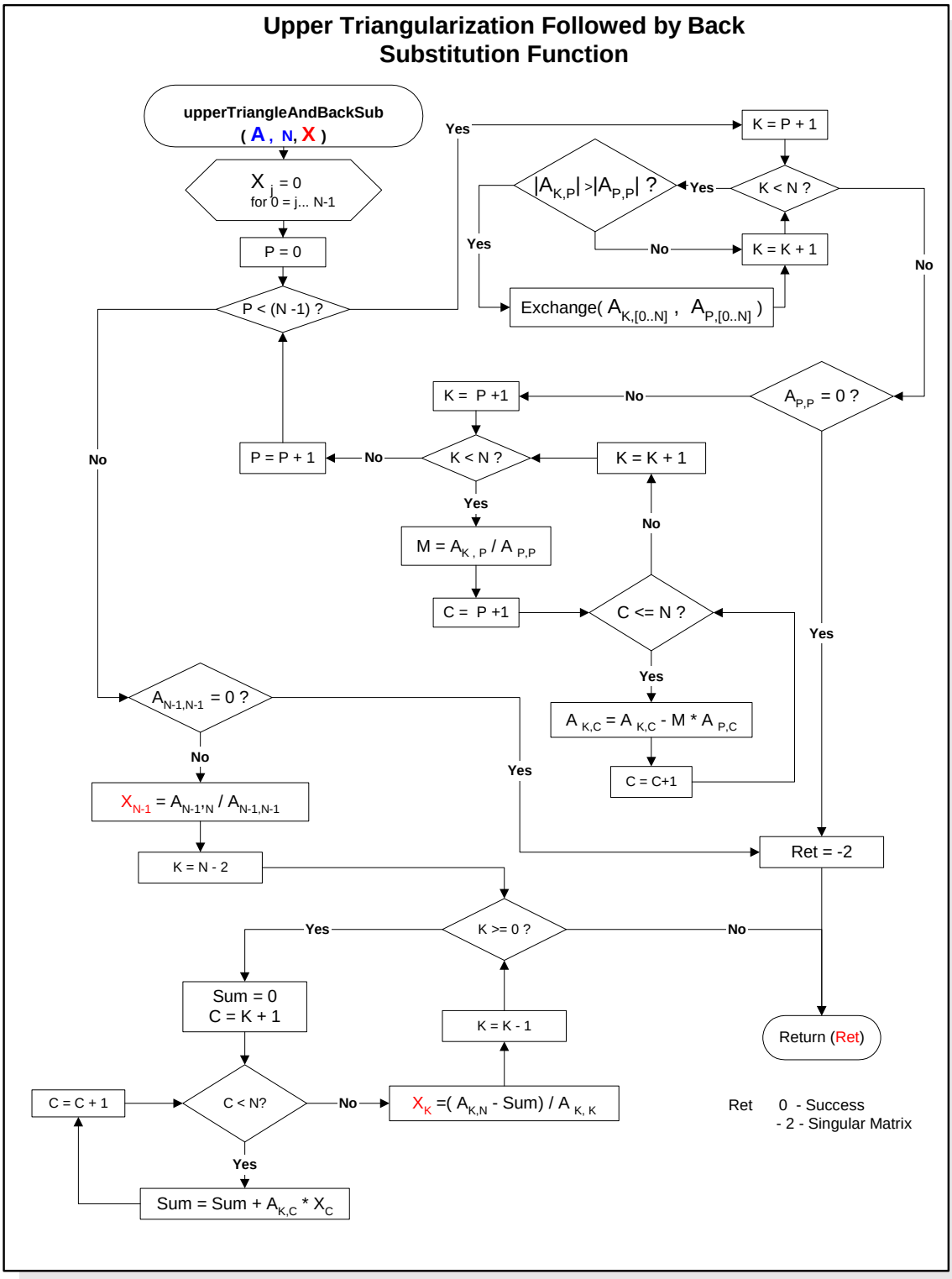
There are also cases when the pivot is 0 in some rows while performing the process. These instances may lead to a divide error since the pivot is used as the divisor in finding for the multiplier *m*. To avoid such from happening, the process must undergo a row interchange with any row below the pivot if there are nonzero elements under the pivot's column. This process is called pivoting and may only succeed if the matrix is nonsingular.

Pivoting can also be taken advantage of if there are more than one nonzero elements under the pivot's column. Since larger magnitudes of a divisor usually reduces precession errors, then the row below the pivot with the largest value of element under the pivot's column shall be chosen.

Algorithm 3.2 (Upper Triangularization Followed by Back Substitution). To construct the solution to AX=B, by first reducing the augmented matrix [A,B] to upper-triangular form and then performing back substitution.

Program Flow




Upper Triangularization Followed by Back Substitution Function
upperTriangleAndBackSub (A , N, X)
X j = 0 for 0 = j... N-1
P = 0
P < (N -1) ?
Yes
No
K = P + 1
|AK,P| >|AP,P| ?
Yes
No
K < N ?
Yes
No
K = K + 1
Exchange(AK,[0..N] , AP,[0..N])
AP,P = 0 ?
No
Yes
K = P +1
No
K < N ?
Yes
No
K = K + 1
P = P + 1
M = AK , P / A P,P
C = P +1
C <= N ?
Yes
No
A K,C = A K,C - M * A P,C
C = C+1
AN-1,N-1 = 0 ?
No
Yes
XN-1 = AN-1'N / AN-1,N-1
K = N - 2
Ret = -2
K >= 0 ?
Yes
No
Sum = 0
C = K + 1
K = K - 1
Return (Ret)
C = C + 1
C < N?
No
Yes
XK =(AK,N - Sum) / A K, K
Sum = Sum + AK,C * XC
Ret 0 - Success
- 2 - Singular Matrix

Program Source Code

```
#include <conio.h>
#include "UpperTriangle.h"
void  printEquations(double* pAugMtrx,int nNumUnknown);
int main(int argc, char* argv[])
{
    double *pAugMtrx = NULL;
    double *pOutput = NULL;
    int    nNumUnknown;
    int    i,j;
    int    nStatus = 0;  /* -1 - Mem error,  -2 - Singular matrix */
    printf("\n  ***  Upper Triangularization with Back Substitution *** \n");
    /*  This is the input stage */
    printf("\n Enter the number of unknowns  : ");
    scanf("%d",&nNumUnknown); fflush(stdin);
    /* Allocate memory */
    pAugMtrx = (double*) malloc(nNumUnknown * (nNumUnknown + 1) * sizeof(pAugMtrx[0]));
    pOutput          = (double*) malloc(nNumUnknown * sizeof(pOutput[0]));
    if(!pAugMtrx || !pOutput)        nStatus = -1;
    if(nStatus == 0)
    {
        /* Ask for the coefficients & constant*/
        for(i = 0; i < nNumUnknown; i++)
        {
            /* Ask for the coefficients */
            for(j = 0; j < nNumUnknown; j++)
            {
                printf(" Eq. (%d): Enter the coefficient for variable %c: ",i+1,j +
'A');
                scanf("%lf",&MATRIXE(pAugMtrx,i,j,(nNumUnknown + 1)));   fflush(stdin);
            }
            /* Ask for the constant */
            printf(" Eq. (%d): Enter the equation constant: ",i+1);
            scanf("%lf",&MATRIXE(pAugMtrx,i,nNumUnknown,(nNumUnknown + 1)));
fflush(stdin);
        }
        /* print the equations */
        printEquations(pAugMtrx,nNumUnknown);
    }
    /*  Solve the linear system */
    if(nStatus == 0)   nStatus = upperTriangleAndBackSub(pAugMtrx,nNumUnknown,pOutput);
    /*  This is the display output stage */
    if(nStatus == 0)
    {
        printf("\nResult(s): ");
        for(i = 0; i < nNumUnknown; i++) printf("\n %c = %lf ",i + 'A',pOutput[i]);
    }
    /* Free dynamic allocations */
    if(pAugMtrx)    free(pAugMtrx);
    if(pOutput)     free(pOutput);
    switch(nStatus)
    {
    case -1:
        printf("\nMemory allocation error!!");
        break;
    case -2:
        printf("\nFailed: Matrix is singular!!");
        break;
    default:
        break;
    }
    printf("\n\nPress any key to continue...");
    getch();
    return nStatus;
}
```

```
#include "UpperTriangle.h"
/* Function name      : upperTriangleAndBackSub
* Description        : Solves a linear system by upper triangularization followed by
back substitution
* Parameter(s)       :
*     *pAugMtrx       [in]  The ugmented matrix [n x (n + 1)]
*                           Location of A(r,c) is at pAugMtrx[r-1][c-1].
*     nSize           [in]  The size of the matrix.
*     *pOutVect       [out] The ouput vactor.
*                           The first element is starts a pOutVect[0].
* Return             :
*    int                 0 - successful, -1 Memory allocation failed, -2 Matrix maybe
singular.
*/
int __cdecl
upperTriangleAndBackSub(double  *pAugMtrx,int nSize,double *pOutVect)
{
    int nRet = 0;
    int nPvtRowIdx;
    int *pRow = NULL;
    pRow = (int *)malloc(nSize * sizeof(int));

    if(!pRow)
        nRet = -1;

    if(nRet == 0)
    {
        /* Initialize the row index */
        for(nPvtRowIdx = 0; nPvtRowIdx < nSize; nPvtRowIdx++)
            pRow[nPvtRowIdx] = nPvtRowIdx;

        /* Initialize the ouput vector */
        memset(pOutVect,0,sizeof(pOutVect[0]) * nSize);

        /* Start the upper triangularization */
        for(nPvtRowIdx = 0; nPvtRowIdx < (nSize - 1); nPvtRowIdx++)
        {
            int nRowIndex;
            int nPvtColIdx;
            nPvtColIdx = nPvtRowIdx;
            /* Search for the largest pivot element to reduce error */
            for(nRowIndex = nPvtRowIdx + 1; nRowIndex < nSize; nRowIndex++)
            {
                if(fabs(MATRIXE(pAugMtrx,pRow[nRowIndex],nPvtColIdx,nSize + 1)) >
                    fabs(MATRIXE(pAugMtrx,pRow[nPvtRowIdx],nPvtColIdx,nSize + 1)))
                {
                    /* Since there is a larger value then swap the row */
                    int nTemp;
                    nTemp = pRow[nRowIndex];
                    pRow[nRowIndex]  = pRow[nPvtRowIdx];
                    pRow[nPvtRowIdx] = nTemp;
                }
            }

            if(MATRIXE(pAugMtrx,pRow[nPvtRowIdx],nPvtColIdx,(nSize + 1)) == 0)
            {
                /* Failed, Matrix is singular */
                nRet = -2;
                break;
            }
```

```
            /* Now eliminate the elements that is located below the
               pivot element.
            */
            for(nRowIndex = nPvtRowIdx + 1; nRowIndex < nSize; nRowIndex++)
            {
                double  nMul; int i;
                /* Compute the multiplier */
                nMul = MATRIXE(pAugMtrx,pRow[nRowIndex],nPvtColIdx,(nSize + 1)) /
                       MATRIXE(pAugMtrx,pRow[nPvtRowIdx],nPvtColIdx,(nSize + 1));

                MATRIXE(pAugMtrx,pRow[nRowIndex],nPvtColIdx,(nSize + 1)) = 0;

                for(i = nPvtColIdx + 1;i < (nSize + 1); i++)
                {
                    MATRIXE(pAugMtrx,pRow[nRowIndex],i,(nSize + 1)) -=
                        (MATRIXE(pAugMtrx,pRow[nPvtRowIdx],i,(nSize + 1)) * nMul);
                }
            }
        }
    }

    if(nRet == 0)
    {
        /* Check if the AugMatrix[n,n] is 0 */
        if(MATRIXE(pAugMtrx,pRow[nSize - 1],(nSize - 1),(nSize + 1))  == 0)
            nRet = -2;
    }

    if(nRet == 0)
    {
        /* Back substitution */
        pOutVect[nSize -1] =
            MATRIXE(pAugMtrx,pRow[nSize - 1],(nSize),(nSize + 1)) /
            MATRIXE(pAugMtrx,pRow[nSize - 1],(nSize - 1),(nSize + 1));

        for(nPvtRowIdx = nSize - 2; nPvtRowIdx >= 0; nPvtRowIdx--)
        {
            double nSum = 0;
            int    i;

            for(i = nPvtRowIdx + 1; i < nSize; i++)
            {
                nSum += MATRIXE(pAugMtrx,pRow[nPvtRowIdx],i,(nSize + 1)) * pOutVect[i];
            }

            pOutVect[nPvtRowIdx] = (MATRIXE(pAugMtrx,pRow[nPvtRowIdx],nSize,(nSize + 1))
- nSum) /
                MATRIXE(pAugMtrx,pRow[nPvtRowIdx],nPvtRowIdx,(nSize + 1));
        }
    }

    /* Deallocate the memory */
    if(pRow)
        free(pRow);

    return nRet;
}
```

Program Output

```
 ***  Upper Triangularization with Back Substitution ***

Enter the number of unknowns  : 4
Eq. (1): Enter the coefficient for variable A: 1
Eq. (1): Enter the coefficient for variable B: 2
Eq. (1): Enter the coefficient for variable C: 1
Eq. (1): Enter the coefficient for variable D: 4
Eq. (1): Enter the equation constant: 13
Eq. (2): Enter the coefficient for variable A: 2
Eq. (2): Enter the coefficient for variable B: 0
Eq. (2): Enter the coefficient for variable C: 4
Eq. (2): Enter the coefficient for variable D: 3
Eq. (2): Enter the equation constant: 28
Eq. (3): Enter the coefficient for variable A: 4
Eq. (3): Enter the coefficient for variable B: 2
Eq. (3): Enter the coefficient for variable C: 2
Eq. (3): Enter the coefficient for variable D: 1
Eq. (3): Enter the equation constant: 20
Eq. (4): Enter the coefficient for variable A: -3
Eq. (4): Enter the coefficient for variable B: 1
Eq. (4): Enter the coefficient for variable C: 3
Eq. (4): Enter the coefficient for variable D: 2
Eq. (4): Enter the equation constant: 6

Equations:
-------------------------------------------------
Eq. (1): 1.000000 A + 2.000000 B + 1.000000 C + 4.000000 D = 13.000000
Eq. (2): 2.000000 A + 4.000000 C + 3.000000 D = 28.000000
Eq. (3): 4.000000 A + 2.000000 B + 2.000000 C + 1.000000 D = 20.000000
Eq. (4): -3.000000 A + 1.000000 B + 3.000000 C + 2.000000 D = 6.000000

-------------------------------------------------

Result(s):
 A = 3.000000
 B = -1.000000
 C = 4.000000
 D = 2.000000

Press any key to continue...
```

PA = LU Factorization with Pivoting

Discussion

Another way of solving linear systems employing the use of matrices is the triangular factorization or LU factorization with pivoting. In Gaussian elimination, the linear system AX = B is solved by adjoining the right hand side vector B to the coefficient matrix A, then performing row combinations that would zero out the subdiagonal entries of the matrix A. LU factorization does the same operations, but ignores the right hand side vector B until after the matrix has been processed.

Given a linear system in the form

$$AX = B \tag{1}$$

one must first solve the lower triangular matrix L (the ones on main diagonal, zero above main diagonal), the upper triangular matrix U (zeros below the main diagonal), and the permutation matrix P that satisfies

$$PA = LU \tag{2}$$

Then, a temporary vector Y shall be created out of L and PB as shown below

$$LY = PB \tag{3}$$

using forward substitution. Finally, X will be solved using the values of U and Y as shown below

$$UX = Y \tag{4}$$

using back substitution.

The details on Triangular Factorization are discussed on pp. 166 – 171 of the textbook.

To avoid confusion when analyzing the actual implementation of this method, the code has a little variance based on what it shown in (2), (3) and (4). Instead of creating 2 different matrices for L and U, they were just merged into one as illustrated in a sample 4 x 4 matrix below:

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ m_{2,1} & 1 & 0 & 0 \\ m_{3,1} & m_{3,2} & 1 & 0 \\ m_{4,1} & m_{4,2} & m_{4,3} & 1 \end{bmatrix} = L \qquad \begin{bmatrix} u_{1,1} & u_{1,2} & u_{1,3} & u_{1,4} \\ 0 & u_{2,2} & u_{2,3} & u_{2,4} \\ 0 & 0 & u_{3,3} & u_{3,4} \\ 0 & 0 & 0 & u_{4,4} \end{bmatrix} = U$$

The ideal way where the L and U are separate matrices

$$\begin{bmatrix} u_{1,1} & u_{1,2} & u_{1,3} & u_{1,4} \\ m_{2,1} & u_{2,2} & u_{2,3} & u_{2,4} \\ m_{3,1} & m_{3,2} & u_{3,3} & u_{3,4} \\ m_{4,1} & m_{4,2} & m_{4,3} & u_{4,4} \end{bmatrix} = LU$$

The actual implementation where matrix L and U are merge and the ones in the diagonal of L are ignored.

Figure 3.3.1 L and U are merged into a single Matrix.

The reason for this variance is to save the memory bulk used by the ideal method since any of them can still be restored on-the-fly by masking out the other. Another variance made in the actual implementation to save memory bulk is the replacement of the permutation matrix with an indirect indexing vector as shown in figure below.

$$\begin{bmatrix} 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix} = P \qquad \begin{bmatrix} 3 \\ 4 \\ 2 \\ 1 \end{bmatrix} = P$$

Permutation Matrix Indirect row indexing vector

Figure 3.3.2 Permutation matrix is replaced with an indirect indexing vector for row interchange tracking.

Algorithm 3.3 (PA =LU Factorization with Pivoting). To construct the solution to AX=B by performing the steps.

1. Find the permutation matrix P, lower-triangular matrix L, and upper-triangular matrix U that satisfy

$$PA = LU.$$

2. Compute PB and compute the equivalent linear system

$$LUX = PB.$$

3. Solve the lower-triangular system

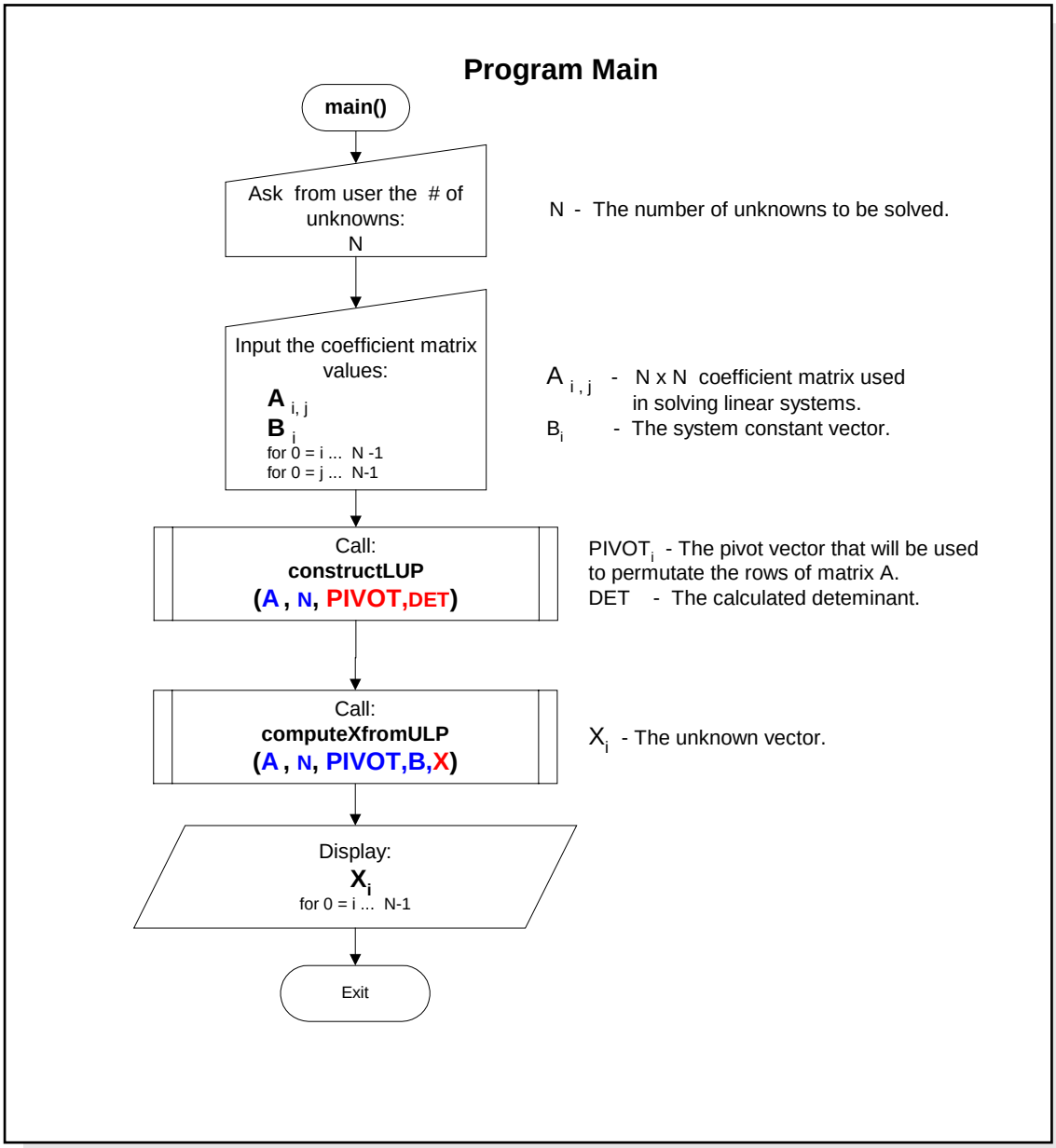
$$LY = PB \text{ for } Y.$$

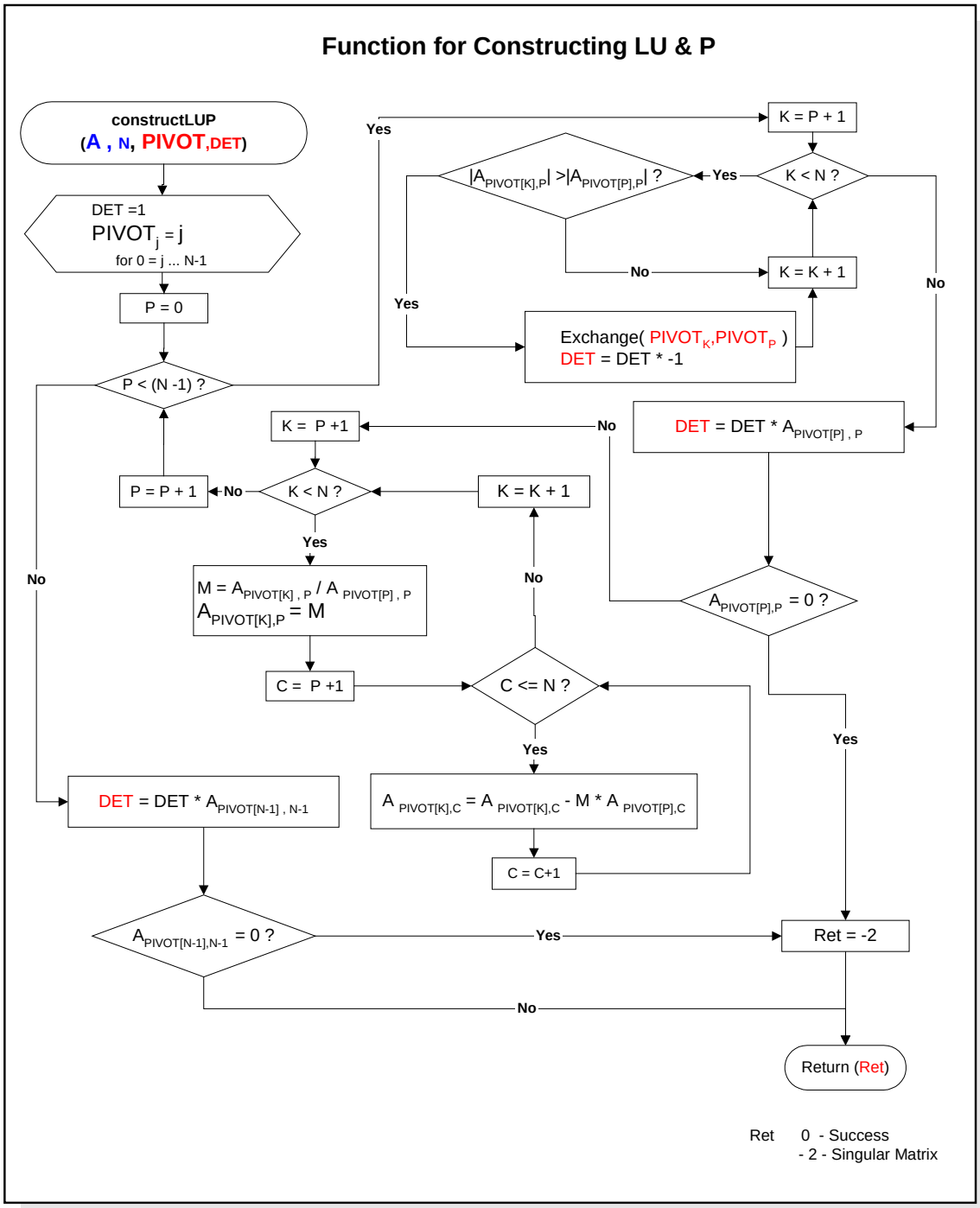
4. Solve the upper-triangular system

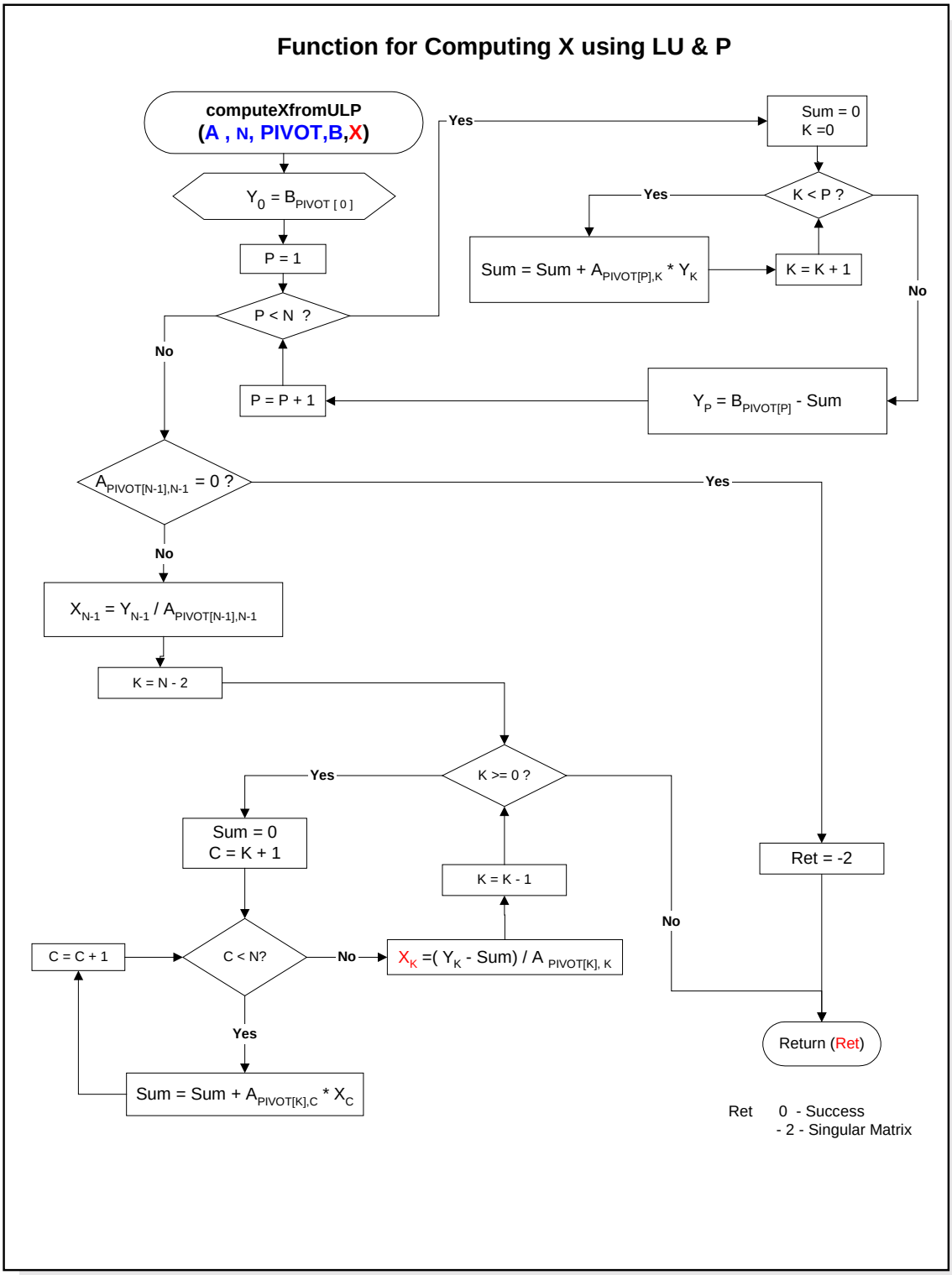
$$UX = Y \text{ for } X.$$

Remarks. This algorithm is an extension of Algorithm 3.2. Since the diagonal elements of L are all 1's, these values do not need to be stored: The coefficients of L below the main diagonal and the nonzero coefficients of U overwrite the matrix A.

Program Flow




Function for Constructing LU & P
constructLUP
(A , N, PIVOT,DET)
DET =1
PIVOT_j = j
for 0 = j ... N-1
P = 0
P < (N -1) ?
K = P + 1
|A_PIVOT[K],P| >|A_PIVOT[P],P| ?
Yes
K < N ?
Yes
No
K = K + 1
No
Yes
Exchange(PIVOT_K,PIVOT_P)
DET = DET * -1
No
DET = DET * A_PIVOT[P] , P
K = P +1
No
P = P + 1
No
K < N ?
K = K + 1
Yes
No
M = A_PIVOT[K] , P / A PIVOT[P] , P
A_PIVOT[K],P = M
A_PIVOT[P],P = 0 ?
C = P +1
C <= N ?
Yes
Yes
DET = DET * A_PIVOT[N-1] , N-1
A PIVOT[K],C = A PIVOT[K],C - M * A PIVOT[P],C
C = C+1
A_PIVOT[N-1],N-1 = 0 ?
Yes
Ret = -2
No
Return (Ret)
Ret 0 - Success
- 2 - Singular Matrix


Function for Computing X using LU & P
computeXfromULP
(A , N, PIVOT,B,X)
Y0 = BPIVOT [0]
P = 1
P < N ?
Yes
No
Sum = 0
K =0
K < P ?
Yes
No
Sum = Sum + APIVOT[P],K * YK
K = K + 1
YP = BPIVOT[P] - Sum
P = P + 1
APIVOT[N-1],N-1 = 0 ?
Yes
No
XN-1 = YN-1 / APIVOT[N-1],N-1
K = N - 2
K >= 0 ?
Yes
No
Sum = 0
C = K + 1
K = K - 1
C = C + 1
C < N?
No
Yes
XK =(YK - Sum) / A PIVOT[K], K
Sum = Sum + APIVOT[K],C * XC
Ret = -2
Return (Ret)
Ret 0 - Success
- 2 - Singular Matrix

Program Source Code

```
#include <stdio.h>
#include <conio.h>
#include "LUFactoring.h"
void  printEquations(double* pMatrix,double* pInputB,int nNumUnknown);

int main(int argc, char* argv[])
{
    double *pMatrix = NULL;
    int    *pPvtRowVect = NULL;
    double *pInputB = NULL;
    double *pOutputX = NULL;
    double deteminant;  int    nNumUnknown;    int    i,j;
    int    nStatus = 0;  /* -1 - Mem error,  -2 - Singular matrix */
    printf("\n  ***  PA = LU Factorization with Pivoting *** \n");
    /*  input stage */
    printf("\n Enter the number of unknowns : ");
    scanf("%d",&nNumUnknown);    fflush(stdin);
    /* Allocate memory */
    pMatrix = (double*) malloc(nNumUnknown * nNumUnknown * sizeof(pMatrix[0]));
    pPvtRowVect     = (int*) malloc(nNumUnknown * sizeof(pPvtRowVect[0]));
    pInputB         = (double*) malloc(nNumUnknown * sizeof(pInputB[0]));
    pOutputX        = (double*) malloc(nNumUnknown * sizeof(pOutputX[0]));
    if(!pMatrix || !pOutputX || !pPvtRowVect || !pInputB)      nStatus = -1;
    if(nStatus == 0)
    {   /* Ask for the coefficients & constant*/
        for(i = 0; i < nNumUnknown; i++)
        {   /* Ask for the coefficients */
            for(j = 0; j < nNumUnknown; j++)
            {
                printf(" Eq. (%d): Enter the coefficient for variable %c: ",i+1,j + 'A');
                scanf("%lf",&MATRIXE(pMatrix,i,j,nNumUnknown));  fflush(stdin);
            }
            /* Ask for the constant */
            printf(" Eq. (%d): Enter the equation constant: ",i+1);
            scanf("%lf",&pInputB[i]);  fflush(stdin);
        }
        /* print the equations */
        printEquations(pMatrix,pInputB,nNumUnknown);
    }

    /*  Computation */
    if(nStatus == 0) nStatus = constructLUP(pMatrix,nNumUnknown,pPvtRowVect,&deteminant);
    if(nStatus == 0) nStatus =
computeXfromULP(pMatrix,nNumUnknown,pPvtRowVect,pInputB,pOutputX);
    /*  This is the display output stage */
    if(nStatus == 0)
    {
        printf("\nResult(s): ");
        for(i = 0; i < nNumUnknown; i++)
            printf("\n %c = %lf ",i + 'A',pOutputX[i]);
    }

    /* Free dynamic allocations */
    if(pMatrix)     free(pMatrix);
    if(pPvtRowVect) free(pPvtRowVect);
    if(pInputB)     free(pInputB);
    if(pOutputX)    free(pOutputX);

    switch(nStatus)
    {
    case -1:
```

```
        printf("\nMemory allocation error!!");
        break;
    case -2:
        printf("\nFailed: Matrix is singular!!");
        break;
    default:
        break;
    }
    printf("\n\nPress any key to continue...");
    getch();
    return nStatus;
}
```

```
#include "LUFactoring.h"
/* Function name      : constructLUP
* Description         : Constructs the L and U  of a matrix and the vector
*                       containing the row indices as a result of row interchange.
* Parameter(s)        :
*     *pMatrix             [in]   The coefficient matrix [n x n]
*                                 Location of A(r,c) is at pMatrix[r-1][c-1].
*     nSize                [in]  The size of the matrix.
*     *pPvtRowVect         [out] The row indices due to permutation/ row interchange.
*     pDeterminant         [out] The determinant of the matrix.
* Return              :
*    int                  0 of successful, -1 - Memory allocation failed,
*                         -2 - Matrix maybe non-singular.
*/
int __cdecl
constructLUP(double  *pMatrix,int nSize,int *pPvtRowVect,double* pDeterminant)
{
    int nRet = 0;
    int nPvtRowIdx;
    int *pRow = pPvtRowVect;
    *pDeterminant = 1;
    /* Initialize the row index */
    for(nPvtRowIdx = 0; nPvtRowIdx < nSize; nPvtRowIdx++) pRow[nPvtRowIdx] = nPvtRowIdx;
    /* Start the upper triangularization */
    for(nPvtRowIdx = 0; nPvtRowIdx < (nSize - 1); nPvtRowIdx++)
    {
        int nRowIndex;
        int nPvtColIdx;   nPvtColIdx = nPvtRowIdx;
        /* Search for the largest pivot element to reduce error */
        for(nRowIndex = nPvtRowIdx + 1; nRowIndex < nSize; nRowIndex++)
        {
            if(fabs(MATRIXE(pMatrix,pRow[nRowIndex],nPvtColIdx,nSize)) >
                fabs(MATRIXE(pMatrix,pRow[nPvtRowIdx],nPvtColIdx,nSize)))
            {
                /* Since there is a larger value then swap the row */
                int nTemp;
                nTemp = pRow[nRowIndex];
                pRow[nRowIndex]  = pRow[nPvtRowIdx];
                pRow[nPvtRowIdx] = nTemp;
                /* Change the sign of the determinant */
                (*pDeterminant) *= -1;
            }
        }
```

```
        (*pDeterminant) *= MATRIXE(pMatrix,pRow[nPvtRowIdx],nPvtColIdx,nSize);
        /* Check if the pivot element is row */
        if(MATRIXE(pMatrix,pRow[nPvtRowIdx],nPvtColIdx,nSize) == 0)
        {
            nRet = -2;
            break;
        }
        /* Now eliminate the elements that is located below the
           pivot element.
        */
        for(nRowIndex = nPvtRowIdx + 1; nRowIndex < nSize; nRowIndex++)
        {
            double  nMul;
            int i;
            /* Compute the multiplier */
            nMul = MATRIXE(pMatrix,pRow[nRowIndex],nPvtColIdx,(nSize)) /
                   MATRIXE(pMatrix,pRow[nPvtRowIdx],nPvtColIdx,(nSize));
            /* Assing the multiplier*/
            MATRIXE(pMatrix,pRow[nRowIndex],nPvtColIdx,(nSize)) = nMul;
            /* Adjust the remaining values of the row*/

            for(i = nPvtColIdx + 1;i < nSize; i++)
            {
                MATRIXE(pMatrix,pRow[nRowIndex],i,(nSize)) -=
                    (MATRIXE(pMatrix,pRow[nPvtRowIdx],i,(nSize)) * nMul);
            }
        }
    }
    if(nRet == 0)
    {
        (*pDeterminant) *= MATRIXE(pMatrix,pRow[nSize - 1],(nSize - 1),nSize);
        /* Check if the Matrix[n,n] is 0 */
        if(MATRIXE(pMatrix,pRow[nSize - 1],(nSize - 1),nSize)  == 0) nRet = -2;
    }
    return nRet;
}
```

```
/* Function name      : computeXfromULP
* Description        : Computes the unknown vector from and LU and P.
* Parameter(s)       :
*     *pMatrix            [in]  The ugmented matrix [n x n]
*                             Location of A(r,c) is at pMatrix[r-1][c-1].
*     nSize               [in]  The size of the matrix.
*     *pPvtRowVect        [in]  The row indices due to permutation/ row interchange.
*     pInputB             [in]  The constants.
*     pOuputX             [out] The unknown vector.
* Return             :
*    int                 0 of successful, -1 - Memory allocation failed.
*                        -2 - Matrix maybe non-singular.
*/
int  __cdecl
computeXfromULP(double  *pMatrix,int nSize,int *pPvtRowVect,double* pInputB,double*
pOuputX)
{
    int nRet = 0;
    int      *pRow = pPvtRowVect;
    double   *pY   = NULL;
    int nIdx;
    /* Allocate the storage for the y */
    pY = (double*) malloc(nSize * sizeof(double));
    if(!pY)  nRet = -1;
    if(nRet == 0)
    {   /* Compute Y using the forward substitution */
        pY[0] = pInputB[pRow[0]];
        for(nIdx = 1; nIdx < nSize; nIdx++)
        {
            double  nSum = 0;
            int    i;
            for(i = 0; i < nIdx; i++)
                nSum += (MATRIXE(pMatrix,pRow[nIdx],i,nSize) * pY[i]);
            /* get the value of the current y */
            pY[nIdx] = pInputB[pRow[nIdx]] - nSum;
        }
        /* Check if the Matrix[n,n] is 0 */
        if(MATRIXE(pMatrix,pRow[nSize - 1],(nSize - 1),nSize)  == 0)
            nRet = -2;
    }
    if(nRet == 0)
    {   /* Start the back substitution */
        pOuputX[nSize -1] = pY[nSize -1]
                                    / MATRIXE(pMatrix,pRow[nSize - 1],(nSize - 1),nSize);
        for(nIdx = nSize - 2; nIdx >= 0; nIdx--)
        {
            double  nSum = 0;
            int    i;
            for(i = nIdx+1; i < nSize; i++)
            {
                nSum += MATRIXE(pMatrix,pRow[nIdx],i,nSize) * pOuputX[i];
            }
            if(MATRIXE(pMatrix,pRow[nIdx],nIdx,nSize) == 0)
            {   /* Failed, Matrix is singular */
                nRet = -2;
                break;
            }
            /* pMatrix[pRow[nIdx],nIdx] is assummed non-0 if constructLUP
                        has been called using same parameters*/
            pOuputX[nIdx] = (pY[nIdx] - nSum) / MATRIXE(pMatrix,pRow[nIdx],nIdx,nSize) ;
        }
    }
    return nRet;
}
```

Program Output

```
  ***  PA = LU Factorization with Pivoting ***

 Enter the number of unknowns : 4
 Eq. (1): Enter the coefficient for variable A: 1
 Eq. (1): Enter the coefficient for variable B: 2
 Eq. (1): Enter the coefficient for variable C: 1
 Eq. (1): Enter the coefficient for variable D: 4
 Eq. (1): Enter the equation constant: 13
 Eq. (2): Enter the coefficient for variable A: 2
 Eq. (2): Enter the coefficient for variable B: 0
 Eq. (2): Enter the coefficient for variable C: 4
 Eq. (2): Enter the coefficient for variable D: 3
 Eq. (2): Enter the equation constant: 28
 Eq. (3): Enter the coefficient for variable A: 4
 Eq. (3): Enter the coefficient for variable B: 2
 Eq. (3): Enter the coefficient for variable C: 2
 Eq. (3): Enter the coefficient for variable D: 1
 Eq. (3): Enter the equation constant: 20
 Eq. (4): Enter the coefficient for variable A: -3
 Eq. (4): Enter the coefficient for variable B: 1
 Eq. (4): Enter the coefficient for variable C: 3
 Eq. (4): Enter the coefficient for variable D: 2
 Eq. (4): Enter the equation constant: 6

 Equations:
 -------------------------------------------------
Eq. (1): 1.000000 A + 2.000000 B + 1.000000 C + 4.000000 D = 13.000000
Eq. (2): 2.000000 A + 4.000000 C + 3.000000 D = 28.000000
Eq. (3): 4.000000 A + 2.000000 B + 2.000000 C + 1.000000 D = 20.000000
Eq. (4): -3.000000 A + 1.000000 B + 3.000000 C + 2.000000 D = 6.000000

 -------------------------------------------------

Result(s):
 A = 3.000000
 B = -1.000000
 C = 4.000000
 D = 2.000000

Press any key to continue...
```

Jacobi Iteration

Discussion

Iterative methods are not only limited to nonlinear equations and systems, but principles used in Fixed-Point iteration may also be extended to linear systems with the form **AX = B**. Such method is called the **Jacobi Iteration.**

Given a system of equations in the form:

$$\begin{array}{ccccccccccc} a_{1,1}x_1 & + & a_{1,2}x_2 & + \ldots & + & a_{1,j}x_j & + \ldots & + & a_{1,N}x_N & = & b_1 \\ a_{2,1}x_1 & + & a_{2,2}x_2 & + \ldots & + & a_{2,j}x_j & + \ldots & + & a_{2,N}x_N & = & b_2 \\ \cdot & & \cdot & \cdot & & \cdot & \cdot & & \cdot & & \cdot \\ \cdot & & \cdot & \cdot & & \cdot & \cdot & & \cdot & & \cdot \\ \cdot & & \cdot & \cdot & & \cdot & \cdot & & \cdot & & \cdot \\ a_{N,1}x_1 & + & a_{N,2}x_2 & + \ldots & + & a_{N,j}x_j & + \ldots & + & a_{N,N}x_N & = & b_N \end{array} \qquad (1)$$

one may create the Jacobi Iteration formula:

$$x_j^{(k+1)} = \frac{b_j - \sum_{\substack{i=1 \\ i \neq j}}^{N} a_{j,i}\, x_i^{(k)}}{a_{j,j}} \qquad (2)$$

for j = 1,..N; where $x_j^{(k)}$ is the input value at *kth* iteration.

Like the Fixed-Point, there are also conditions that need to be satisfied in order to ensure that the said iteration will converge when the repetition count approaches to a considerably large value. Such conditions include the checking of whether or not the coefficient matrix A of the linear system is **strictly diagonally dominant**. As stated in Definition 3.8 on p. 185 of the textbook, A matrix **A** of dimension *N x N* is said to be strictly diagonally dominant provided that:

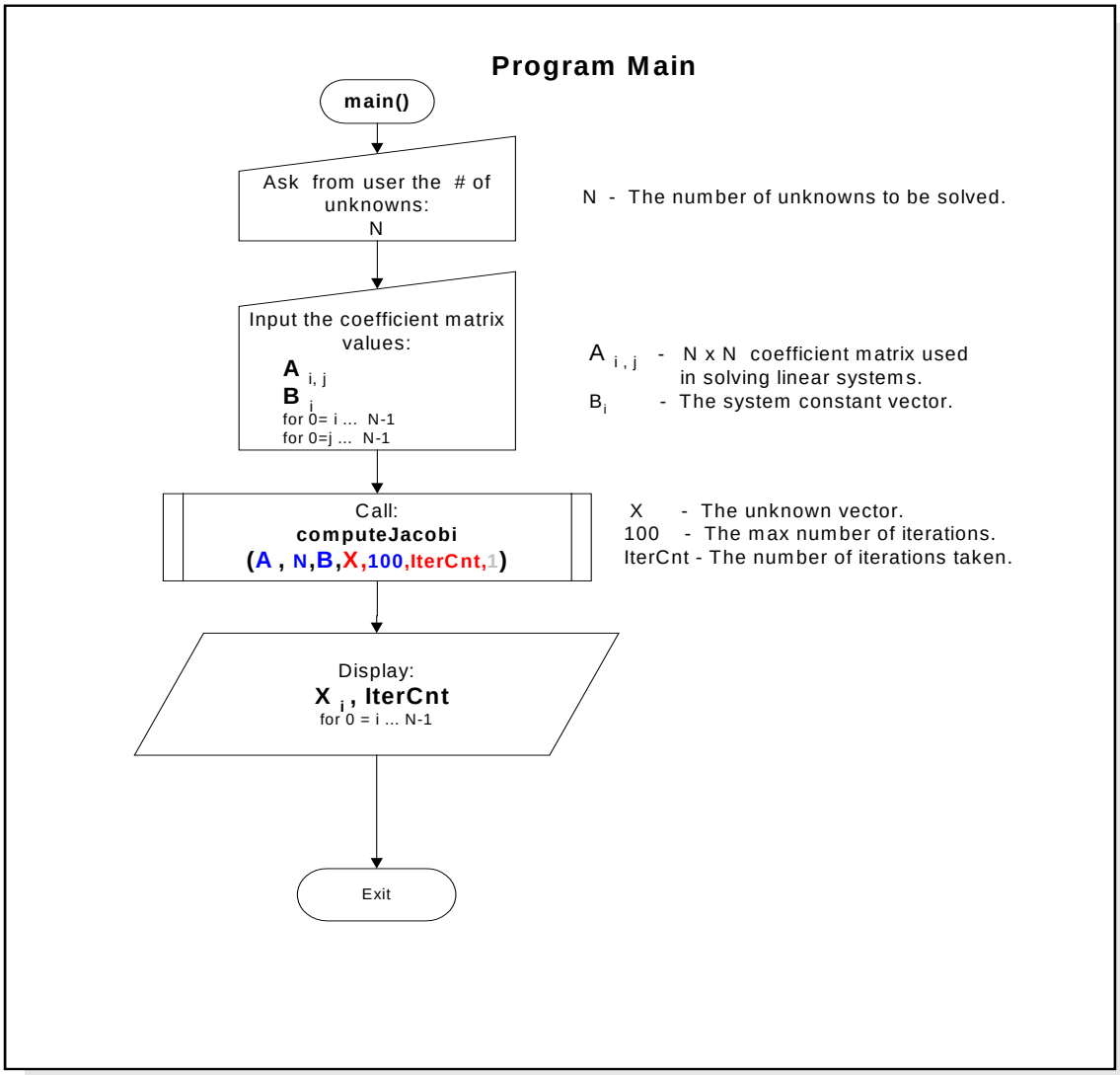
$$a_{j,j} > \sum_{\substack{i=1 \\ i \neq j}}^{N} a_{j,i} \qquad (3)$$

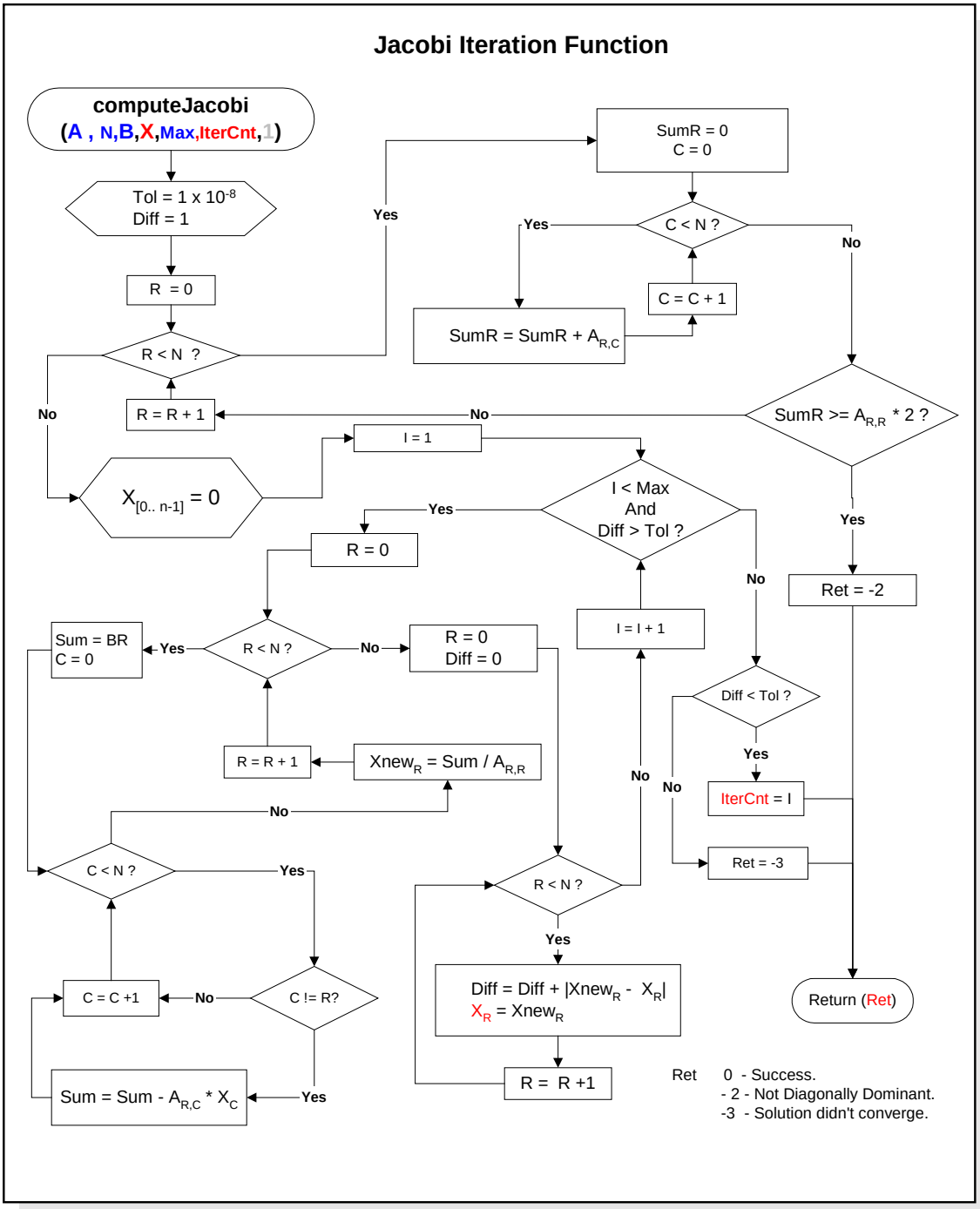
for j = 1,..N.

A detailed discussion on diagonal dominance is provided for on p. 184 of the textbook.

Algorithm 3.4 (Jacobi Iteration). To solve the linear system AX=B by starting with P_0 = 0 and generating a sequence $\{P_k\}$ the converges to the solution (i.e., AP = B). A sufficient method to be applicable is that A is diagonally dominant.

Program Flow




Jacobi Iteration Function
computeJacobi
(A , N,B,X,Max,IterCnt,1)
Tol = 1 x 10-8
Diff = 1
R = 0
R < N ?
Yes
No
R = R + 1
X[0.. n-1] = 0
I = 1
I < Max
And
Diff > Tol ?
Yes
No
R = 0
Sum = BR
C = 0
R < N ?
Yes
No
R = 0
Diff = 0
R = R + 1
Xnew_R = Sum / A_R,R
No
C < N ?
Yes
C = C +1
No
C != R?
Yes
Sum = Sum - A_R,C * X_C
SumR = 0
C = 0
C < N ?
Yes
No
C = C + 1
SumR = SumR + A_R,C
SumR >= A_R,R * 2 ?
No
Yes
Ret = -2
I = I + 1
Diff < Tol ?
Yes
No
IterCnt = I
Ret = -3
R < N ?
Yes
No
Diff = Diff + |Xnew_R - X_R|
X_R = Xnew_R
R = R +1
Return (Ret)
Ret 0 - Success.
- 2 - Not Diagonally Dominant.
-3 - Solution didn't converge.

Program Source Code

```
#include "JacobiFunc.h"
#include <conio.h>
void  printEquations(double* pMatrix,double* pInputB,int nNumUnknown);

int main(int argc, char* argv[])
{
    int    nIterCnt;
    double *pMatrix = NULL;
    double *pInputB = NULL;
    double *pOutputX = NULL;
    int    nNumUnknown;  int    i,j;
    int    nStatus = 0;  /* -1 - Mem error,  -2 - Singular matrix */
    printf("\n  ***  Jacobi Iteration *** \n");
    /*  This is the input stage */
    printf("\n Enter the number of unknowns : ");
    scanf("%d",&nNumUnknown); fflush(stdin);
    /* Allocate memory*/
    pMatrix = (double*) malloc(nNumUnknown * nNumUnknown * sizeof(pMatrix[0]));
    pInputB           = (double*) malloc(nNumUnknown * sizeof(pInputB[0]));
    pOutputX          = (double*) malloc(nNumUnknown * sizeof(pOutputX[0]));
    if(!pMatrix || !pOutputX || !pInputB)        nStatus = -1;
    if(nStatus == 0)
    {   /* Ask for the coefficients & constant*/
        for(i = 0; i < nNumUnknown; i++)
        {    /* Ask for the coefficients */
            for(j = 0; j < nNumUnknown; j++)
            {
                printf(" Eq. (%d): Enter the coefficient for variable %c: ",i+1,j + 'A');
                scanf("%lf",&MATRIXE(pMatrix,i,j,nNumUnknown)); fflush(stdin);
            }
            /* Ask for the constant */
            printf(" Eq. (%d): Enter the equation constant: ",i+1);
            scanf("%lf",&pInputB[i]); fflush(stdin);
        }
        /* print the equations */
        printEquations(pMatrix,pInputB,nNumUnknown);
    }
    /* Solve the system */
     if(nStatus == 0)
                nStatus =
computeJacobi(pMatrix,nNumUnknown,pInputB,pOutputX,100,&nIterCnt,1);
    /*  display output */
    if(nStatus == 0)
    {
        printf("\n\nResult(s): ");
        for(i = 0; i < nNumUnknown; i++)
            printf("\n %c = %lf ",i + 'A',pOutputX[i]);
        printf("\n\nNumber of Iterations(s): %d",nIterCnt);
    }
     /* Free dynamic allocations */
    if(pMatrix)  free(pMatrix);
    if(pInputB)  free(pInputB);
    if(pOutputX) free(pOutputX);
    switch(nStatus)
    {
    case -1:
        printf("\nMemory allocation error!!");
        break;
    case -2:
        printf("\nFailed: Not diagonally dominant!!");
        break;
```

```
case -3:
        printf("\nFailed: Solution didn't converge!!");
        break;
    default:
        break;
    }
    printf("\n\nPress any key to continue...");
    getch();
    return nStatus;
}
```

```
#include "JacobiFunc.h"
/* Function name       : computeJacobi
* Description         : Solves linear systems using the Jacobi iteration.
* Parameter(s)        :
*     *pMatrix        [in]  The coeffecient n x n Matrix.
*     nSize           [in]  The dimension of matrix.
*     pInputB         [in]  The constants.
*     pOuputX         [out] The unknown vectors.
*     nMaxIter        [in]  The maximum number of iteration.
*     pnActualIter    [out] The actual iteration taken until the system is solved.
*     nDispIter        [in]  If non-0 the vectors values in each iteration will be
displayed.
*
* Return              :
*    int
*              0   - Success,  -1   - Memory allocation error.
*             -2   - Not diagonally dominant,  -3   - Solution didn't converge
*/
int  __cdecl
computeJacobi(double  *pMatrix,int nSize,double* pInputB,double* pOuputX,int nMaxIter,int*
pnActualIter,int nDispIter)
{
    int nRet = 0;
    double *pOuputXNew = NULL;
    double  nTol  = 10e-8;    /* The tolerance convergence */
    double  nDiff = 1;        /* The summation of difference between
                                 the old values and new values */
    int     nIdx;
    *pnActualIter = nMaxIter;
    /* Check of the diagonal domimance */
    for(nIdx = 0; nIdx < nSize; nIdx++)
    {
        double nSumRow = 0;
        int  nCol;
        for(nCol = 0; nCol < nSize; nCol++)
            nSumRow += fabs(MATRIXE(pMatrix,nIdx,nCol,nSize));
        if(nSumRow >= fabs(MATRIXE(pMatrix,nIdx,nIdx,nSize)) * 2)
        {
            /* Not diagonally dominant */
            nRet = -2;
            break;
        }
    }
    if(nRet == 0)
    {   /* Allocate memory for new output values */
        pOuputXNew = (double*) malloc(nSize * sizeof(double));
        if(!pOuputXNew)
            nRet = -1;  /* Memory allocation error */
    }
```

```
if(nRet == 0)
{
    int  nIterCtr = 1;
    /* Initialize the output */
    for(nIdx = 0; nIdx < nSize; nIdx++)
    {
        pOuputX[nIdx] = 0;
        /* Display iteration values */
        if(nDispIter)
        {
            if(nIdx)
                printf(" %2.7lf ",pOuputX[nIdx]);
            else
                        {
                int i;
                printf("\n   N   ");
                for(i = 0; i < nSize; i++)
                   printf("  X(%02d)    ",i);
                printf("\n (%02d) ",0);
                printf(" %2.7lf ",pOuputX[nIdx]);
            }
        }
    }

    /* Start the iteration */
    while(nIterCtr < nMaxIter &&  nDiff > nTol)
    {
        for(nIdx = 0; nIdx < nSize; nIdx++)
        {
            double nSumRow;
            int    nCol;
            nSumRow = pInputB[nIdx];
            for(nCol = 0; nCol < nSize; nCol++)
            {
                if(nCol != nIdx)
                    nSumRow -= (MATRIXE(pMatrix,nIdx,nCol,nSize) * pOuputX[nCol]);
            }
            /* Compute the new value */
            pOuputXNew[nIdx] = nSumRow/MATRIXE(pMatrix,nIdx,nIdx,nSize);
        }
        nDiff = 0;
        /* Calculate the new iteration difference
           and assign the new values */

        for(nIdx = 0; nIdx < nSize; nIdx++)
        {
            /* compute the difference */
            nDiff += fabs(pOuputXNew[nIdx] - pOuputX[nIdx]);
            pOuputX[nIdx] = pOuputXNew[nIdx];

            if(nDispIter)
            {
                if(nIdx)
                    printf(" %2.7lf ",pOuputX[nIdx]);
                else
                {
                    printf("\n (%02d) ",nIterCtr);
                    printf(" %2.7lf ",pOuputX[nIdx]);
                }
            }
        }
```

```
            nIterCtr++;
        }

        if(nDiff < nTol)
            *pnActualIter = nIterCtr;
        else
            nRet = -3;
    }

    /* Free the allocation */
    if(pOuputXNew)    free(pOuputXNew);

    return nRet;
}
```

Program Output

```
 ***  Jacobi Iteration ***

Enter the number of unknowns : 3
Eq. (1): Enter the coefficient for variable A: 4
Eq. (1): Enter the coefficient for variable B: -1
Eq. (1): Enter the coefficient for variable C: 1
Eq. (1): Enter the equation constant: 7
Eq. (2): Enter the coefficient for variable A: 4
Eq. (2): Enter the coefficient for variable B: -8
Eq. (2): Enter the coefficient for variable C: 1
Eq. (2): Enter the equation constant: -21
Eq. (3): Enter the coefficient for variable A: -2
Eq. (3): Enter the coefficient for variable B: 1
Eq. (3): Enter the coefficient for variable C: 5
Eq. (3): Enter the equation constant: 15

Equations:
-------------------------------------------------
Eq. (1): 4.000000 A - 1.000000 B + 1.000000 C = 7.000000
Eq. (2): 4.000000 A - 8.000000 B + 1.000000 C = -21.000000
Eq. (3): -2.000000 A + 1.000000 B + 5.000000 C = 15.000000

-------------------------------------------------

  N     X(00)      X(01)      X(02)
(00)  0.0000000  0.0000000  0.0000000
(01)  1.7500000  2.6250000  3.0000000
(02)  1.6562500  3.8750000  3.1750000
(03)  1.9250000  3.8500000  2.8875000
(04)  1.9906250  3.9484375  3.0000000
(05)  1.9871094  3.9953125  3.0065625
(06)  1.9971875  3.9943750  2.9957812
(07)  1.9996484  3.9980664  3.0000000
(08)  1.9995166  3.9998242  3.0002461
(09)  1.9998945  3.9997891  2.9998418
(10)  1.9999868  3.9999275  3.0000000
(11)  1.9999819  3.9999934  3.0000092
(12)  1.9999960  3.9999921  2.9999941
(13)  1.9999995  3.9999973  3.0000000
(14)  1.9999993  3.9999998  3.0000003
(15)  1.9999999  3.9999997  2.9999998
(16)  2.0000000  3.9999999  3.0000000
(17)  2.0000000  4.0000000  3.0000000
(18)  2.0000000  4.0000000  3.0000000

Result(s):
 A = 2.000000
 B = 4.000000
 C = 3.000000

Number of Iterations(s): 19

Press any key to continue...
```

Gauss-Seidel Iteration

Discussion

As Seidel Iteration is the enhanced version of Fixed-Point in solving for nonlinear systems, so is **Gauss-Seidel Iteration** the enhanced version of the Jacobi Iteration in calculating for linear systems. Everything that is discussed about Jacobi Iteration is also true and applicable for **Gauss-Seidel Iteration** except for the Gauss-Seidel Iteration formula:

$$x_j^{(k+1)} = \frac{b_j - \sum_{i=1}^{j-1} a_{j,i} x_i^{(k+1)} - \sum_{i=j+1}^{N} a_{j,i} x_i^{(k)}}{a_{j,j}} \qquad (1)$$

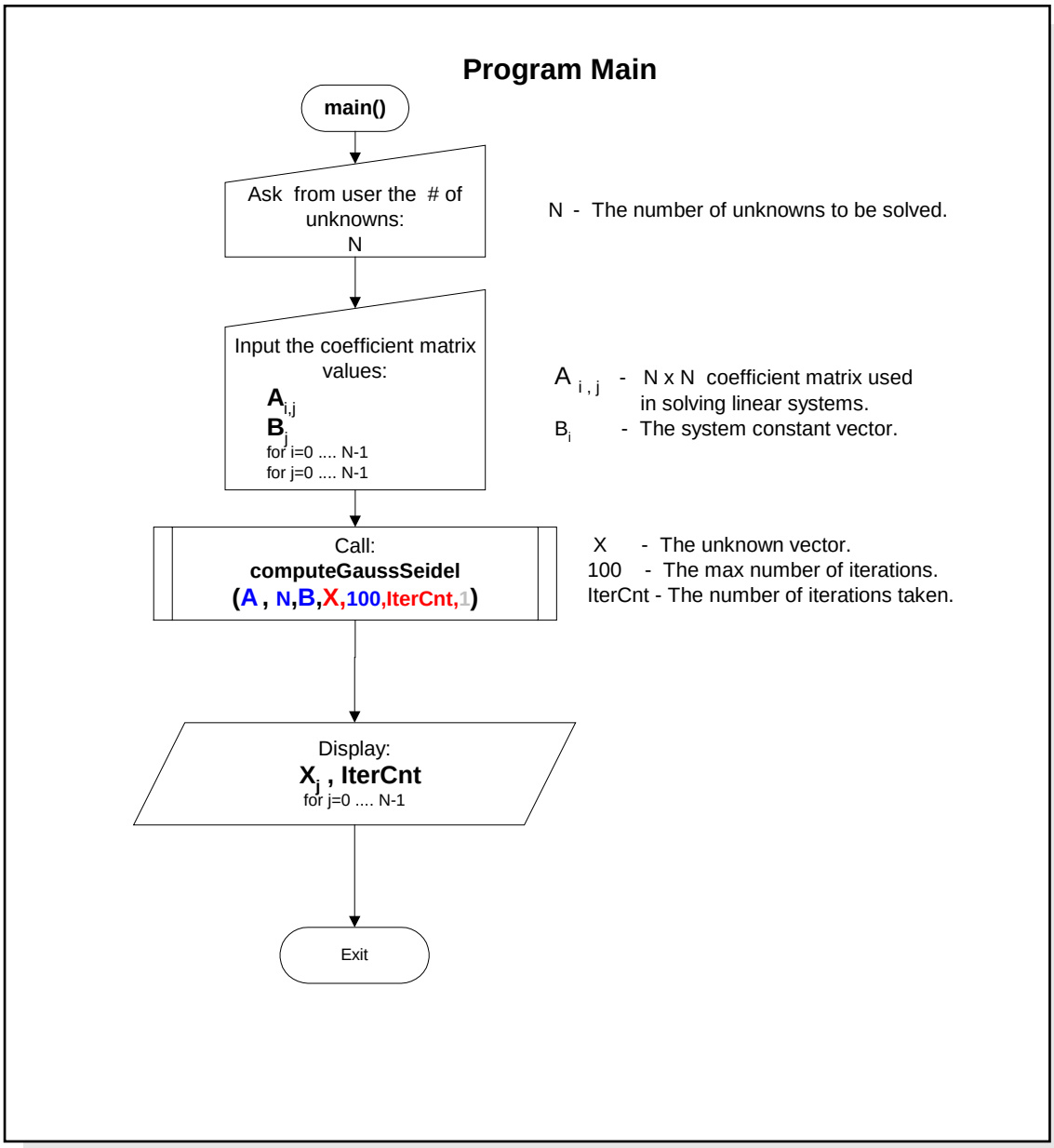
for j = 1,..N; where $x_j^{(k+1)}$ is the input value at *kth* iteration when i < j.

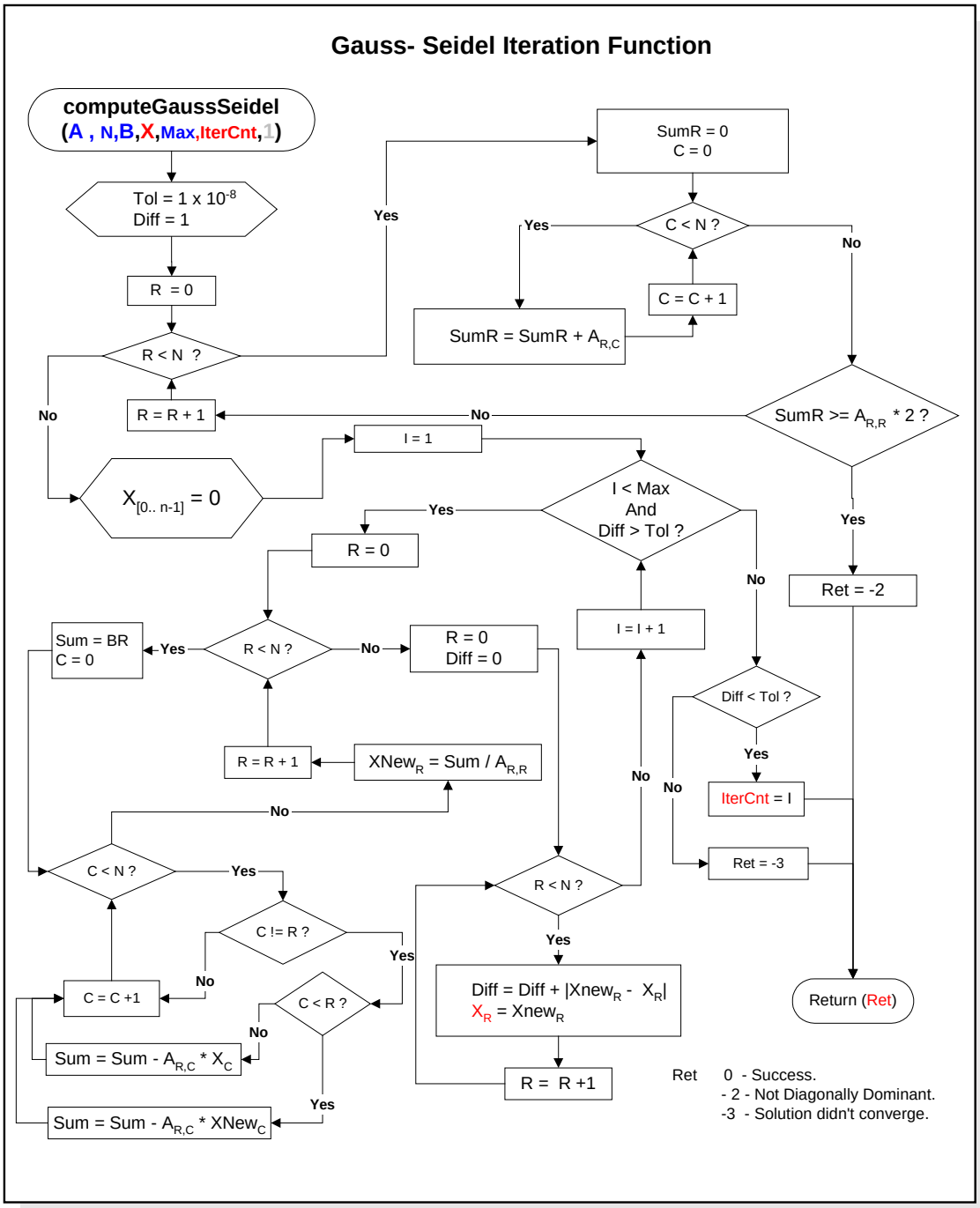
$x_j^{(k)}$ is the input value at *kth* iteration when i > j.

since, (like the Seidel iteration for nonlinear systems), uses the new coordinates or values as inputs as soon as they are available. This evidently makes the iteration converge faster than the previous method.

Algorithm 3.5 (Gauss-Seidel Iteration). To solve the linear system AX=B by starting with $P_0 = 0$ and generating a sequence $\{P_k\}$ the converges to the solution (i.e., AP = B). A sufficient method to be applicable is that A is diagonally dominant.

Program Flow




Gauss- Seidel Iteration Function
computeGaussSeidel
(A , N,B,X,Max,IterCnt,1)
Tol = 1 x 10-8
Diff = 1
R = 0
R < N ?
Yes
No
SumR = 0
C = 0
C < N ?
Yes
No
SumR = SumR + AR,C
C = C + 1
R = R + 1
No
SumR >= AR,R * 2 ?
Yes
X[0.. n-1] = 0
I = 1
I < Max
And
Diff > Tol ?
Yes
No
R = 0
Ret = -2
I = I + 1
Sum = BR
C = 0
R < N ?
Yes
No
R = 0
Diff = 0
Diff < Tol ?
Yes
No
R = R + 1
XNewR = Sum / AR,R
IterCnt = I
No
Ret = -3
C < N ?
Yes
R < N ?
Yes
C != R ?
Yes
No
C = C +1
C < R ?
No
Yes
Diff = Diff + |XnewR - XR|
XR = XnewR
Return (Ret)
Sum = Sum - AR,C * XC
R = R +1
Sum = Sum - AR,C * XNewC
Ret 0 - Success.
- 2 - Not Diagonally Dominant.
-3 - Solution didn't converge.

Program Source Code

```
#include "GaussSeidelFunc.h"
#include <conio.h>
void  printEquations(double* pMatrix,double* pInputB,int nNumUnknown);

int main(int argc, char* argv[])
{
    int    nIterCnt;
    double *pMatrix = NULL;
    double *pInputB = NULL;
    double *pOutputX = NULL;
    int    nNumUnknown;     int    i,j;
    int    nStatus = 0;  /* -1 - Mem error,  -2 - Singular matrix */
    printf("\n  ***  Gauss-Seidel Iteration *** \n");
    /*  This is the input stage */
    printf("\n Enter the number of unknowns : ");
    scanf("%d",&nNumUnknown);   fflush(stdin);
    /* Allocate memory  */
    pMatrix  = (double*) malloc(nNumUnknown * nNumUnknown * sizeof(pMatrix[0]));
    pInputB  = (double*) malloc(nNumUnknown * sizeof(pInputB[0]));
    pOutputX = (double*) malloc(nNumUnknown * sizeof(pOutputX[0]));
    if(!pMatrix || !pOutputX || !pInputB)   nStatus = -1;
    if(nStatus == 0)
    {   /* Ask for the coefficients & constant*/
        for(i = 0; i < nNumUnknown; i++)
        {
            /* Ask for the coefficients */
            for(j = 0; j < nNumUnknown; j++)
            {
                printf(" Eq. (%d): Enter the coefficient for variable %c: ",i+1,j + 'A');
                scanf("%lf",&MATRIXE(pMatrix,i,j,nNumUnknown)); fflush(stdin);
            }
            /* Ask for the constant */
            printf(" Eq. (%d): Enter the equation constant: ",i+1);
            scanf("%lf",&pInputB[i]);   fflush(stdin);
        }
        /* print the equations */
        printEquations(pMatrix,pInputB,nNumUnknown);
    }
    if(nStatus == 0) /* Solve the system */
        nStatus =
computeGaussSeidel(pMatrix,nNumUnknown,pInputB,pOutputX,100,&nIterCnt,1);
    /*  This is the display output stage */
    if(nStatus == 0)
    {
        printf("\n\nResult(s): ");
        for(i = 0; i < nNumUnknown; i++)  printf("\n %c = %lf ",i + 'A',pOutputX[i]);
        printf("\n\nNumber of Iterations(s): %d",nIterCnt);
    }
    /* Free dynamic allocations */
    if(pMatrix)   free(pMatrix);
    if(pInputB)   free(pInputB);
    if(pOutputX)  free(pOutputX);
```

```
    switch(nStatus)
    {
    case -1:
        printf("\nMemory allocation error!!");
        break;
    case -2:
        printf("\nFailed: Not diagonally dominant!!");
        break;
    case -3:
        printf("\nFailed: Solution didn't converge!!");
        break;
    default:
        break;
    }
    printf("\n\nPress any key to continue...");
    getch();
    return nStatus;
}
```

```
#include "GaussSeidelFunc.h"

/* Function name       : computeGaussSeidel
* Description         : Solves linear systems using the Gauss - Seidel iteration.
* Parameter(s)        :
*     *pMatrix         [in]  The coeffecient n x n Matrix.
*     nSize            [in]  The dimension of matrix.
*     pInputB          [in]  The constants.
*     pOuputX          [out] The unknown vectors.
*     nMaxIter         [in]  The maximum number of iteration.
*     pnActualIter     [out] The actual iteration taken until the system is solved.
*     nDispIter        [in]  If non-0 the vectors values in each iteration will be
displayed.
*
* Return              :
*    int
*              0   - Success, -1   - Memory allocation error,
*             -2   - Not diagonally dominant, -3   - Solution didn't converge
*/
int  __cdecl
computeGaussSeidel(double  *pMatrix,int nSize,double* pInputB,double* pOuputX,int
nMaxIter,int* pnActualIter,int nDispIter)
{
    int nRet = 0;
    double *pOuputXNew = NULL;
    double  nTol  = 10e-8;    /* The tolerance convergence */
    double  nDiff = 1;        /* The summation of difference between
                                 the old values and new values */
    int     nIdx;
    *pnActualIter = nMaxIter;
    /* Check of the diagonal domimance */
    for(nIdx = 0; nIdx < nSize; nIdx++)
    {
        double nSumRow = 0;
        int  nCol;
        for(nCol = 0; nCol < nSize; nCol++)
            nSumRow += fabs(MATRIXE(pMatrix,nIdx,nCol,nSize));
        if(nSumRow >= fabs(MATRIXE(pMatrix,nIdx,nIdx,nSize)) * 2)
        {   /* Not diagonally dominant */
            nRet = -2;
            break;
        }
    }
    if(nRet == 0)
    {   /* Allocate memory for new output values */
        pOuputXNew = (double*) malloc(nSize * sizeof(double));
        if(!pOuputXNew)  nRet = -1;
    }

    if(nRet == 0)
    {
        int  nIterCtr = 1;
        /* Initialize the output */
        for(nIdx = 0; nIdx < nSize; nIdx++)
        {
```

```
            pOuputX[nIdx] = 0;
            /* Display iteration values */
            if(nDispIter)
            {
                if(nIdx)
                    printf(" %2.7lf ",pOuputX[nIdx]);
                else
                {
                    int i;
                    printf("\n   N   ");
                    for(i = 0; i < nSize; i++)  printf("  X(%02d)    ",i);
                    printf("\n (%02d) ",0);
                    printf(" %2.7lf ",pOuputX[nIdx]);
                }
            }
        }
        /* Start the iteration */
        while(nIterCtr < nMaxIter &&  nDiff > nTol)
        {
            for(nIdx = 0; nIdx < nSize; nIdx++)
            {
                double nSumRow;
                int    nCol;
                nSumRow = pInputB[nIdx];
                for(nCol = 0; nCol < nSize; nCol++)
                {
                    if(nCol != nIdx)
                    {
                        double  nEffectiveP;
                        /* Use the new points its has already been computed. */
                        nEffectiveP = nCol < nIdx ?  pOuputXNew[nCol] : pOuputX[nCol];
                        nSumRow -= (MATRIXE(pMatrix,nIdx,nCol,nSize) * nEffectiveP);
                    }
                }
                /* Compute the new value */
                pOuputXNew[nIdx] = nSumRow/MATRIXE(pMatrix,nIdx,nIdx,nSize);
            }
            nDiff = 0;
            /* Calculate the new iteration difference and assign the new values */
            for(nIdx = 0; nIdx < nSize; nIdx++)
            {
                /* compute the difference */
                nDiff += fabs(pOuputXNew[nIdx] - pOuputX[nIdx]);
                pOuputX[nIdx] = pOuputXNew[nIdx];
                /* Display iteration values */
                if(nDispIter)
                {
                    if(nIdx)
                        printf(" %2.7lf ",pOuputX[nIdx]);
                    else
                    {
                        printf("\n (%02d) ",nIterCtr);
                        printf(" %2.7lf ",pOuputX[nIdx]);
                    }
                }
            }
            nIterCtr++;
        }
```

```
        if(nDiff < nTol)
            *pnActualIter = nIterCtr;
        else
            nRet = -3;
    }

   /* Free the allocation */
    if(pOuputXNew)     free(pOuputXNew);
    return nRet;
}
```

Program Output

```
  ***  Gauss-Seidel Iteration ***

 Enter the number of unknowns : 3
 Eq. (1): Enter the coefficient for variable A: 4
 Eq. (1): Enter the coefficient for variable B: -1
 Eq. (1): Enter the coefficient for variable C: 1
 Eq. (1): Enter the equation constant: 7
 Eq. (2): Enter the coefficient for variable A: 4
 Eq. (2): Enter the coefficient for variable B: -8
 Eq. (2): Enter the coefficient for variable C: 1
 Eq. (2): Enter the equation constant: -21
 Eq. (3): Enter the coefficient for variable A: -2
 Eq. (3): Enter the coefficient for variable B: 1
 Eq. (3): Enter the coefficient for variable C: 5
 Eq. (3): Enter the equation constant: 15

 Equations:
 -------------------------------------------------
Eq. (1): 4.000000 A - 1.000000 B + 1.000000 C = 7.000000
Eq. (2): 4.000000 A - 8.000000 B + 1.000000 C = -21.000000
Eq. (3): -2.000000 A + 1.000000 B + 5.000000 C = 15.000000

 -------------------------------------------------

   N      X(00)       X(01)       X(02)
 (00)  0.0000000   0.0000000   0.0000000
 (01)  1.7500000   3.5000000   3.0000000
 (02)  1.8750000   3.9375000   2.9625000
 (03)  1.9937500   3.9921875   2.9990625
 (04)  1.9982813   3.9990234   2.9995078
 (05)  1.9998789   3.9998779   2.9999760
 (06)  1.9999755   3.9999847   2.9999932
 (07)  1.9999979   3.9999981   2.9999995
 (08)  1.9999996   3.9999998   2.9999999
 (09)  2.0000000   4.0000000   3.0000000
 (10)  2.0000000   4.0000000   3.0000000

Result(s):
 A = 2.000000
 B = 4.000000
 C = 3.000000

Number of Iterations(s): 11

Press any key to continue...
```

Lagrange Approximation

Discussion

Basically, computers operate on 1's and 0's called binary digits no matter how complicated the software maybe. Behind the scenes of evaluating functions such as sin(x), cos(x) and/or e^x is polynomial approximation. Given sample points which maybe obtained from a function analytically, one may use interpolation in order to estimate the missing points based on the values of the existing ones. One of the simplest methods for doing this is the linear approximation, (can be easily done by ratio and proportion), which uses a line segment that passes through two points.

$$y = P(x) = y_0 + (y_1 - y_0)\frac{x - x_0}{x_1 - x_0} \quad .(1)$$

Expanding (1) would result to a polynomial with degree ≤ 1. And when evaluated at x_0 and x_1, the resulting values will be y_0 and y_1 respectively:

$$\begin{aligned} P(x_0) &= y_0 + (y_1 - y_0)0 = y_0 \\ P(x_1) &= y_0 + (y_1 - y_0)1 = y_1. \end{aligned} \quad (2)$$

A French mathematician Joseph Louis Lagrange uses another way of writing the polynomial by:

$$y = P_1(x) = y_0 \frac{x - x_1}{x_0 - x_1} + y_1 \frac{x - x_0}{x_1 - x_0} \quad (3)$$

which can also be used in approximating polynomials with degree > 1 provided that sufficient input data points exist. Proofs are illustrated on pp. 215-217 of the textbook.

The generalized form of (3) for polynomials with degree N that passes thru N + 1 input points is:

$$P_N(x) = \sum_{k=0}^{N} y_k L_{N,k}(x) \quad (4)$$

where $\boldsymbol{L_{N,k}(x)}$ is the Lagrange coefficient polynomial based on the input data points.

$$L_{N,k}(x) = \frac{\prod_{\substack{j=0 \\ j \neq k}}^{N} (x - x_k)}{\prod_{\substack{j=0 \\ j \neq k}}^{N} (x_k - x_j)} \tag{5}$$

so that (3) can be written into:

$$y = P_1(x) = y_0 L_{1,0}(x) + y_1 L_{1,1}(x) \tag{6}$$

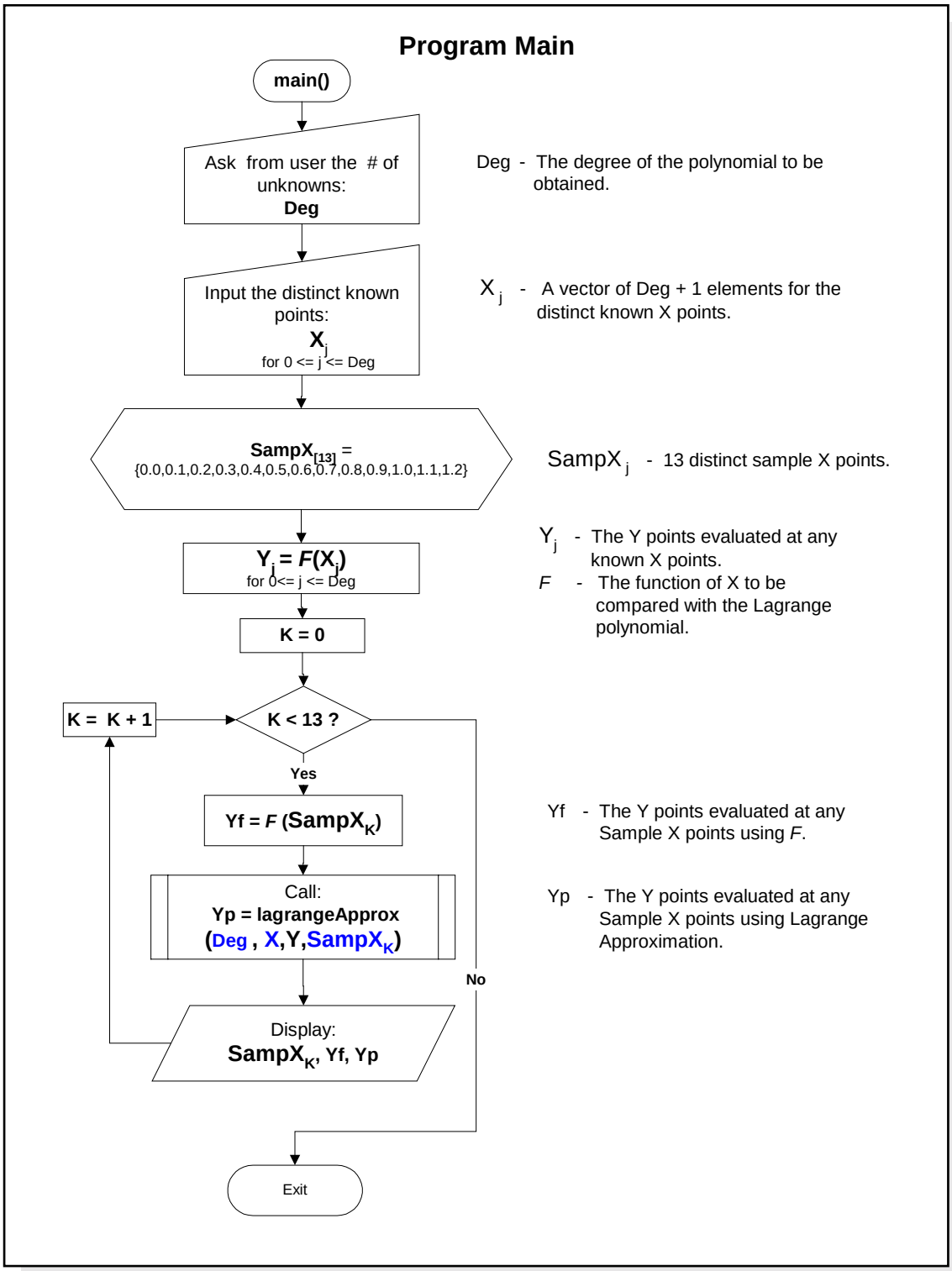
Since the above-mentioned method is just a polynomial approximation of the given function, one should take note of the error terms and error bounds of (4) which has the form:

$$E_N(x) = \frac{(x-x_0)(x-x_1)...(x-x_N) f^{(N+1)}(c)}{(N+1)!} \tag{7}$$

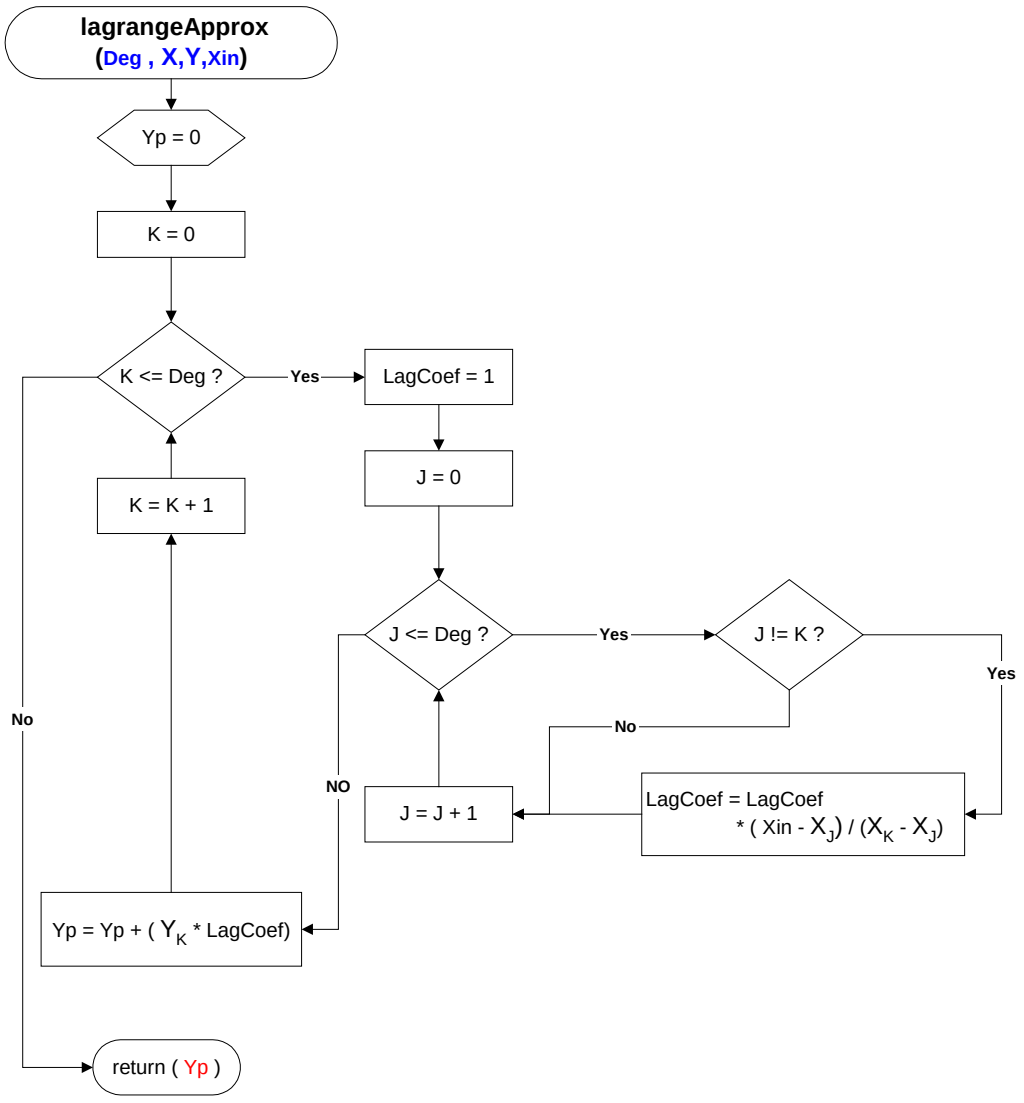
has similar form of the truncation error of a Taylor polynomial, except that the factor $(x-x_0)^{N+1}$ is replaced with the product $(x-x_0)(x-x_1)...(x-x_N)$. Details on error terms and error bounds are discussed on pp. 219-220 of the textbook.

Algorithm 4.3 (Lagrange Approximation). To evaluate the Lagrange polynomial $P(x) = \sum_{k=0}^{N} y_k L_{N,k}(x)$, of degree N, based on the N+1 points (x_k, y_k) for k = 0,1 ... N.

Program Flow



Lagrange Approximation Function



Yp - contains the Lagrange approximation.

Program Source Code

```
#include "stdafx.h"
#include <math.h>
#include <conio.h>
#include "LgrngApx.h"
double   functionOfX(double fx);
int main(int argc, char* argv[])
{
	/* This will be the sample input and sample output */
	const double nInputXValues[]   =
{0.0,0.1,0.2,0.3,0.4,0.5,0.6,0.7,0.8,0.9,1.0,1.1,1.2};
	/* Lets makes the degree variable which may  range from 1 - 12     */
	double nXNodes[sizeof(nInputXValues)/sizeof(nInputXValues[0])];
	double nYNodes[sizeof(nInputXValues)/sizeof(nInputXValues[0])];
	int    nDegree;
	int    i,k;
	int    nSampleCount = sizeof(nInputXValues)/sizeof(nInputXValues[0]);
	printf("\n *** Lagrange Approximation program *** \n");
	printf("\n\n f(x) = sin(x) \n");
	/*  This is the input stage */
	do{
		printf("\n Enter the polynomial degree [1 - %d]: ",nSampleCount-1);
		scanf("%d",&nDegree); fflush(stdin);
	}while(nDegree <= 0 || nDegree > (nSampleCount-1));

	printf("\n");
	for(i = 0; i <= nDegree; i++)
	{
		int j;
		int distinct = 1;
		do{
		printf(" Enter a distinct known point X%d (must be within
[%2.2lf,%2.2lf]): "
			,i,nInputXValues[0],nInputXValues[nSampleCount - 1]);
		scanf("%lf",&nXNodes[i]);
		/* Check if distinct */
		for(j = 0; j < i; j++)
		{
			if(nXNodes[i] == nXNodes[j])
			{
				distinct = 0;
				break;
			}
		}
		fflush(stdin);

		}while(nXNodes[i] < nInputXValues[0]
			|| nXNodes[i] > nInputXValues[nSampleCount - 1] || distinct == 0);
		/* Compute the y value using f(x) */
		nYNodes[i]     = functionOfX(nXNodes[i]);
	}

	/*  This is computation/output stage  */
	/* ---------------------------------------------- */
	printf("\n\n                  ***   Result  ***              \n\n");
	printf("\n-------------------------------------------------------------\n");
	printf("\n   Xk       f(xk)      Pn(xk)    | f(xk) - Pn(xk) | ");
	for(k = 0; k < nSampleCount; k++)
	{
		double  Fx;
		double  Px;
```

```
                Fx = functionOfX(nInputXValues[k]);
                Px = lagrangeApprox(nDegree,nXNodes,nYNodes,nInputXValues[k]);
                printf("\n %2.6lf   %2.6lf  %2.6lf  %2.6lf",nInputXValues[k],Fx,Px,fabs(Fx-
Px));
        }
        printf("\n-------------------------------------------------------------\n");
        printf("\npress any key to continue..");
        /* remove all inputs from the buffer */
        while(kbhit());
        getch();
        return 0;
}
/* This will be the function of X which is currently equal to sin(x). */
double
functionOfX(double x)
{       return sin(x); }
```

```
#include "LgrngApx.h"
/* Function name      : lagrangeApprox
* Description        : This is used to get the approximated value of
*                      Y for a given input X.
* Parameter(s)       :
*     nDegree        [in]  The degree of the polynomial.
*     xNodes[]       [in]  The node array containing the known
*                          points of X.
*                          (The size of this array must be nDegree + 1).
*     yNodes[]        [in]  The node array containing Y = f(x) values
*                           corresponding each nodes in xNodes.
*                          (The size of this array must be nDegree + 1).
*     nInputX        [in]  The independent variable X inorder to obtain
*                          Y = Pn(x).
* Return             :
*    double              Returns the computed Y = Pn(x).
*/
double
lagrangeApprox(int nDegree,double xNodes[],double yNodes[],double nInputX)
{
    double dblRet = 0;

    register int k;
    for(k = 0; k <= nDegree; k++)
    {
        /* This outer loop will be used to compute the summation
          of YkLn,k(x). 0 <= k <= Degree.
          ------------------------------------------------------
        */
        double nLagCoef = 1;  /* Since we will compute for the products
                              we will start for an initial value of 1 */
        register int j;
        for(j = 0; j <= nDegree; j++)
        {
            /* This inner loop will be used to compute the product
              of Lj,k(x). 0 <= j <= Degree.
              ------------------------------------------------------
            */
            if(j != k)   /* Skip the term when j == k */
            {
                /* Compute the product */
                nLagCoef *= ((nInputX - xNodes[j])/(xNodes[k] - xNodes[j]));
            }
        }
        dblRet += (yNodes[k] * nLagCoef);

        /*
          ------------------------------------------------------
        */
    }

    return dblRet;
}
```

Program Output

```
*** Lagrange Approximation program ***

f(x) = sin(x)

Enter the polynomial degree [1 - 12]: 4

Enter a distinct known point X0 (must be within [0.00,1.20]): 0
Enter a distinct known point X1 (must be within [0.00,1.20]): .4
Enter a distinct known point X2 (must be within [0.00,1.20]): .6
Enter a distinct known point X3 (must be within [0.00,1.20]): .8
Enter a distinct known point X4 (must be within [0.00,1.20]): 1

                  ***   Result   ***

------------------------------------------------------------

  Xk        f(xk)       Pn(xk)     ¦ f(xk) - Pn(xk) ¦
0.000000   0.000000  0.000000  0.000000
0.100000   0.099833  0.099764  0.000069
0.200000   0.198669  0.198614  0.000056
0.300000   0.295520  0.295498  0.000023
0.400000   0.389418  0.389418  0.000000
0.500000   0.479426  0.479431  0.000005
0.600000   0.564642  0.564642  0.000000
0.700000   0.644218  0.644213  0.000004
0.800000   0.717356  0.717356  0.000000
0.900000   0.783327  0.783336  0.000009
1.000000   0.841471  0.841471  0.000000
1.100000   0.891207  0.891132  0.000076
1.200000   0.932039  0.931741  0.000298
------------------------------------------------------------

press any key to continue..
```

Least-Squares Line

Discussion

Plotting a set of data points can always and easily be done as long as the function *f(x)* is available. But in actual situations such as scientific experiments, where most of the time the function *f(x)* is missing, one must determine the formula $y = f(x)$ that relates to an obtained set of experimental data points $\{x_k\}$ and $\{y_k\}$. The goal of Least-Squares Line is to find the best linear approximation of the form shown in (1) below that goes near these set of points.

$$y = f(x) = Ax + B. \tag{1}$$

Because the obtained set of data points is subject to error, then one must compensate these values in order to obtain the true value for $f(x_k)$ by:

$$f(x_k) = y_k + e_k \tag{2}$$

where e_k is the measurement error. And therefore

$$e_k = f(x_k) - y_k \qquad \text{for } 1 \le k \le N. \tag{3}$$

There are different norms which can be used to measure how far (1) is away from the data points. Below is the list of these norms and each is discussed in details on pp. 252-253 of the textbook.

Maximum Error : $$E_\infty(f) = \max_{for\, 1 \le k \le N} \{|f(x_k) - y_k|\}, \tag{4}$$

Average Error : $$E_1(f) = \frac{1}{N}\sum_{k=1}^{N} |f(x_k) - y_k|, \tag{5}$$

Root - mean - square : $$E_2(f) = \left(\frac{1}{N}\sum_{k=1}^{N} |f(x_k) - y_k|^2 \right)^{1/2}. \tag{6}$$

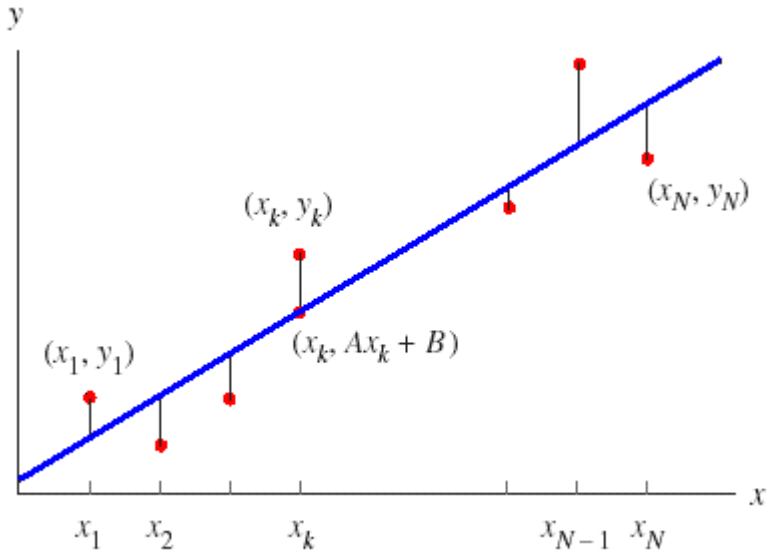


Figure 5.1.1 The vertical distances of points $\{(x_k, y_k)\}$ and the least-squares line y=Ax+B.

Above is a plot of data points of unequal weight showing just the range measured. The best linear approximation line y=Ax + B is displayed. Under the assumption that the errors have a normal distribution, the best linear approximation is obtained by minimizing the root-mean-square $E_2(f)$ in (6) from the corresponding *fitted* $\{y_k\}$. So in fitting a straight line (1) to N data points, one must minimize the sum:

$$E(A,B) = \frac{1}{N}\sum_{k=1}^{N}(Ax_k + B - y_k)^2 \tag{7}$$

then applying partial derivative with respect to A and B, whose details can be found in pp. 254-255 of the textbook, the solution to the following system, known as **normal equations** is:

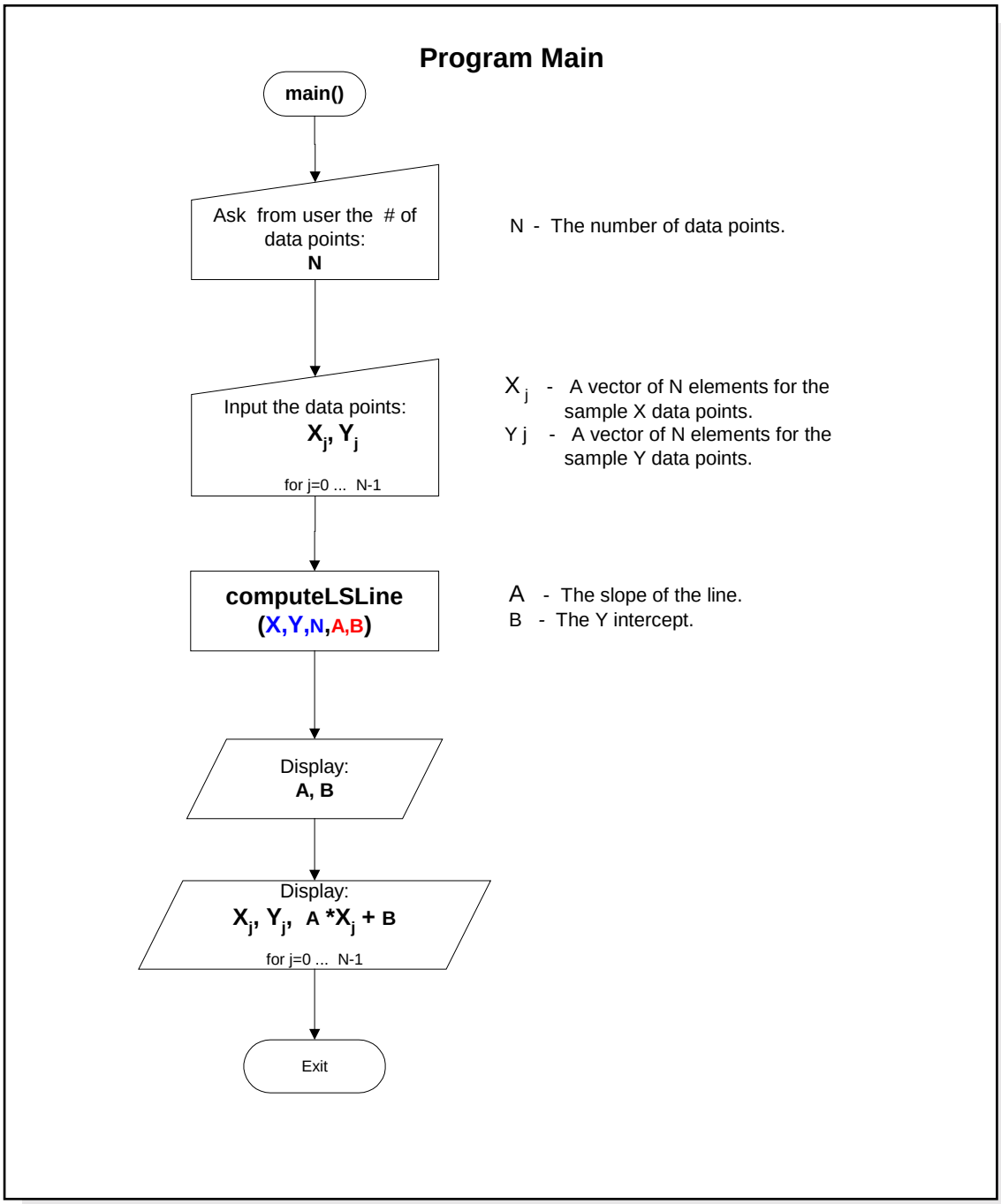
$$\begin{aligned} \left(\sum_{k=1}^{N} x_k^2\right)A \quad \left(\sum_{k=1}^{N} x_k\right)B &= \sum_{k=1}^{N} x_k y_k, \\ \left(\sum_{k=1}^{N} x_k\right)A \qquad NB &= \sum_{k=1}^{N} x_k. \end{aligned} \tag{8}$$

Obtaining the values of A and B from (8) can easily be solved using Cramer's rule or any solution to linear system.

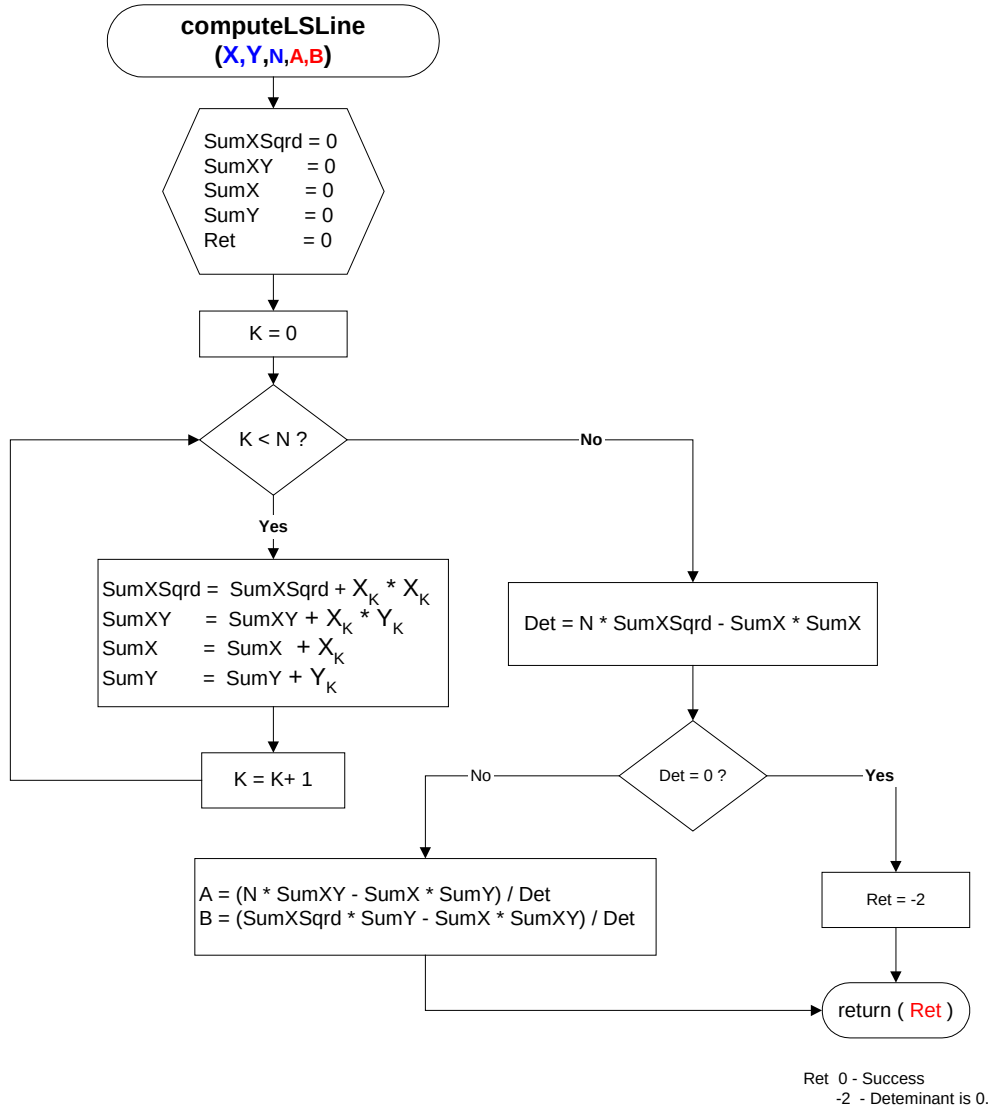
Algorithm 5.1 (Least-Squares Line). To construct the least-squares lines y= Ax + B that fits the N data points, (x_1, y_1) .... (x_N, y_N).

Remarks: The algorithm in the book doesn't really follow based on its discussion, so I modified it.

Program Flow



Least - Squares Line Function



Program Source Code

```
#include <stdio.h>
#include <malloc.h>
#include <conio.h>
#include "LSLineFunc.h"
int main(int argc, char* argv[])
{
    int     nStatus = 0;
    int     nNumPoints;
    double  *pnXPoints = NULL;
    double  *pnYPoints = NULL;
    double  nA;
    double  nB;
    printf("\n  ***  Least-Squares Line *** \n");
    /*  This is the input stage */
    printf("\n Enter the number of data points : ");
    scanf("%d",&nNumPoints);    fflush(stdin);
    /* Allocate memory */
    pnXPoints          = (double*) malloc(nNumPoints * sizeof(pnXPoints[0]));
    pnYPoints          = (double*) malloc(nNumPoints * sizeof(pnYPoints[0]));
    if(!pnXPoints || !pnYPoints)  nStatus = -1;
    if(nStatus == 0)
    {
        int i;
        /* Ask for the points */
        for(i = 0; i < nNumPoints; i++)
        {
            printf(" Enter the value for (X%d): ",i);
            scanf("%lf",&pnXPoints[i]);  fflush(stdin);
            printf(" Enter the value for (Y%d): ",i);
            scanf("%lf",&pnYPoints[i]);  fflush(stdin);
        }
    }
    /*  This is the computation stage */
    if(nStatus == 0)
    {
        nStatus = computeLSLine(pnXPoints,pnYPoints,nNumPoints,&nA,&nB);
        if(nStatus)  nStatus -= 2;
    }

    /*  Display */
    if(nStatus == 0)
    {
        int i;
        printf("\n The least-squares line: y = %2.8lf x + %2.8lf ",nA,nB);
        printf("\n\n  --------------------------- ");
        printf("\n\n N    \t     Xk     \t     Yk    \t     f(Xk) \n");
        for(i = 0; i < nNumPoints; i++)
            printf("\n %d  \t %2.8lf \t %2.8lf \t %2.8lf"
                   ,i,pnXPoints[i],pnYPoints[i], (nA * pnXPoints[i] + nB));
        printf("\n\n  --------------------------- ");
    }
    switch(nStatus)
    {
    case -1:
        printf("\nMemory allocation error!!");
        break;
```

```
    case -2:
        printf("\nFailed: Insuffient data points!!");
        break;
    case -3:
        printf("\nFailed: Determinant is 0!!");
        break;
    default:
        break;
    }
    printf("\n\nPress any key to continue...");
    /* Free the Points */
    if(pnXPoints)  free(pnXPoints);
    if(pnYPoints)  free(pnYPoints);
    getch();
    return nStatus;
}
```

```
#include "LSLineFunc.h"
/* Function name       : computeLSLine
* Description         : Computes the coefficients A and B for a Least Square Line.
* Parameter(s)        :
*     *pXPoints         [in]  The X coordinate vector.
*     *pYPoints         [in]  The Y coordinate vector corresponding to each point in X.
*     nNumPoints        [in]  The number of elements or data points.
*     pA                [out] The A coefficient.
*     pB                [out] The B coefficient.
* Return              :
*    int _cdecl       0  - Success, -1  - Insufficient data points.
*/
int _cdecl
computeLSLine(double *pXPoints,double *pYPoints,int nNumPoints,double* pA,double* pB)
{
    int nRet = -1;
    double    nSumXSqrd;  /*  Sum (X^2) */
    double    nSumXY;     /*  Sum (XY)  */
    double    nSumX;      /*  Sum (X)   */
    double    nSumY;      /*  Sum (Y)   */
    double    nDet;       /*  The determinant   */
    int       i;
    *pA = 0;
    *pB = 0;
    /* Must require atleast two points */
    if(nNumPoints > 1)
    {
        nSumXSqrd = 0;
        nSumXY    = 0;
        nSumX     = 0;
        nSumY     = 0;
        /* Find the summation values */
        for(i = 0; i < nNumPoints; i ++)
        {
            nSumXSqrd += pXPoints[i] * pXPoints[i];  /* Sum up all the x^2 */
            nSumXY    += pXPoints[i] * pYPoints[i];  /* Sum up all the xy  */
            nSumX     += pXPoints[i];                /* Sum up all the x  */
            nSumY     += pYPoints[i];                /* Sum up all the y  */
        }
        /* Get the value for A using cramer's rule */
        nDet = nNumPoints * nSumXSqrd - nSumX * nSumX;
        if(nDet == 0)  nRet = -2;
        else           nRet = 0;
    }

    if(nRet == 0)
    {
        *pA = (nNumPoints * nSumXY - nSumX * nSumY) / nDet;
        *pB = (nSumXSqrd * nSumY - nSumX * nSumXY) / nDet;

        nRet = 0;
    }

    return nRet;
}
```

Program Output

```
  ***  Least-Squares Line ***

 Enter the number of data points : 8
 Enter the value for (X0): -1
 Enter the value for (Y0): 10
 Enter the value for (X1): 0
 Enter the value for (Y1): 9
 Enter the value for (X2): 1
 Enter the value for (Y2): 7
 Enter the value for (X3): 2
 Enter the value for (Y3): 5
 Enter the value for (X4): 3
 Enter the value for (Y4): 4
 Enter the value for (X5): 4
 Enter the value for (Y5): 3
 Enter the value for (X6): 5
 Enter the value for (Y6): 0
 Enter the value for (X7): 6
 Enter the value for (Y7): -1

 The least-squares line: y = -1.60714286 x + 8.64285714

  ----------------------------

 N           Xk              Yk              f(Xk)

 0       -1.00000000     10.00000000     10.25000000
 1       0.00000000      9.00000000      8.64285714
 2       1.00000000      7.00000000      7.03571429
 3       2.00000000      5.00000000      5.42857143
 4       3.00000000      4.00000000      3.82142857
 5       4.00000000      3.00000000      2.21428571
 6       5.00000000      0.00000000      0.60714286
 7       6.00000000      -1.00000000     -1.00000000

  ----------------------------

Press any key to continue...
```

Least-Squares Polynomial

Discussion

There are also times that the curve, which represents the data points, has a higher degree than that of a line. If that would be the case, then one may fit them into a polynomial of degree M:

$$f(x) = c_1 + c_2 x + c_3 x^2 + \ldots + c_{M+1} x^M . \tag{1}$$

The example used in the textbook is a polynomial of degree M=2, with N number of data points $\{(x_k, y_k)\}_{k=1}^{N}$ whose abscissas are distinct, showing a least-squares parabola:

$$y = f(x) = Ax^2 + Bx + C. \tag{2}$$

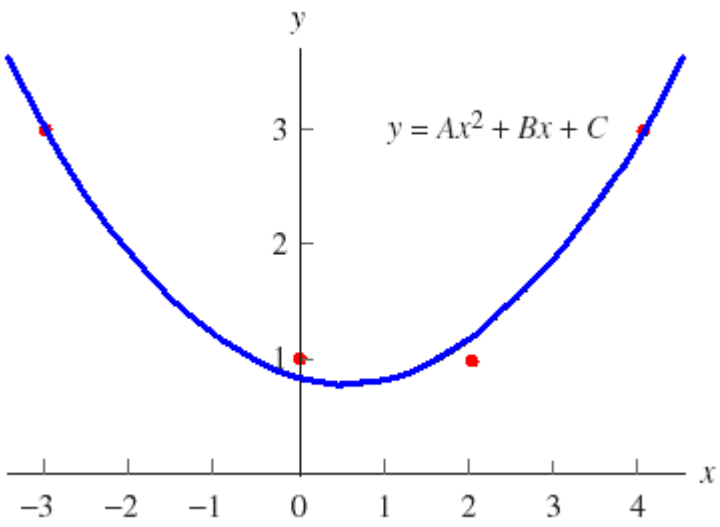


Figure 5.2.1 The least-squares parabola.

Just like in least-squares line, one must minimize the sum:

$$E(A, B, C) = \frac{1}{N} \sum_{k=1}^{N} (Ax_k^2 + Bx_k + C - y_k)^2 . \tag{3}$$

Then applying partial derivative with respect to A, B and C, whose details can be found in p. 271 of the textbook, will now form a linear system:

$$\begin{aligned}
\left(\sum_{k=1}^{N} x_k^4\right)A \quad \left(\sum_{k=1}^{N} x_k^3\right)B \quad \left(\sum_{k=1}^{N} x_k^2\right)C &= \sum_{k=1}^{N} x_k y_k^2, \\
\left(\sum_{k=1}^{N} x_k^3\right)A \quad \left(\sum_{k=1}^{N} x_k^2\right)B \quad \left(\sum_{k=1}^{N} x_k\right)C &= \sum_{k=1}^{N} x_k y_k, \\
\left(\sum_{k=1}^{N} x_k^2\right)A \quad \left(\sum_{k=1}^{N} x_k\right)B \quad NC &= \sum_{k=1}^{N} x_k.
\end{aligned} \tag{4}$$

Obtaining the values for A, B and C from (4) can easily be solved by employing any of the solutions to linear systems.

To generalize (4) from the polynomial of degree M in (1), the linear system for solving the polynomial coefficients will be:

$$\begin{aligned}
\left(\sum_{k=1}^{N} x_k^{2M}\right)c_{m+1} \quad \left(\sum_{k=1}^{N} x_k^{2M-1}\right)c_m \quad \ldots \quad \left(\sum_{k=1}^{N} x_k^{M}\right)c_1 &= \sum_{k=1}^{N} x_k y_k^{M}, \\
\left(\sum_{k=1}^{N} x_k^{2M-1}\right)c_{m+1} \quad \left(\sum_{k=1}^{N} x_k^{2M-2}\right)c_m \quad \ldots \quad \left(\sum_{k=1}^{N} x_k^{M-1}\right)c_1 &= \sum_{k=1}^{N} x_k y_k^{M-1}, \\
\vdots \qquad\qquad \vdots \qquad\qquad \vdots \qquad\qquad \vdots \qquad\qquad \vdots & \\
\left(\sum_{k=1}^{N} x_k^{M}\right)c_{m+1} \quad \left(\sum_{k=1}^{N} x_k^{M-1}\right)c_m \quad \ldots \quad Nc_1 &= \sum_{k=1}^{N} x_k.
\end{aligned}$$

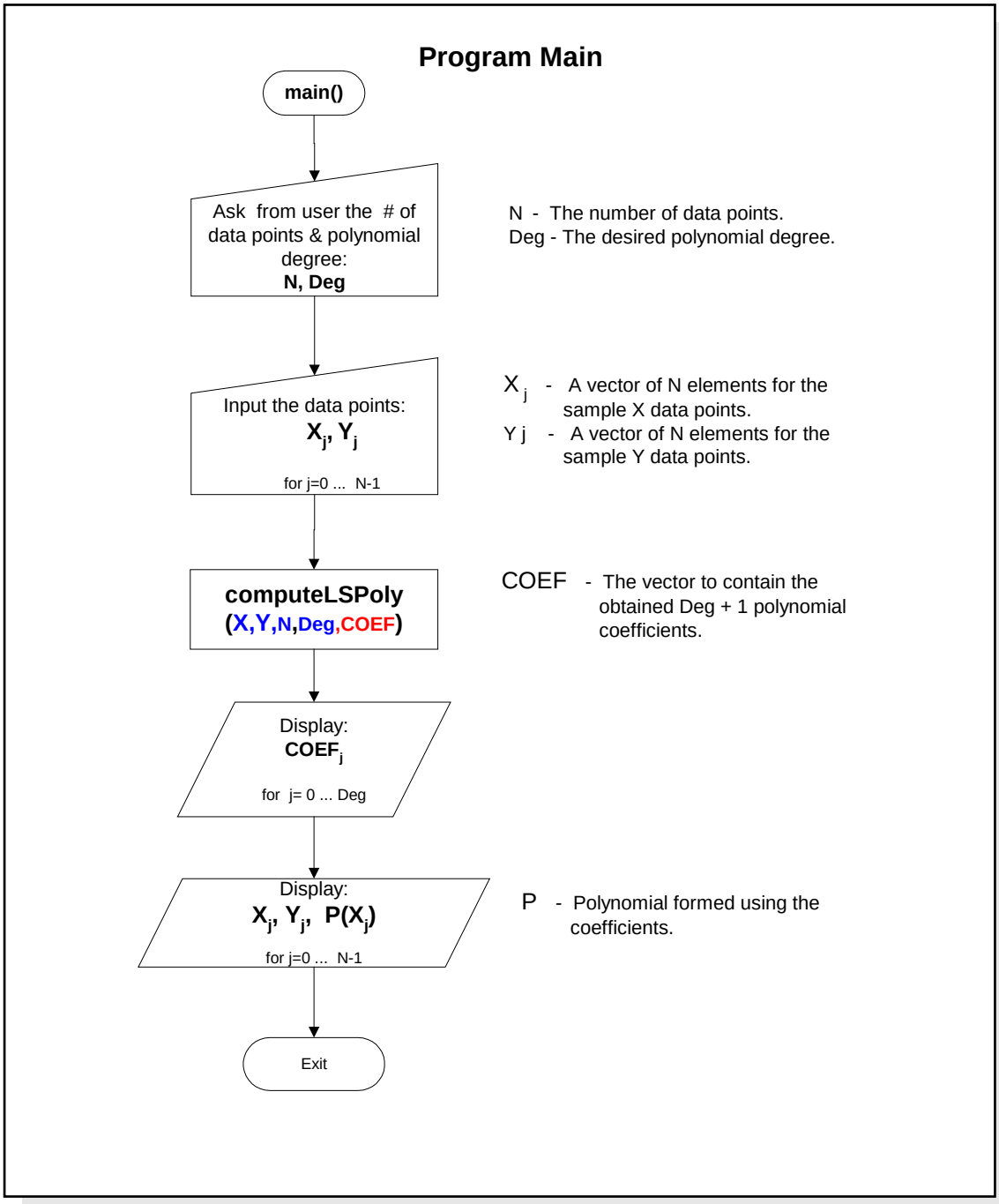
If the data to be fit doesn't really represent a polynomial nature, then a phenomenon called "polynomial wiggle" will surely occur.

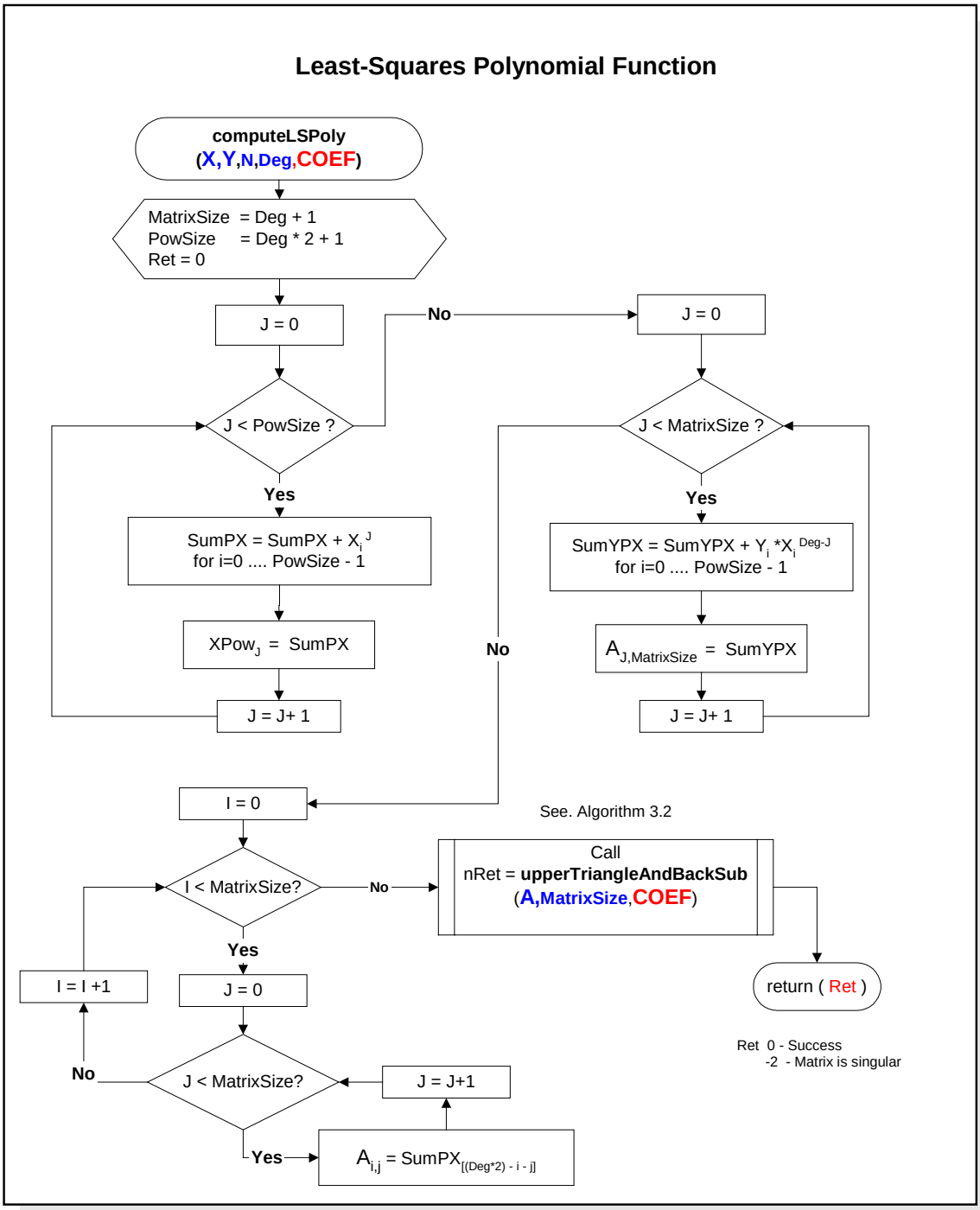
Algorithm 5.2 (Least-Squares Polynomial). To construct the least-squares polynomial of degree M of the form

$$P_M(x) = c_1 + c_2 x + c_3 x^2 + \ldots + c_{M+1} x^M$$

that fits the N data points, (x_1, y_1) .... (x_N, y_N).

Program Flow




Least-Squares Polynomial Function
computeLSPoly
(X,Y,N,Deg,COEF)
MatrixSize = Deg + 1
PowSize = Deg * 2 + 1
Ret = 0
J = 0
J < PowSize ?
Yes
No
SumPX = SumPX + X_i^J
for i=0 .... PowSize - 1
XPow_J = SumPX
J = J+ 1
J = 0
J < MatrixSize ?
Yes
SumYPX = SumYPX + Y_i *X_i^Deg-J
for i=0 .... PowSize - 1
A_J,MatrixSize = SumYPX
J = J+ 1
No
I = 0
See. Algorithm 3.2
I < MatrixSize?
No
Call
nRet = upperTriangleAndBackSub
(A,MatrixSize,COEF)
return (Ret)
Yes
I = I +1
J = 0
No
J < MatrixSize?
J = J+1
Yes
A_i,j = SumPX_[(Deg*2) - i - j]
Ret 0 - Success
-2 - Matrix is singular

Program Source Code

```
#include <stdio.h>
#include <conio.h>
#include "LSPolyFunc.h"

int main(int argc, char* argv[])
{
    int     nStatus = 0;
    int     nNumPoints,nPolyDegree;
    double  *pnXPoints = NULL, *pnYPoints = NULL, *pnCoefOutput = NULL;

    printf("\n  ***  Least-Squares Polynomial *** \n");
    /*  This is the input stage */
    printf("\n Enter the number of data points : ");
    scanf("%d",&nNumPoints);   fflush(stdin);
    printf("\n Enter the polymonial degree     :");
    scanf("%d",&nPolyDegree);  fflush(stdin);
    /* Allocate memory */
    pnXPoints          = (double*) malloc(nNumPoints * sizeof(pnXPoints[0]));
    pnYPoints          = (double*) malloc(nNumPoints * sizeof(pnYPoints[0]));
    pnCoefOutput       = (double*) malloc((nPolyDegree + 1) * sizeof(pnCoefOutput[0]));

    if(!pnXPoints || !pnYPoints || !pnCoefOutput)   nStatus = -1;
    if(nStatus == 0)
    {
        int i;
        /* Ask for the points */
        for(i = 0; i < nNumPoints; i++)
        {
            printf(" Enter the value for (X%d): ",i);
            scanf("%lf",&pnXPoints[i]); fflush(stdin);
            printf(" Enter the value for (Y%d): ",i);
            scanf("%lf",&pnYPoints[i]); fflush(stdin);
        }
    }

    /*  This is the computation stage */
    if(nStatus == 0)
    {
        nStatus = computeLSPoly(pnXPoints,pnYPoints,nNumPoints,nPolyDegree,pnCoefOutput);
        if(nStatus)  nStatus -= 2;
    }
    /*  Display */
    if(nStatus == 0)
    {
        int i,j,nPrint = 0;
        printf("\n The least-squares polynomial: \n P%d = ",nPolyDegree);
        for(j = 0; j <=  nPolyDegree; j++)
        {
            if(pnCoefOutput[j])
            {
                printf(" %c %2.8lf (x^%d)",(nPrint ? '+':' '),pnCoefOutput[j],nPolyDegree
- j);
                nPrint++;
            }
        }
        printf("\n\n  ---------------------------- ");
        printf("\n\n N    \t     Xk     \t     Yk    \t     f(Xk) \n");
        for(i = 0; i < nNumPoints; i++)
        {
```

```
            double dblFx = 0;
            for(j = 0; j <=  nPolyDegree; j++)
                dblFx += (pnCoefOutput[j] * pow(pnXPoints[i],(double)nPolyDegree - j));
            printf("\n %d  \t %2.8lf \t %2.8lf \t
%2.8lf",i,pnXPoints[i],pnYPoints[i],dblFx);
        }
        printf("\n\n  ---------------------------- ");
    }

    switch(nStatus)
    {
    case -1:
        printf("\nMemory allocation error!!");
        break;
    case -2:
        printf("\nFailed: Insuffient data points!!");
        break;
    case -3:
        printf("\nFailed: Determinant is 0!!");
        break;
    default:
        break;
    }
    printf("\n\nPress any key to continue...");

    /* Free the Y Points */
    if(pnXPoints)  free(pnXPoints);
    if(pnYPoints)  free(pnYPoints);
    if(pnCoefOutput)  free(pnCoefOutput);
    getch();
    return nStatus;
}
```

```
#include "LSPolyFunc.h"
#include "UpperTriangle.h"
/* Function name       : computeLSPoly
* Description         : Computes the coefficients A and B for a Least-Squares Polynomial.
* Parameter(s)        :
*     *pXPoints         [in]  The X coordinate vector.
*     *pYPoints         [in]  The Y coordinate vector corresponding to each point in X.
*     nNumPoints        [in]  The number of elements or data points.
*     nNumPoints        [in]  The degree of the polynomial to be obtain.
*     pCoefficients     [out] The coefficient vector output.
* Return              :
*    int _cdecl       0  - Success, -1  - Insufficient memory.
*                    -2  - Insufficient data points, -3  - Singular matrix.
*/
int _cdecl
computeLSPoly(double *pXPoints,double *pYPoints,int nNumPoints,int nDegree,double*
pCoefficients)
{
    int nRet = 0;
    int       nMatrixSize;
    double    *pAugmentedMatrix = NULL; /* The augmented matrix for the X^n and the
constants */
    double    *pSumXPwrN        = NULL; /* The vetor of the summation of X^n */
    /* Must require atleast two points */
    if(nNumPoints <= 1)    nRet = -2;
    /* Allocate memory for the Augmented matrix and Summation of X of power N*/
    if(nRet == 0)
    {
              nMatrixSize = nDegree + 1;
        pAugmentedMatrix = (double*) malloc(nMatrixSize * (nMatrixSize + 1) *
sizeof(pAugmentedMatrix[0]));
        pSumXPwrN        = (double*) malloc(((nDegree * 2) + 1) * sizeof(pSumXPwrN[0]));
              if(!pAugmentedMatrix || !pSumXPwrN)     nRet = -1;
    }
    /* Compute the summation of powers */
    if(nRet == 0)
    {
        int i,j;
        pSumXPwrN[0] = nNumPoints;
        for(i = 1; i < ((nDegree * 2) + 1); i++)
        {
            pSumXPwrN[i] = 0;
            for(j = 0; j < nNumPoints; j++)
                pSumXPwrN[i] += pow(pXPoints[j],(double)i);
        }
    }
    /* Compute and assign the constant vectors */
    if(nRet == 0)
    {
        int i,j;
        /*  B[i] = Sum( Y * X^(Degree - i) ) */
        for(i = 0; i < nMatrixSize; i++)
        {
            MATRIXE(pAugmentedMatrix,i,nMatrixSize,(nMatrixSize + 1))    = 0;
            for(j = 0; j < nNumPoints; j++)
            {
                MATRIXE(pAugmentedMatrix,i,nMatrixSize,(nMatrixSize + 1))    +=
                    (pYPoints[j] * pow(pXPoints[j],(double)(nDegree - i)));
            }
        }
    }
```

```
    /* Assign the matrix  */
    if(nRet == 0)
    {
        int i,j;
        /*  A[i,j] = Sum(X^(Degree * 2) - i - j) ) */
        for(i = 0; i < nMatrixSize; i++)
        {
            for(j = 0; j < nMatrixSize; j++)
            {
                MATRIXE(pAugmentedMatrix,i,j,(nMatrixSize + 1))
                    = pSumXPwrN[(nDegree * 2) - i - j];
            }
        }
    }
    /*  Now compute the linear system  */
    if(nRet == 0)
    {
        nRet = upperTriangleAndBackSub(pAugmentedMatrix,nMatrixSize,pCoefficients);
        /* Adjust the error to our convention */
        if(nRet == -2)  nRet--;
    }

    /* Deallocate the memory */
    if(pAugmentedMatrix)  free(pAugmentedMatrix);
    if(pSumXPwrN)         free(pSumXPwrN);

    return nRet;
}
```

Program Output

```
  ***  Least-Squares Polynomial ***

 Enter the number of data points : 4

 Enter the polymonial degree     : 2
 Enter the value for (X0): -3
 Enter the value for (Y0): 3
 Enter the value for (X1): 0
 Enter the value for (Y1): 1
 Enter the value for (X2): 2
 Enter the value for (Y2): 1
 Enter the value for (X3): 4
 Enter the value for (Y3): 3

 The least-squares polynomial:
 P2 =    0.17846248 (x^2) + -0.19249542 (x^1) + 0.85051861 (x^0)

  ----------------------------

 N           Xk              Yk             f(Xk)

 0       -3.00000000     3.00000000     3.03416718
 1       0.00000000      1.00000000     0.85051861
 2       2.00000000      1.00000000     1.17937767
 3       4.00000000      3.00000000     2.93593655

  ----------------------------

 Press any key to continue...
```

Nonlinear Curve Fitting

Discussion

Aside from the least-squares line and least-squares polynomial, one may also fit the given set of data points into a nonlinear curve such as:

$$y = Ce^{Ax}. \tag{1}$$

Two ways are used to complete the task. First is the **data linearization method,** second is the **nonlinear least-squares method**. Data linearization involves the transformation of the original point (x_k, y_k) in the xy-plane into (X_k, Y_k) in the XY-plane thereby represented by a linear equation:

$$Y = AX + B. \tag{2}$$

Since the data points have been linearized, one may apply the least-squares lines method to the transformed data points. Finally, after obtaining the values of the linear coefficients, one may transform them back to their nonlinear equivalent. A detailed sample, using (1) as the nonlinear equation, can be found on pp 268-269 of the textbook.

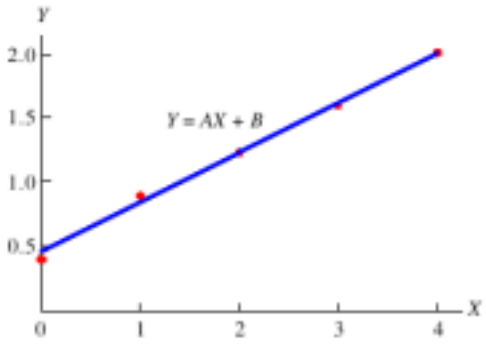


Figure 5.3.1 The transformed data points.

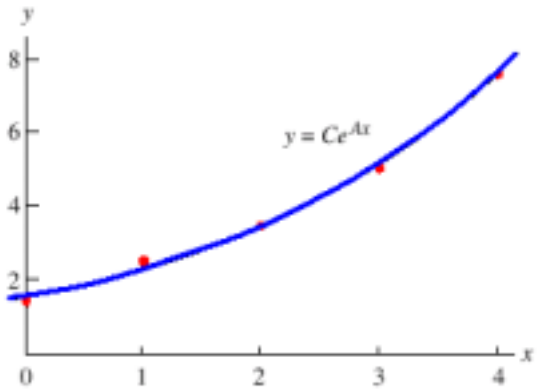


Figure 5.3.2 The exponential fit $y = Ce^{Ax}$ obtained after data linearization.

Nonlinear least-squares method on the other hand, solves the nonlinear curve as it is (details are shown on pp. 270-271 of the textbook) and uses the Newton's method in obtaining its coefficients. But since the solution in getting the coefficients needs a lot of time, requires good starting values for the iteration process, and when compared with the former, differs only for a small amount of value. It is more preferable to use the **data linearization** method.

The table below contains the list of nonlinear equations and its equivalent conversions for data linearization.

Function, $y=f(x)$	Linearized form, $Y=AX+B$	Change of variable(s) and constants
$y = \frac{A}{x} + B$	$y = A\frac{1}{x} + B$	$X = \frac{1}{x}, Y = y$
$y = \frac{D}{x + C}$	$y = \frac{-1}{C}(xy) + \frac{D}{C}$	$X = xy, Y = y$ $C = \frac{-1}{A}, D = \frac{-B}{A}$
$y = \frac{1}{Ax + B}$	$\frac{1}{y} = Ax + B$	$X = x, Y = \frac{1}{y}$
$y = \frac{x}{A + Bx}$	$\frac{1}{y} = A\frac{1}{x} + B$	$X = \frac{1}{x}, Y = \frac{1}{y}$
$y = A\ln(x) + B$	$y = A\ln(x) + B$	$X = \ln(x), Y = y$
$y = Ce^{Ax}$	$\ln(y) = Ax + \ln(C)$	$X = x, Y = \ln(y)$ $C = e^{B}$
$y = Cx^{A}$	$\ln(y) = A\ln(x) + \ln(C)$	$X = \ln(x), Y = \ln(y)$ $C = e^{B}$
$y = \frac{1}{(Ax + B)^2}$	$y^{-1/2} = Ax + B$	$X = x, Y = y^{-1/2}$

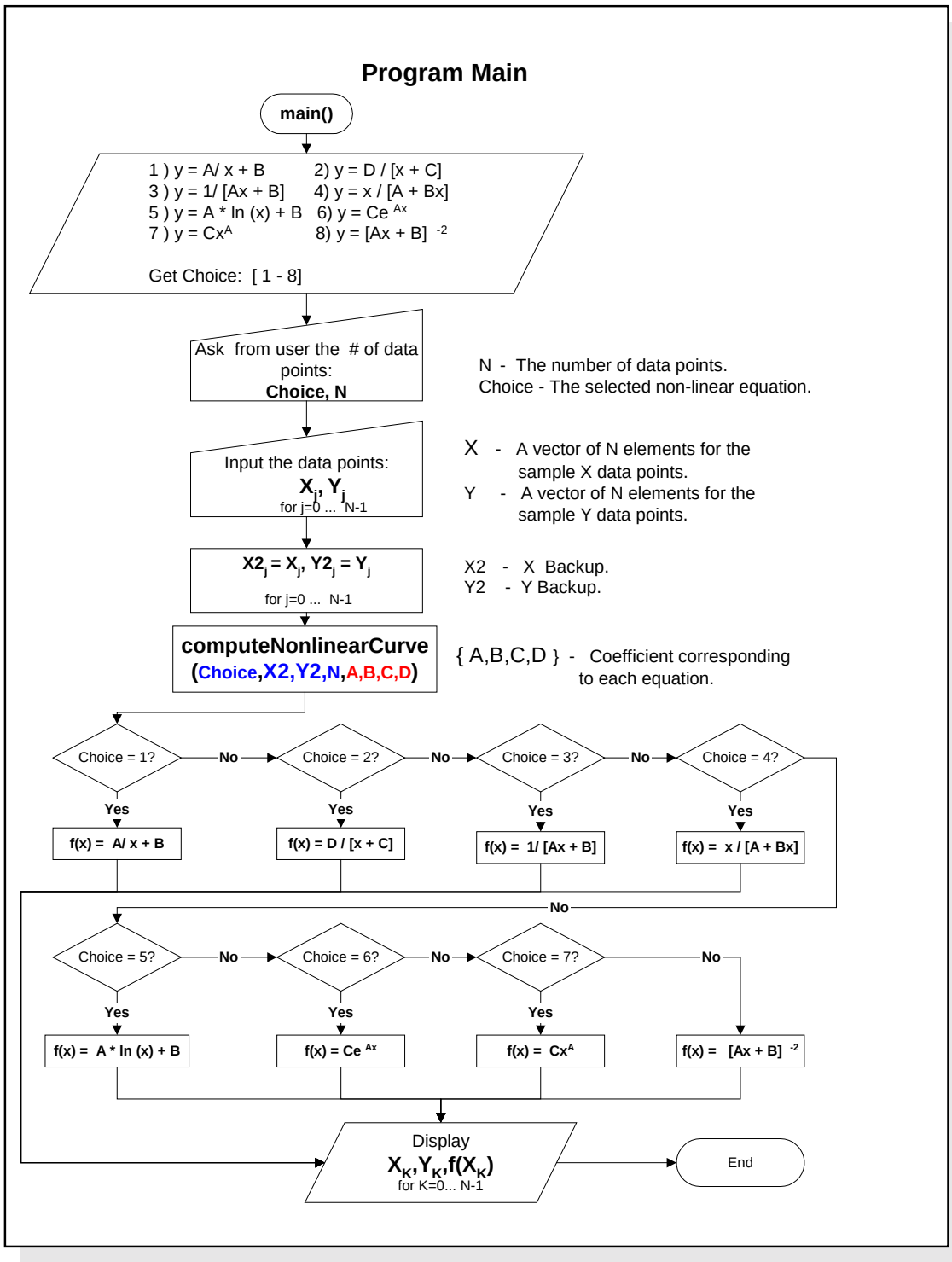
$y = Cx\,e^{-Dx}$	$\ln\left(\frac{y}{x}\right) = -Dx + \ln(C)$	$X = x, Y = \ln\left(\frac{y}{x}\right)$ $C = e^{B}, D = -A$
$y = \frac{L}{1 + Ce^{Ax}}$	$\ln\left(\frac{L}{y} - 1\right) = Ax + \ln(C)$	$X = x, Y = \ln\left(\frac{L}{y} - 1\right)$ $C = e^{B}$ and L is a constant that must be given.

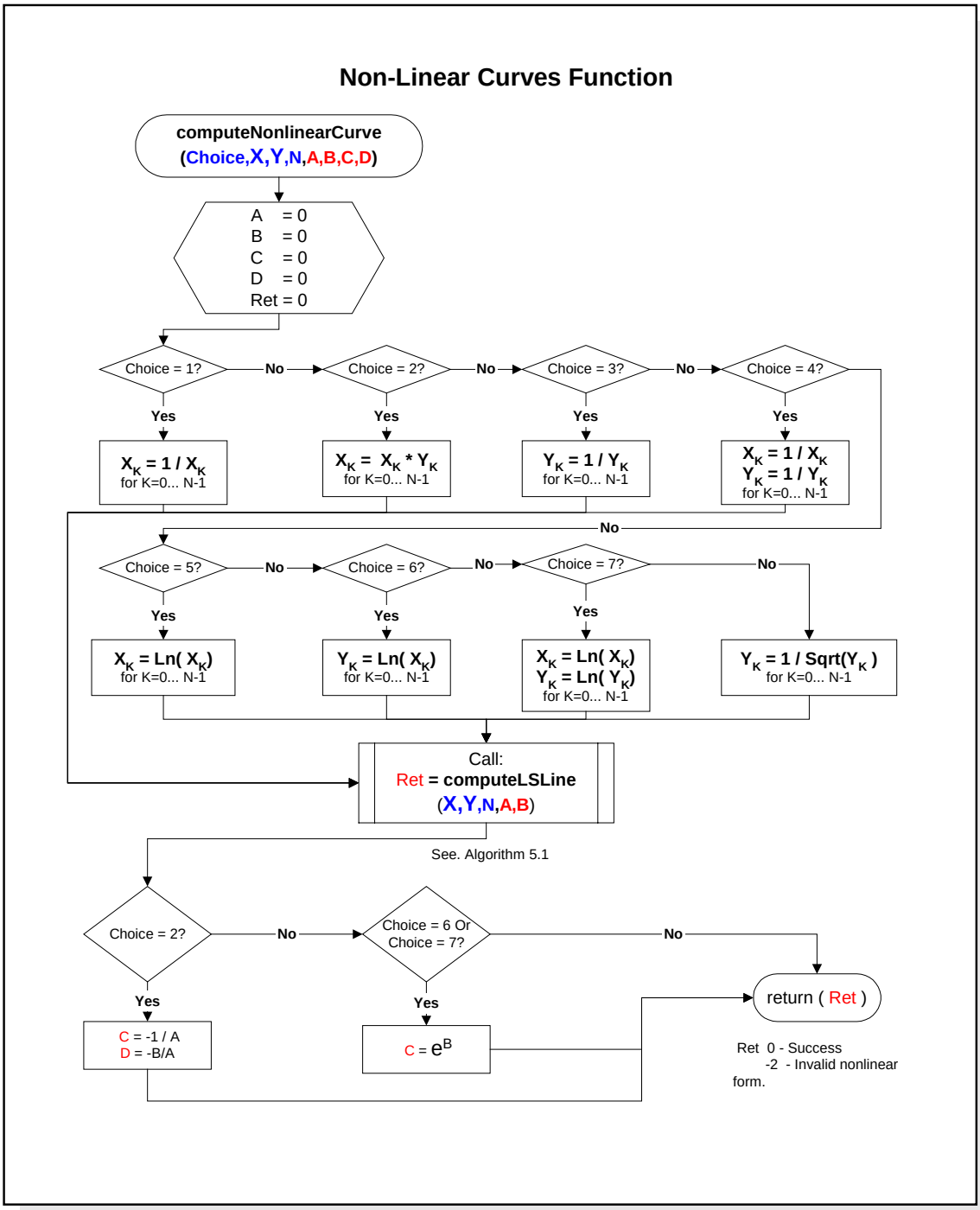
Figure 5.3.1 Change of Variable(s) for Data Linearization.

Algorithm 5.3 (Nonlinear Curve Fitting). To construct a nonlinear fit y=g(x) given the N data points, (x_1, y_1) .... (x_N, y_N) by using "data linearization."
Available curves are:

$$y = \frac{A}{x} + B, \qquad y = \frac{D}{x + C}, \quad y = \frac{1}{Ax + B}, \qquad y = \frac{x}{A + Bx},$$
$$y = A\ln(x) + B, \quad y = Ce^{Ax}, \qquad y = Cx^{A}, \qquad y = \frac{1}{(Ax + B)^2}.$$

Program Flow




Non-Linear Curves Function
computeNonlinearCurve
(Choice,X,Y,N,A,B,C,D)
A = 0
B = 0
C = 0
D = 0
Ret = 0
Choice = 1?
No
Choice = 2?
No
Choice = 3?
No
Choice = 4?
Yes
Yes
Yes
Yes
X_K = 1 / X_K
for K=0... N-1
X_K = X_K * Y_K
for K=0... N-1
Y_K = 1 / Y_K
for K=0... N-1
X_K = 1 / X_K
Y_K = 1 / Y_K
for K=0... N-1
No
Choice = 5?
No
Choice = 6?
No
Choice = 7?
No
Yes
Yes
Yes
X_K = Ln(X_K)
for K=0... N-1
Y_K = Ln(X_K)
for K=0... N-1
X_K = Ln(X_K)
Y_K = Ln(Y_K)
for K=0... N-1
Y_K = 1 / Sqrt(Y_K)
for K=0... N-1
Call:
Ret = computeLSLine
(X,Y,N,A,B)
See. Algorithm 5.1
Choice = 2?
No
Choice = 6 Or
Choice = 7?
No
Yes
Yes
return (Ret)
C = -1 / A
D = -B/A
C = e^B
Ret 0 - Success
-2 - Invalid nonlinear
form.

Program Source Code

```
#include  <conio.h>
#include  <math.h>
#include  <conio.h>
#include  <memory.h>
#include  <malloc.h>
#include "LSLineFunc.h"
#include "NonLinearFunc.h"

int main(int argc, char* argv[])
{
    int     nStatus = 0;
    int     nNumPoints;
    double  *pnXPoints = NULL;
    double  *pnYPoints = NULL;
    double  *pnXOrigPoints = NULL;
    double  *pnYOrigPoints = NULL;
    double  nA,nB,nC,nD;

    int nForm;
    char    cChoice;

    printf("\n  ***  Non-Linear Curve Fitting *** \n");
    /*  This is the input stage */
    do{
        nForm = -1;

        printf("\n Select the nonlinear form you wish to fit:\n");
        printf("\n ------------------------------------------------------");
        printf("\n\n 1 ) y = A/ x + B          2) y = D / [x + C]     ");
        printf("\n\n 3 ) y = 1/ [Ax + B]       4) y = x / [A + Bx]    ");
        printf("\n\n 5 ) y = A * ln (x) + B    6) y = C * exp(Ax)     ");
        printf("\n\n 7 ) y = C * x^(A)         8) y = [Ax + B] ^ (-2) ");
        printf("\n\n ------------------------------------------------------\n");

        cChoice = getch();

        switch(cChoice)
        {
        case '1':   case '2':   case '3': case '4':
        case '5':   case '6':   case '7': case '8':
            nForm = (cChoice & 0x0f)-1;
            break;
        default:
            break;
        }

    }while(nForm < 0 || nForm > 7);

    printf("\n Enter the number of data points : ");
    scanf("%d",&nNumPoints);  fflush(stdin);

    /* Allocate memory */
    pnXPoints         = (double*) malloc(nNumPoints * sizeof(pnXPoints[0]));
    pnYPoints         = (double*) malloc(nNumPoints * sizeof(pnYPoints[0]));
    pnXOrigPoints     = (double*) malloc(nNumPoints * sizeof(pnXOrigPoints[0]));
    pnYOrigPoints     = (double*) malloc(nNumPoints * sizeof(pnYOrigPoints[0]));

    if(!pnXPoints || !pnYPoints || !pnXOrigPoints || !pnYOrigPoints)
        nStatus = -1;
```

```
    if(nStatus == 0)
    {
        int i;
        /* Ask for the points */
        for(i = 0; i < nNumPoints; i++)
        {
            printf(" Enter the value for (X%d): ",i);
            scanf("%lf",&pnXPoints[i]); fflush(stdin);
                      pnXOrigPoints[i] = pnXPoints[i];
            printf(" Enter the value for (Y%d): ",i);
            scanf("%lf",&pnYPoints[i]); fflush(stdin);
            pnYOrigPoints[i] = pnYPoints[i];

        }
    }
    /*  This is the computation stage */
    if(nStatus == 0)
    {
        nStatus =
computeNonlinearCurve(nForm,pnXPoints,pnYPoints,nNumPoints,&nA,&nB,&nC,&nD);
        if(nStatus)   nStatus -= 2;
    }

    /*  Display */
    if(nStatus == 0)
    {
        int i;

        printf("\n The least-squares curve is: ");

        switch(nForm)
        {
        case 0:
            printf(" y = %2.8lf / x + %2.8lf",nA,nB);
            break;
        case 1:
            printf(" y = %2.8lf / [x + %2.8lf] ",nD,nC);
            break;
        case 2:
            printf(" y = 1 / [%2.8lfx + %2.8lf] ",nA,nB);
            break;
        case 3:
            printf(" y = x / [%2.8lf + %2.8lfx] ",nA,nB);
            break;
        case 4:
            printf(" y = %2.8lf * ln(x) + %2.8lf",nA,nB);
            break;
        case 5:
            printf(" y = %2.8lf * exp(%2.8lfx)",nC,nA);
            break;
        case 6:
            printf(" y = %2.8lf * x^(%2.8lf)",nC,nA);
            break;
        case 7:
            printf(" y = [%2.8lfx + %2.8lf] ^ (-2)",nA,nB);
            break;
        default:
            break;
        }

        printf("\n\n  --------------------------- ");
        printf("\n\n N    \t     Xk     \t     Yk    \t     f(Xk) \n");
        for(i = 0; i < nNumPoints; i++)
```

```
        {
            double yVal = 0;
            switch(nForm)
            {
            case 0:
                yVal = nA / pnXOrigPoints[i] + nB;
                break;
            case 1:
                yVal =  nD / (pnXOrigPoints[i] + nC);
                break;
            case 2:
                yVal =  1 / (nA * pnXOrigPoints[i] + nB);
                break;
            case 3:
                yVal =  pnXOrigPoints[i] / (nA + nB * pnXOrigPoints[i]);
                break;
            case 4:
                yVal =  nA * log(pnXOrigPoints[i]) + nB;
                break;
            case 5:
                yVal =  nC * exp(nA * pnXOrigPoints[i]);
                break;
            case 6:
                yVal =  nC * pow(pnXOrigPoints[i],nA);
                break;
            case 7:
                yVal =  pow((nA * pnXOrigPoints[i]) + nB,-2);
                break;
            default:
                break;
            }
            printf("\n %d  \t %2.8lf \t %2.8lf \t
%2.8lf",i,pnXOrigPoints[i],pnYOrigPoints[i], yVal);
        }

        printf("\n\n  --------------------------- ");
    }

    switch(nStatus)
    {
    case -1:
        printf("\nMemory allocation error!!");
        break;
    case -2:
        printf("\nFailed: Insuffient data points!!");
        break;
    case -3:
        printf("\nFailed: Invalid nonlinear form!!");
        break;
    default:
        break;
    }
    printf("\n\nPress any key to continue...");
    /* Free the  Points */
    if(pnXPoints)   free(pnXPoints);
    if(pnYPoints)   free(pnYPoints);
    if(pnXOrigPoints) free(pnXOrigPoints);
    if(pnYOrigPoints)  free(pnYOrigPoints);

    getch();
    return nStatus;
}
```

```
#include "NonLinearFunc.h"
#include "LSLineFunc.h"

/* Function name       : computeNonlinearCurve
* Description         : Computes a non-linear curv using data linearization.
* Parameter(s)        :
*     nForm               [in]  The form of non-linear equation.
*     *pXPoints           [in]  The X coordinate vector.
*     *pYPoints           [in]  The Y coordinate vector corresponding to each point in X.
*     nNumPoints          [in]  The number of elements or data points.
*     pA                  [out] The A coefficient.
*     pB                  [out] The B coefficient.
*     pC                  [out] The C coefficient.
*     pD                  [out] The D coefficient.
* Return              :
*    int _cdecl       0  - Success
*                    -1  - Insufficient data points.
*                    -2  - Invalid nonlinear form.
*/
int  _cdecl
computeNonlinearCurve(int nForm,double *pXPoints,double *pYPoints,int nNumPoints,double*
pA,double* pB,double* pC,double* pD)
{
    int nRet = 0;
    int i;
    *pA = 0,  *pB = 0, *pC = 0, *pD = 0;
    switch(nForm)
    {
    case 0:
        /* Convert the Xj = 1/xj, Yj = yj; */
        for(i = 0; i < nNumPoints; i++)
        {
            pXPoints[i] = 1 / pXPoints[i];
            /* pYPoints[i] = pYPoints[i]; */
        }
        break;
    case 1:
        /* Convert the Xj = xjyj, Yj = yj; */
        for(i = 0; i < nNumPoints; i++)
        {
            pXPoints[i] = pXPoints[i] * pYPoints[i];
            /* pYPoints[i] = pYPoints[i]; */
        }
        break;
    case 2:
        /* Convert the Xj = xj, Yj = 1/yj; */
        for(i = 0; i < nNumPoints; i++)
        {
            /* pXPoints[i] = pXPoints[i]; */
            pYPoints[i] = 1 / pYPoints[i];
        }
        break;
    case 3:
        /* Convert the Xj = 1/xj, Yj = 1/yj; */
        for(i = 0; i < nNumPoints; i++)
        {
            pXPoints[i] = 1 / pXPoints[i];
            pYPoints[i] = 1 / pYPoints[i];
        }
        break;
    case 4:
```

```
        /* Convert the Xj = ln(xj), Yj = yj; */
        for(i = 0; i < nNumPoints; i++)
        {
            pXPoints[i] = log(pXPoints[i]);
           /* pYPoints[i] = pYPoints[i]; */
        }
        break;
    case 5:
        /* Convert the Xj = xj, Yj = ln(yj); */
        for(i = 0; i < nNumPoints; i++)
        {
            /* pXPoints[i] = pXPoints[i]; */
             pYPoints[i] = log(pYPoints[i]);
        }
        break;
    case 6:
        /* Convert the Xj = ln(xj), Yj = ln(yj); */
        for(i = 0; i < nNumPoints; i++)
        {
             pXPoints[i] = log(pXPoints[i]);
             pYPoints[i] = log(pYPoints[i]);
        }
        break;
    case 7:
        /* Convert the Xj = xj, Yj = (yj)^0.5; */
        for(i = 0; i < nNumPoints; i++)
        {
             /*pXPoints[i] = pXPoints[i]; */
             pYPoints[i] = 1 / sqrt(pYPoints[i]);
        }
        break;
    default:
        /* Invalid form */
        nRet = -2;
        break;
    }


    if(nRet == 0)
        /* Compute the linearized data */
        nRet = computeLSLine(pXPoints,pYPoints,nNumPoints,pA,pB);

    if(nRet == 0)
    {
        switch(nForm)
        {
        case 1:
            /* C = -1/A, D = -B/A */
            *pC = -1 / (*pA);
            *pD = -1 * (*pB) / (*pA);
            break;
        case 5:
        case 6:
            /* C = exp(B) */
            *pC  = exp(*pB);
            break;
        default:
            break;
        }
    }

    return nRet;
}
```

Program Output

```
 ***  Non-Linear Curve Fitting ***
Select the nonlinear form you wish to fit:
-----------------------------------------------------
1 ) y = A/ x + B          2) y = D / [x + C]
3 ) y = 1/ [Ax + B]       4) y = x / [A + Bx]
5 ) y = A * ln (x) + B    6) y = C * exp(Ax)
7 ) y = C * x^(A)         8) y = [Ax + B] ^ (-2)
-----------------------------------------------------
Enter the number of data points : 5
Enter the value for (X0): 0
Enter the value for (Y0): 1.5
Enter the value for (X1): 1
Enter the value for (Y1): 2.5
Enter the value for (X2): 2
Enter the value for (Y2): 3.5
Enter the value for (X3): 3
Enter the value for (Y3): 5
Enter the value for (X4): 4
Enter the value for (Y4): 7.5
The least-squares curve is:  y = 1.57990915 * exp(0.39120230x)
 ------------------------------
N           Xk              Yk              f(Xk)
0       0.00000000      1.50000000      1.57990915
1       1.00000000      2.50000000      2.33630272
2       2.00000000      3.50000000      3.45482550
3       3.00000000      5.00000000      5.10884959
4       4.00000000      7.50000000      7.55475034
 ------------------------------
Press any key to continue...
```

Composite Trapezoidal Rule

Discussion

Ever wondered how one could solve for definite integration programmatically the same way that some of the advanced scientific calculators do? The common conception is that it utilizes the same procedures applied as to one is solving for it analytically. But with knowledge on numerical integration, one can contend that every piece of the puzzle has fallen into place.

Before tackling on the **Composite Trapezoidal Rule,** a sneak peek on numerical integration should be taken first.

Numerical integration is the approximation of the definite integral of *f(x)* over the interval [*a*, *b*] by evaluating *f(x)* at a finite number of sample points. As described in definition 7.1 on p 347 of the textbook, it has the formula of the form:

$$Q[f] = \sum_{j=0}^{M} w_j f(x_j) \tag{1}$$

with a property of

$$\int_a^b f(x)dx = Q[f] + E[f]. \tag{2}$$

Q[*f*] can also be called **quadrature formula** while *E*[*f*] in (2) is the truncation error. The values $\{x_j\}_{j=0}^{M}$ and $\{w_j\}_{j=0}^{M}$ are called the quadrature nodes and weights, respectively.

Trapezoidal rule is one of the Newton-Cotes quadrature formulas which integrates the 1st degree polynomial $y = P_1(x)$ over the interval $[x_0, x_1]$ with a formula:

$$\int_{x_0}^{x_1} f(x)dx \approx \int_{x_0}^{x_1} P_1(x)dx = \frac{h}{2}(f_0 + f_1) \tag{3}$$

where f_0 and f_1 are the values evaluated at $f(x_0)$ and $f(x_1)$, respectively. Since its derivation is not shown in the textbook, one is provided below using the Lagrange approximation method.

$$\int_{x_0}^{x_1} f(x)dx \approx \int_{x_0}^{x_1} P_1(x)dx = \int_{x_0}^{x_1} f_0 \frac{(x-x_1)}{(x_0-x_1)} + \int_{x_0}^{x_1} f_1 \frac{(x-x_0)}{(x_1-x_0)}$$

Let $x = x_0 + ht$, $dx = hdt$ and integrating with respect to t with limits t=0 and t=1.

$$= f_0 \int_0^1 \frac{h(t-1)}{-h} h dt + f_1 \int_0^1 \frac{ht}{h} h dt$$

$$= h\left[f_0 \int_0^1 (1-t)\,dt + f_1 \int_0^1 t dt \right]$$

$$= h\left[f_0 \left[t - \frac{1}{2}t^2 \right]_0^1 + f_1 \left[\frac{1}{2}t^2 \right]_0^1 \right]$$

$$\boxed{= \frac{h}{2}[f_0 + f_1].}$$

Composite trapezoidal rule is a series of approximations in finding for the area under the curve $y = f(x)$ over an interval $[a, b]$ using the trapezoidal rule. With the interval $[a, b]$ subdivided into M subintervals $[x_k, x_{k+1}]$, of equal width $h = (b - a)/$M, the formulas for the composite trapezoidal rule can be expressed by:

$$T(f,h) = \frac{h}{2}\sum_{k=1}^{M}[f(x_{k-1}) + f(x_k)] \tag{4a}$$

or

$$T(f,h) = \frac{h}{2}[f_0 + 2f_1 + 2f_2 + 2f_3 + \ldots + 2f_{M-2} + 2f_{M-1} + f_M] \tag{4b}$$

or

$$T(f,h) = \frac{h}{2}[f(a) + f(b)] + h\sum_{k=1}^{M-1} f(x_k). \tag{4c}$$

The error term for the composite trapezoidal rule is:

$$E_T(f,h) = \frac{-(b-a)f^{(2)}(c)h^2}{12} \tag{5}$$

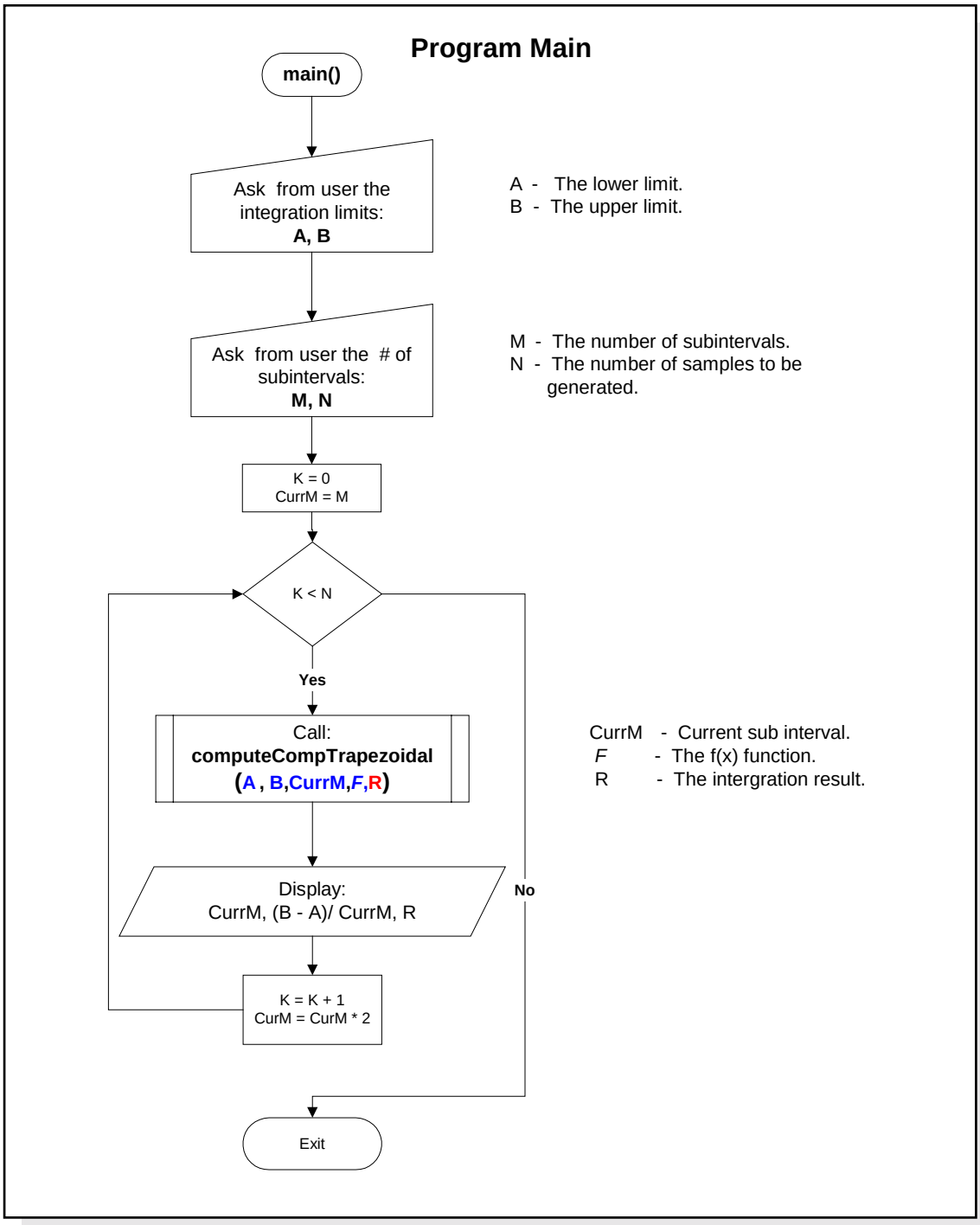
with an order of $\boldsymbol{O(h^2)}$. By decreasing the width of h in (6), a reduction on its error can also be observed. Details on error analysis of the composite trapezoidal rule are discussed on pp 361-360 of the textbook.

Algorithm 7.1 (Composite Trapezoidal Rule). To approximate the integral

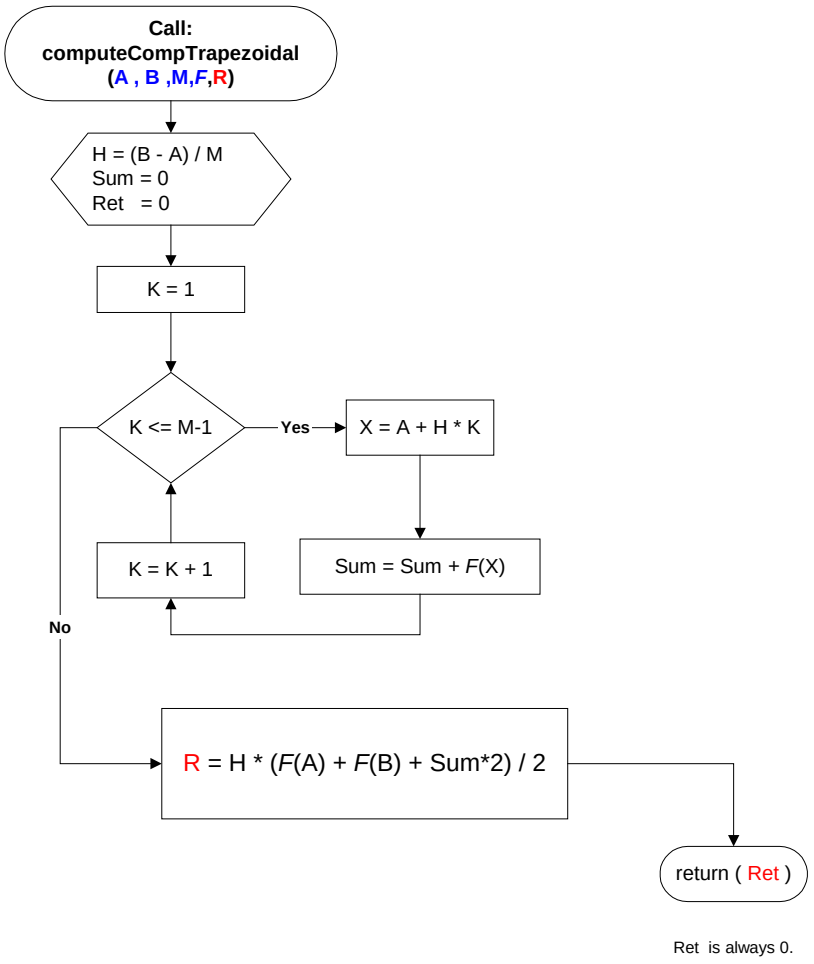
$$\int_A^B f(x)dx \approx \frac{h}{2}[f(A)+f(B)] + h\sum_{k=1}^{M-1} f(x_k).$$

By sampling $f(x)$ at the M+1 equally spaced points x_k = A + hk for k=0,1,2,...,M. Notice that x_0=A and x_M=B.

Program Flow



Composite Trapeziodal Rule Function



Ret is always 0.

Program Source Code

```
#include <stdio.h>
#include <conio.h>
#include "CompTrapFunc.h"

double  functionOfX(double nX);

int main(int argc, char* argv[])
{
    int     i;
    double  nLowerLimit;
    double  nUpperLimit;
    int     nStartingInterval;
    int     nNumberOfSamples;

    printf("\n ***  Composite Trapezoidal Rule *** \n");
    printf("\n f(x) =  2x cos (x) dx \n\n");

    /*  This is the input stage */
    printf(" Enter the lower limit: ");
    scanf("%lf",&nLowerLimit);    fflush(stdin);
    printf(" Enter the upper limit: ");
    scanf("%lf",&nUpperLimit);    fflush(stdin);

    /*  Ask for the interval*/
    do{
        printf(" Enter the interval M      [ >= 10]: ");
        scanf("%d",&nStartingInterval);  fflush(stdin);
    }while(nStartingInterval < 10);

    /*  Ask for the number of trials*/
    do{
        printf(" Enter the number of samples [ > 0]: ");
        scanf("%d",&nNumberOfSamples);    fflush(stdin);
    }while(nNumberOfSamples <= 0);

    /* Compute and display the output */
    printf("\n Result : \n");
    printf("\n --------------------------- ");
    printf("\n     M   \t     h     \t     T(f,h)   ");
    for(i = 0; i < nNumberOfSamples; i++)
    {
        double nResult;

computeCompTrapezoidal(nLowerLimit,nUpperLimit,nStartingInterval,functionOfX,&nResult);
        printf("\n %8d   \t   %2.8lf  \t   %2.8lf",nStartingInterval,(nUpperLimit -
nLowerLimit) / (double) nStartingInterval,nResult);
        nStartingInterval *= 2;
    }

    printf("\n --------------------------- \n");

    printf("\n Press any key to continue....");
    getch();

    return 0;
}


double  functionOfX(double nX)
{
    return 2 * nX  * cos(nX);
}
```

```
#include "CompTrapFunc.h"

/* Function name       : computeCompTrapezoidal
* Description         : Computes the numerical intergal of a function  using the
*                       Composite Trapezoidal Rule.
* Parameter(s)        :
*     nLowerLimit        [in]  The lower limit of intergration.
*     nUpperLimit        [in]  The upper limit of integration.
*     nInterval          [in]  The number of interval for intergration.
*     funcOfX            [in]  Pointer to the function the will evalute the f(x).
*     *pResult           [out] The integration result.
* Return              :
*    int           0 -  Success
*/
int _cdecl
computeCompTrapezoidal(double nLowerLimit,double nUpperLimit,int nInterval, double
(*funcOfX)(double nX),double *pResult)
{
    int nRet = 0;

    double nWeightH;
    double nSum;

    int   i;

    nWeightH = (nUpperLimit - nLowerLimit) / (double) nInterval;

    nSum = 0;

    /* Compute first the summation */
    for(i = 1; i <= (nInterval - 1); i++)
    {
        double nX;

        nX = nLowerLimit + nWeightH * i;

        /* Compute using the passed function of x */
        nSum += funcOfX(nX);
    }


    *pResult = nWeightH * (funcOfX(nLowerLimit) + funcOfX(nUpperLimit) + nSum * 2)/2;


    return nRet;
}
```

Program Output

```
***  Composite Trapezoidal Rule ***

f(x) =  2x cos (x) dx

Enter the lower limit: 0
Enter the upper limit: 2
Enter the interval M      [ >= 10]: 20
Enter the number of samples [ > 0]: 10

Result :

-----------------------------
    M                h              T(f,h)
   20          0.10000000      0.79950311
   40          0.05000000      0.80354812
   80          0.02500000      0.80455908
  160          0.01250000      0.80481180
  320          0.00625000      0.80487497
  640          0.00312500      0.80489077
 1280          0.00156250      0.80489472
 2560          0.00078125      0.80489571
 5120          0.00039063      0.80489595
10240          0.00019531      0.80489601
-----------------------------

Press any key to continue....
```

Composite Simpson's Rule

Discussion

Just like Composite Trapezoidal Rule, the Composite **Simpson's rule,** which is a series of approximations used in finding for the area under the curve $y = f(x)$ over an interval $[a, b]$ using the Simpson's rule, is another way in solving for numerical integration

Simpson's rule is also one of the Newton-Cotes quadrature formulas which integrates the 2nd degree polynomial $y = P_2(x)$ over the interval $[x_0, x_2]$ with a formula:

$$\int_{x_0}^{x_2} f(x)dx \approx \int_{x_0}^{x_2} P_2(x)dx = \frac{h}{3}(f_0 + 4f_1 + f_2) \qquad (1)$$

where f_0, f_1 and f_2 are the values evaluated at $f(x_0)$, $f(x_1)$ and $f(x_2)$, respectively. The derivation for the Simpson's rule can be found in pp. 350-351 of the textbook.

With the interval $[a, b]$ subdivided into 2M subintervals $[x_k, x_{k+1}]$, of equal width $h = (b - a)/(2M)$, the formulas for the composite Simpson's rule can be expressed by:

$$S(f,h) = \frac{h}{3}\sum_{k=1}^{M}[f(x_{2k-2}) + 4f(x_{2k-1}) + f(x_{2k})] \qquad (2a)$$

or

$$S(f,h) = \frac{h}{3}[f_0 + 4f_1 + 2f_2 + 4f_3 + ... + 2f_{2M-2} + 4f_{2M-1} + f_{2M}] \qquad (2b)$$

or

$$S(f,h) = \frac{h}{3}[f(a) + f(b)] + \frac{2h}{3}\sum_{k=1}^{M-1} f(x_{2k}) + \frac{4h}{3}\sum_{k=1}^{M} f(x_{2k-1}). \qquad (2c)$$

The error term for the composite Simpson's rule is:

$$E_s(f,h) = \frac{-(b-a)f^{(4)}(c)h^4}{180} \qquad (3)$$

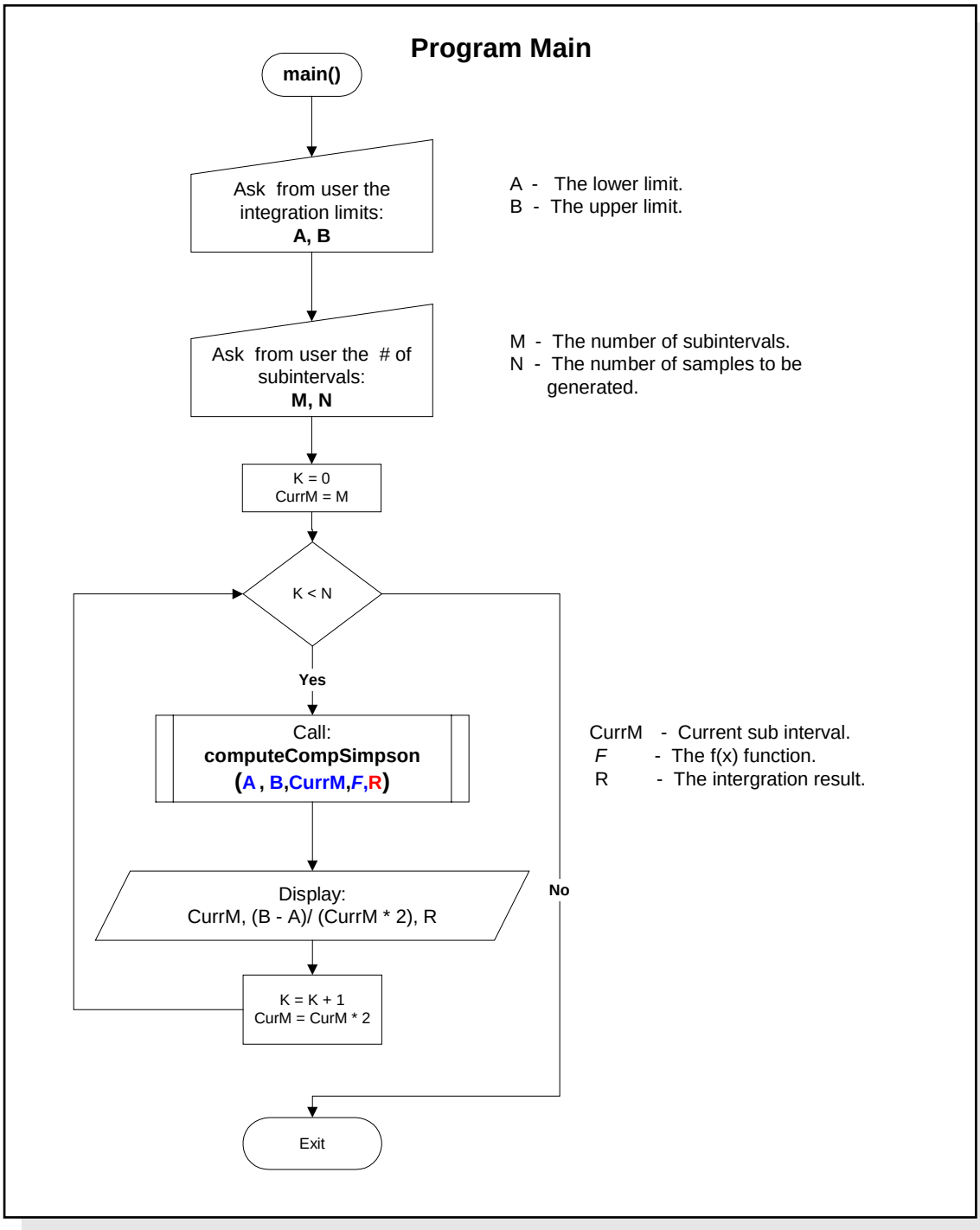
with an order of $O(h^4)$. By decreasing the width of h in (3), a reduction on its error can be observed. In fact, this decline is faster than the composite trapezoidal rule due to its given order. Details on the error analysis of the composite Simpson's rule are discussed on pp. 362-363 of the textbook.

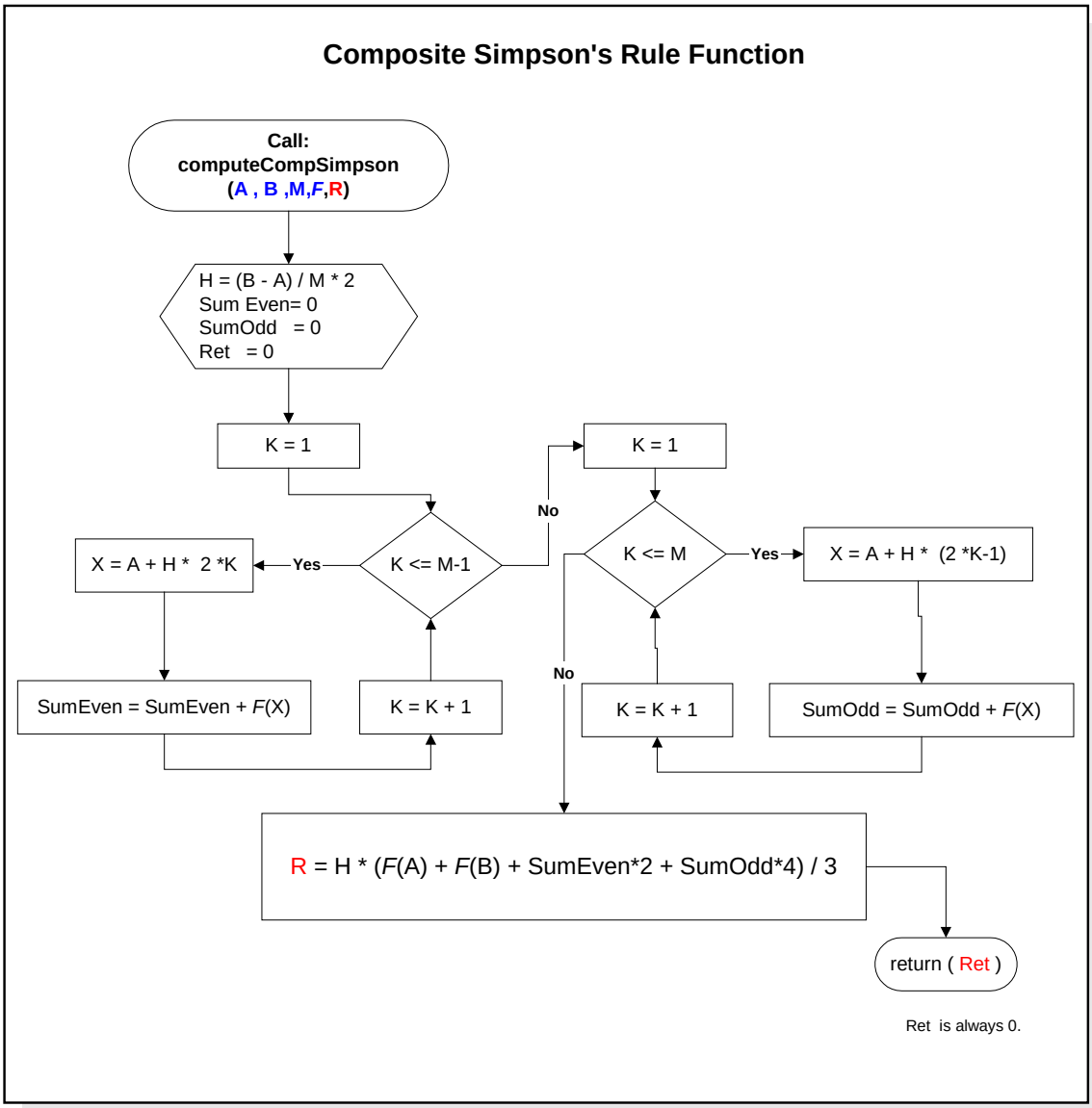
Algorithm 7.2 (Composite Simpson's Rule). To approximate the integral

$$\int_A^B f(x)dx \approx \frac{h}{3}[f(A)+f(B)]+\frac{2h}{3}\sum_{k=1}^{M-1} f(x_{2k})+\frac{4h}{3}\sum_{k=1}^{M} f(x_{2k-1}).$$

By sampling $f(x)$ at the 2M+1 equally spaced points x_k = A + hk for k=0,1,2,...,2M. Notice that x_0=A and x_{2M}=B.

Program Flow




Composite Simpson's Rule Function
Call:
computeCompSimpson
(A , B ,M,F,R)
H = (B - A) / M * 2
Sum Even= 0
SumOdd = 0
Ret = 0
K = 1
K <= M-1
Yes
No
X = A + H * 2 *K
SumEven = SumEven + F(X)
K = K + 1
K = 1
K <= M
Yes
No
X = A + H * (2 *K-1)
SumOdd = SumOdd + F(X)
K = K + 1
R = H * (F(A) + F(B) + SumEven*2 + SumOdd*4) / 3
return (Ret)
Ret is always 0.

Program Source Code

```
#include <stdio.h>
#include <conio.h>
#include "CompSimpFunc.h"

double  functionOfX(double nX);

int main(int argc, char* argv[])
{

    int     i;
    double  nLowerLimit;
    double  nUpperLimit;
    int     nStartingInterval;
    int     nNumberOfSamples;

    printf("\n ***  Composite Simpson's Rule *** \n");
    printf("\n f(x) =  2x cos (x) dx \n\n");

    /*  This is the input stage */
    printf(" Enter the lower limit: ");
    scanf("%lf",&nLowerLimit);   fflush(stdin);
    printf(" Enter the upper limit: ");
    scanf("%lf",&nUpperLimit);   fflush(stdin);

    do{
        printf(" Enter the interval M      [ >= 10]: ");
        scanf("%d",&nStartingInterval);  fflush(stdin);
    }while(nStartingInterval < 10);

    do{
        printf(" Enter the number of samples [ > 0]: ");
        scanf("%d",&nNumberOfSamples); fflush(stdin);
    }while(nNumberOfSamples <= 0);


    /* Compute and display the output */
    printf("\n Result : \n");
    printf("\n --------------------------- ");
    printf("\n   M      \t      h      \t      T(f,h)   ");
    for(i = 0; i < nNumberOfSamples; i++)
    {
        double nResult;

computeCompSimpson(nLowerLimit,nUpperLimit,nStartingInterval,functionOfX,&nResult);
        printf("\n %8d   \t  %2.8lf  \t   %2.8lf",nStartingInterval,(nUpperLimit -
nLowerLimit) / (double) (nStartingInterval * 2),nResult);
        nStartingInterval *= 2;
    }

    printf("\n --------------------------- \n");
    printf("\n Press any key to continue....");
    getch();

    return 0;
}


double  functionOfX(double nX)
{
    return 2 * nX  * cos(nX);
    /*return (2 + sin(2 * sqrt(nX)));*/
}
```

```
#include "CompSimpFunc.h"

/* Function name       : computeCompSimpson
* Description         : Computes the numerical intergal of a function  using the
*                       Composite Simpson's Rule.
* Parameter(s)        :
*     nLowerLimit        [in]  The lower limit of intergration.
*     nUpperLimit        [in]  The upper limit of integration.
*     nInterval          [in]  The number of interval for intergration.
*     funcOfX            [in]  Pointer to the function the will evaluate the f(x).
*     *pResult           [out] The integration result.
* Return              :
*    int  0 - Success
*/
int _cdecl
computeCompSimpson(double nLowerLimit,double nUpperLimit,int nInterval, double
(*funcOfX)(double nX),double *pResult)
{
    int nRet = 0;

    double nWeightH;
    double nSumEven;
    double nSumOdd;
    int   i;
    nWeightH = (nUpperLimit - nLowerLimit) / (double) (nInterval * 2);

    nSumEven = 0;
    /* Compute the summation of even intervals */
    for(i = 1; i <= (nInterval - 1); i++)
    {
        double nX;

        nX = nLowerLimit + nWeightH * 2 * i;

        /* Compute using the passed function of x */
        nSumEven += funcOfX(nX);
    }

    nSumOdd = 0;
    /* Compute the summation of odd intervals */
    for(i = 1; i <= nInterval; i++)
    {
        double nX;

        nX = nLowerLimit + nWeightH * (2 * i - 1);

        /* Compute using the passed function of x */
        nSumOdd += funcOfX(nX);
    }

    *pResult = nWeightH * (funcOfX(nLowerLimit) + funcOfX(nUpperLimit) + (nSumEven * 2) +
(nSumOdd * 4))/3;

    return nRet;
}
```

Program Output

```
***  Composite Simpson's Rule ***

f(x) =  2x cos (x) dx

Enter the lower limit: 0
Enter the upper limit: 2
Enter the interval M      [ >= 10]: 20
Enter the number of samples [ > 0]: 10

Result :

----------------------------
  M                  h              T(f,h)
     20         0.05000000       0.80489646
     40         0.02500000       0.80489606
     80         0.01250000       0.80489604
    160         0.00625000       0.80489603
    320         0.00312500       0.80489603
    640         0.00156250       0.80489603
   1280         0.00078125       0.80489603
   2560         0.00039063       0.80489603
   5120         0.00019531       0.80489603
  10240         0.00009766       0.80489603
----------------------------

Press any key to continue....
```

Euler's Method

Discussion

There are a lot of possible solutions for a given differential equation. Thus, when plotted on a graphical representation, one would usually find a family of curves, called the **general solution** to a given differential equation. One that can be obtained from the general solution by giving specific values to every arbitrary constant is called a **particular solution**.

Initial value problem is a particular solution to a differential equation since it is given an initial value in order to trace the path of the specific curve.

There are several known methods to solve for the initial value problem, one of which is the **Euler's method**. It has a limited usage because of the larger error that it generates as the process goes on. However, it is still an essential method for learning since apart from being the simplest among all the methods, its concepts are used in the advanced ones.

Given an I.V.P $y' = f(t, y)$ with $y(a) = y_0$ within an interval $[a, b]$ where the approximated points $\{(t_k, y_k)\}$of the solution will be taken, one must first provide the abscissas within the said interval. The easiest way of doing this is to subdivide the interval $[a, b]$ into M equal subintervals and select the mesh points.

$$t_k = a + hk \quad \text{for } k = 0,1,...,M \quad \text{where } h = \frac{b-a}{M}. \tag{1}$$

The value h in (1) is called the step size. Once the abscissas have been set up, one may now proceed to the approximation part for

$$y' = f(t, y) \quad \text{over} \quad [t_0, t_M] \quad \text{with} \quad y(t_0) = y_0. \tag{2}$$

Euler's approximation uses

$$y(t_1) = y(t_0) + hf(t_0, y_0). \tag{3}$$

in finding for the value of $y(t_1)$ which is derived from Taylor's theorem. Its details can be found on pp. 429-430 of the textbook.

Using (3), the process is repeated to obtain the sequence of points that approximates the solution curve $y = y(t)$ with the general step

$$t_{k+1} = t_k + h, \quad y_{k+1} = y_k + hf(t_k, y_k) \quad \text{for } k = 0,1,....M-1. \tag{4}$$

The figure below shows the vertical values of Euler's approximation and the curve of the unique solution $y=y(t)$ as they move in a horizontal direction.

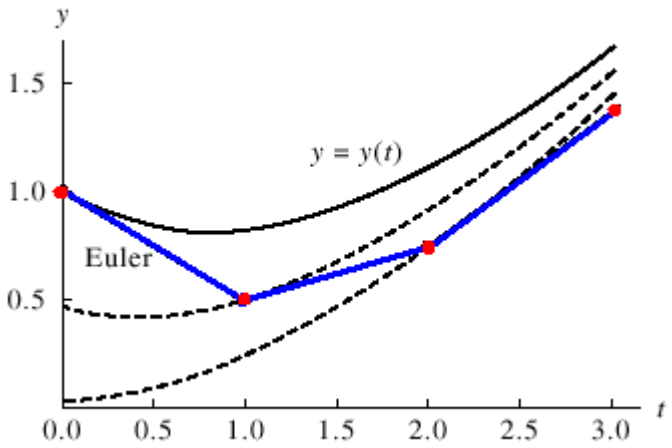


Figure 9.1.1 Euler's approximations using (4).

Euler's method is a **difference method** or a **discrete variable method** since the solution is approximated on a set of discrete points. Because of this, two sources of errors are generated. These are the *discretization* and *round-off errors*. The discretization error is classified into two, one of which is the difference between the unique solution and the solution obtained by the discrete variable method called the **global discretization error** e_k which is defined by:

$$e_k = y(t_k) - y_k \qquad \text{for } k = 1,2,....M. \tag{5}$$

The other error, committed in the single step from t_k to t_{k+1}, is called the **local discretization error** $\in_k$ defined by:

$$\in_k = y(t_{k+1}) - y_k - h\Phi(t_k, y_k) \qquad \text{for } k = 0,1,....M-1. \tag{6}$$

where Φ, which is used in the approximation, in (6) is called the **increment function**.

The **final global error (F.G.E)**, the one at the end of the interval is defined by:

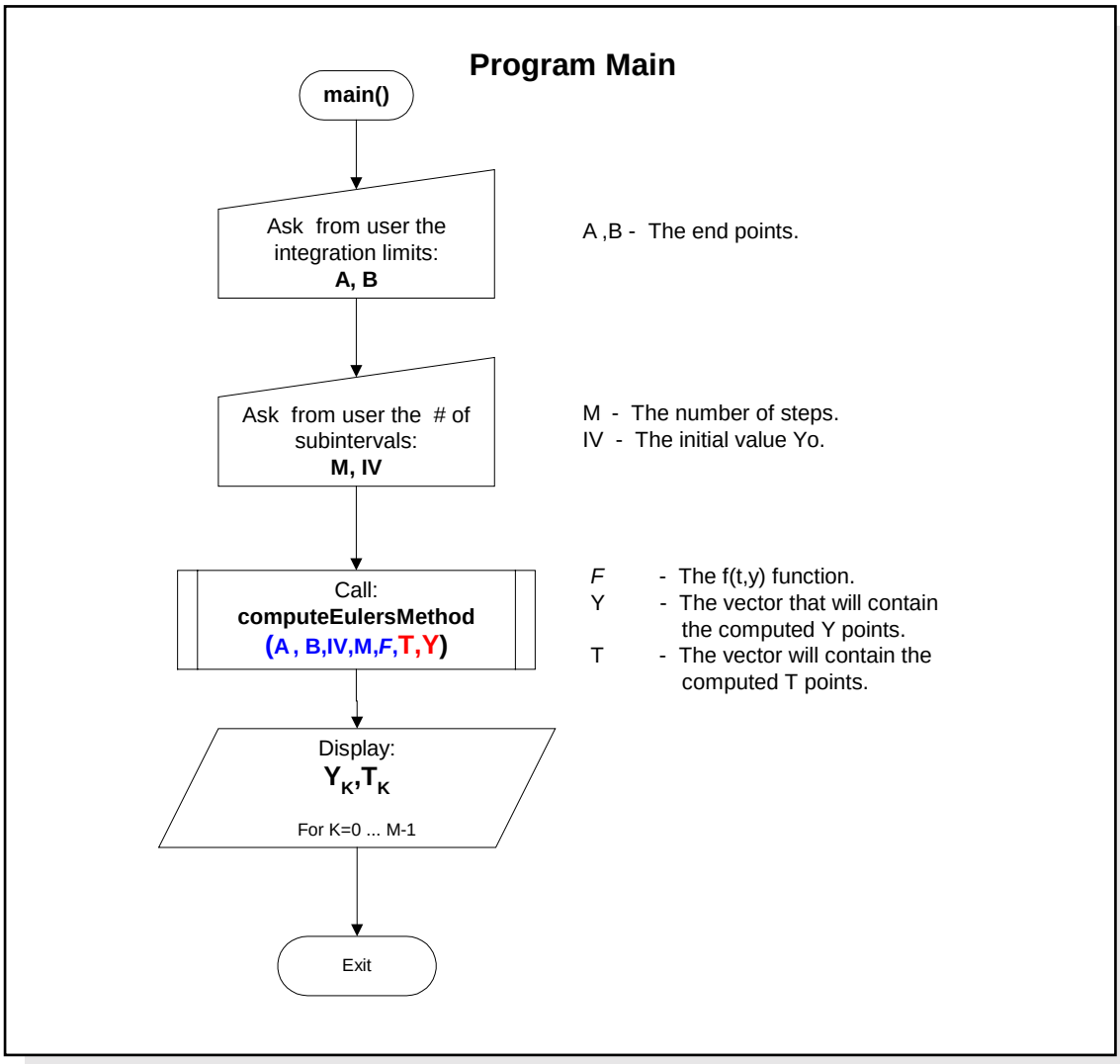
$$E(y(b), h) = |y(b) - yM| = O(h). \tag{7}$$

Details on the F.G.E can be found on pp. 432-433 of the textbook.

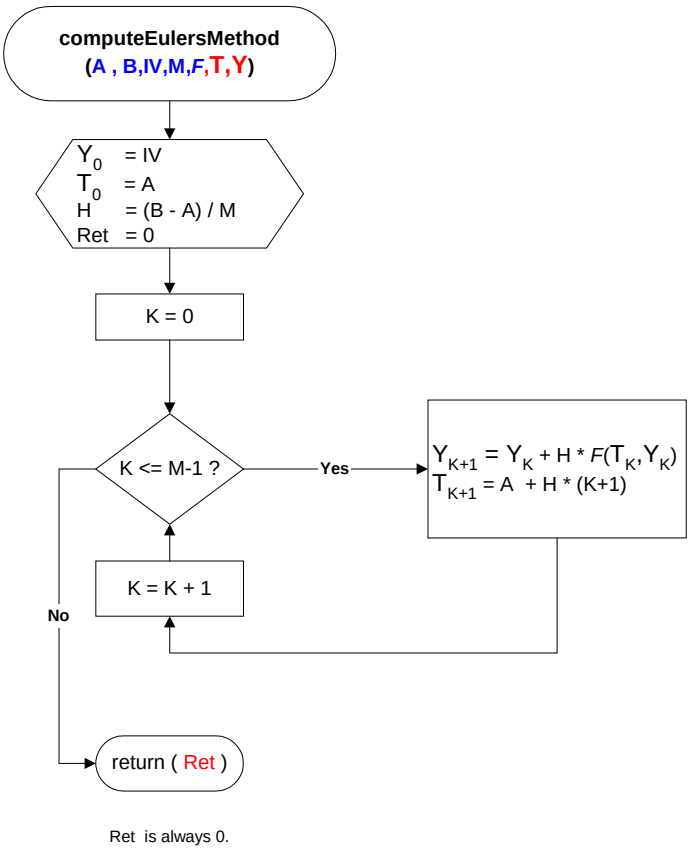
Algorithm 9.1 (Euler's Method). To approximate the solution of the initial value problem $y' = f(t, y)$ with $y(a) = y0$ over $[a, b]$ by computing

$$y_{k+1} = y_k + hf(t_k, y_k) \quad \text{for } k = 0,1,....M-1.$$

Program Flow



Euler's (Method) Function



Ret is always 0.

Program Source Code

```
#include  <memory.h>
#include  <stdio.h>
#include  <conio.h>
#include  "EulersFunc.h"
/* The function y' = f(t,y) */
double  functionOfTY(double nT,double nY);
int main(int argc, char* argv[])
{
    int nStatus = 0;
    int  i;
    double*  pTPoints = NULL;
    double*  pYPoints = NULL;
    int      nInterval;
    double   nLowerLimit,nUpperLimit,nInitYValue;
    printf("\n ***  Euler's Method *** \n");
    printf("\n y' =  (t - y)/2 \n\n");
    /*  This is the input stage */
    printf(" Enter the lower limit: ");
    scanf("%lf",&nLowerLimit);   fflush(stdin);
    printf(" Enter the upper limit: ");
    scanf("%lf",&nUpperLimit);   fflush(stdin);
    /*  Ask for the interval*/
    do{
        printf(" Enter the interval M  [ > 0]: ");
        scanf("%d",&nInterval);   fflush(stdin);
    }while(nInterval <= 0);
    /*  Ask for the initial value y(0)*/
    printf(" Enter the initial value y(0): ");
    scanf("%lf",&nInitYValue); fflush(stdin);
    pTPoints = (double*) malloc((nInterval + 1) * sizeof(pTPoints[0]));
    pYPoints = (double*) malloc((nInterval + 1) * sizeof(pYPoints[0]));
    if(!pTPoints || !pYPoints)    nStatus = -1;
    if(nStatus == 0)
    {
computeEulersMethod(nLowerLimit,nUpperLimit,nInitYValue,nInterval,functionOfTY,pTPoints,p
YPoints);
        printf("\n Result:");
        printf("\n ------------------------- ");
        printf("\n   t(k)     \t     y(k)     ");
        for(i = 0; i <= nInterval; i++ )
            printf("\n %2.8lf \t  %2.8lf",pTPoints[i],pYPoints[i]);
        printf("\n ------------------------- ");
    }
    /* Report any errors */
    switch(nStatus)
    {
    case -1:
        printf("\nMemory allocation error!!");
        break;
    default:
        break;
    }
    /* Free the allocation */
    if(pTPoints)   free(pTPoints);
    if(pYPoints)   free(pYPoints);

    printf("\nPress any key to continue....");
    getch();
    return 0;
}
/* Evaluates the function y' = f(t,y) */
double  functionOfTY(double nT,double nY)
{    return ((nT - nY) / 2); }
```

```
#include  "EulersFunc.h"

/* Function name       : computeEulersMethod
* Description          :  Computes the initial value problem y' = f(t,y) using
*                         Euler's method.
* Parameter(s)         :
*     nLowerLimit       [in]  The lower limit.
*     nUpperLimit       [in]  The upper limit.
*     nInitYValue       [in]  The initial value Y(0).
*     nInterval         [in]  The interval M.
*     pFuncOfTY         [in]  The function y' = f(t,y).
*     *pTPoints         [out] Stores the values of t(k).
*     *pYPoints         [out] Stores the values of y(k).
* Return              :
*    int           - 0  Success.
*/
int _cdecl
computeEulersMethod(double nLowerLimit,double nUpperLimit,double nInitYValue,int
nInterval,double (*pFuncOfTY)(double t,double y),double *pTPoints,double *pYPoints)
{
    int nRet = 0;

    int     i;
    double  nWeightH;

    /* Clear the pTPoints && pYPoints. */
    memset(pTPoints,0,(nInterval + 1) * sizeof(pTPoints[0]));
    memset(pYPoints,0,(nInterval + 1) * sizeof(pTPoints[0]));

    nWeightH = (nUpperLimit - nLowerLimit) / (double)nInterval;

    /* Set the lower limit is the initial value of t */
    pTPoints[0] = nLowerLimit;
    pYPoints[0] = nInitYValue;

    /* Start the iteration using the Euler's Method */
    for(i = 0; i < nInterval; i++)
    {
        pYPoints[i+1] = pYPoints[i] + nWeightH * pFuncOfTY(pTPoints[i],pYPoints[i]);
        pTPoints[i+1] = nLowerLimit + nWeightH * (i+1);
    }

       return nRet;
}
```

Program Output

```
***  Euler's Method ***

y' =  (t - y)/2

Enter the lower limit: 0
Enter the upper limit: 3
Enter the interval M  [ > 0]: 24
Enter the initial value y(0): 1

Result:
---------------------------
  t(k)              y(k)
0.00000000       1.00000000
0.12500000       0.93750000
0.25000000       0.88671875
0.37500000       0.84692383
0.50000000       0.81742859
0.62500000       0.79758930
0.75000000       0.78680247
0.87500000       0.78450232
1.00000000       0.79015842
1.12500000       0.80327352
1.25000000       0.82338143
1.37500000       0.85004509
1.50000000       0.88285477
1.62500000       0.92142635
1.75000000       0.96539970
1.87500000       1.01443722
2.00000000       1.06822239
2.12500000       1.12645849
2.25000000       1.18886734
2.37500000       1.25518813
2.50000000       1.32517637
2.62500000       1.39860285
2.75000000       1.47525267
2.87500000       1.55492438
3.00000000       1.63742910
---------------------------
Press any key to continue....
```

Heun's Method

Discussion

Another method in solving for the I.V.P. is the **Heun's method**, which utilizes the fundamental theorem of calculus**.** Given the I.V.P:

$$y' = f(t,y) \quad \text{over} \quad [a,b] \quad \text{with} \quad y(t_0) = y_0, \tag{1}$$

one can approximate $y(t_1)$ by using the formula:

$$y(t_1) \approx y(t_0) + \frac{h}{2}\left[f(t_0, y(t_0)) + f(t_1, y(t_1))\right]. \tag{2}$$

Details for obtaining (2) from (1) are tackled on pp. 437-438 of the textbook. Notice that the second term on the right side uses the trapezoidal rule. That same term, however, uses $y(t_1)$ which is yet to be solved. In order to cope with this shortcoming, the Euler's method is used to estimate the value for $y(t_1)$. Substituting Euler's method in (2) in finding for the values of (t_1, y_1) now becomes the Heun's method:

$$y_1 = y(t_0) + \frac{h}{2}\left[f(t_0, y_0) + f(t_1, y_0 + hf(t_0, y_0))\right]. \tag{3}$$

which has a general step of:

$$\begin{aligned} P_{k+1} &= y_k + hf(t_k, y_k), \qquad t_{k+1} = t_k + h, \\ y_{k+1} &= y_k + \frac{h}{2}\left[f(t_k, y_k) + f(t_{k+1}, P_{k+1})\right] \end{aligned} \tag{4}$$

For this method, the final global error (F.G.E), is defined by:

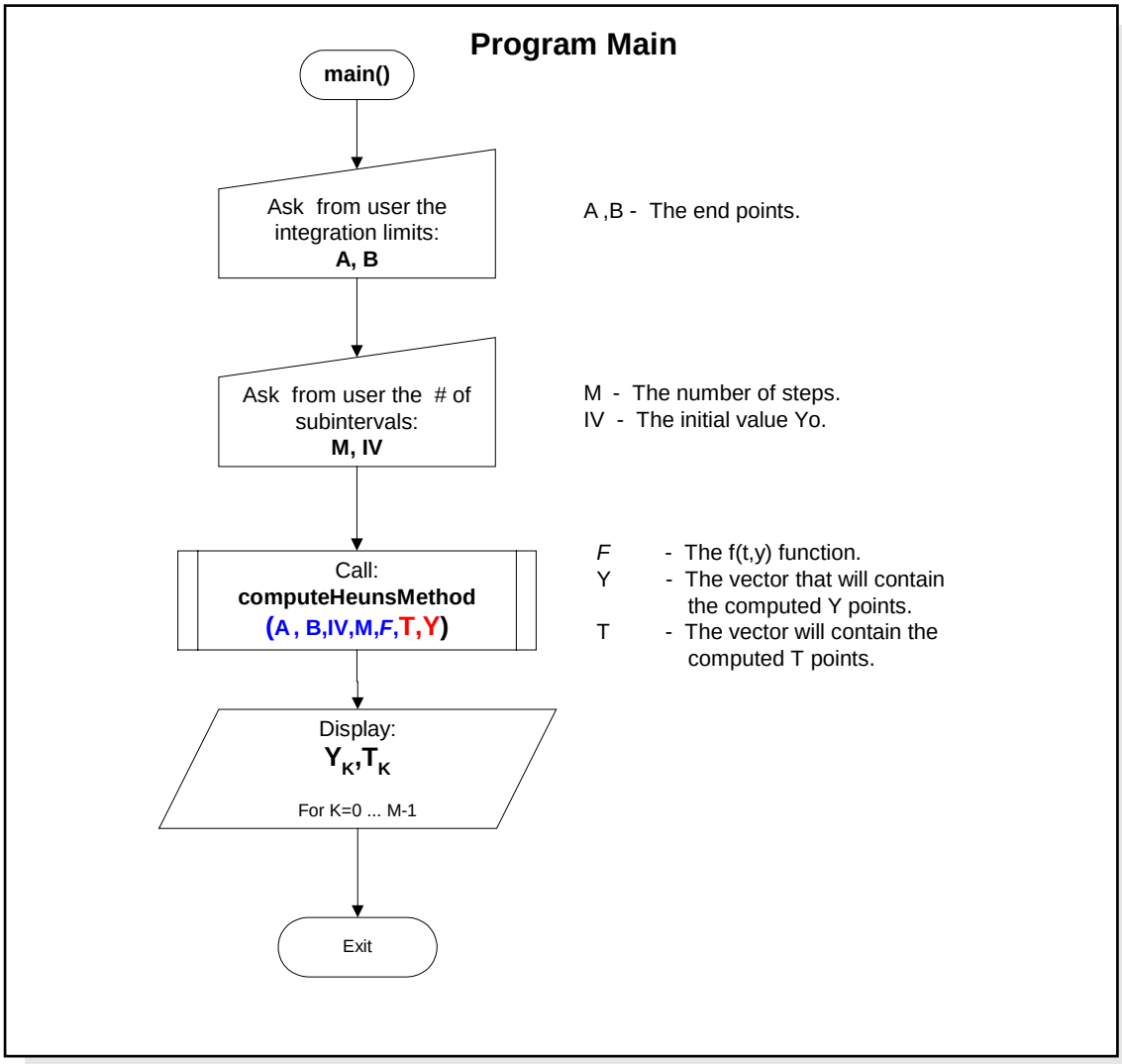
$$E(y(b), h) = \left|y(b) - yM\right| = O(h^2). \tag{5}$$

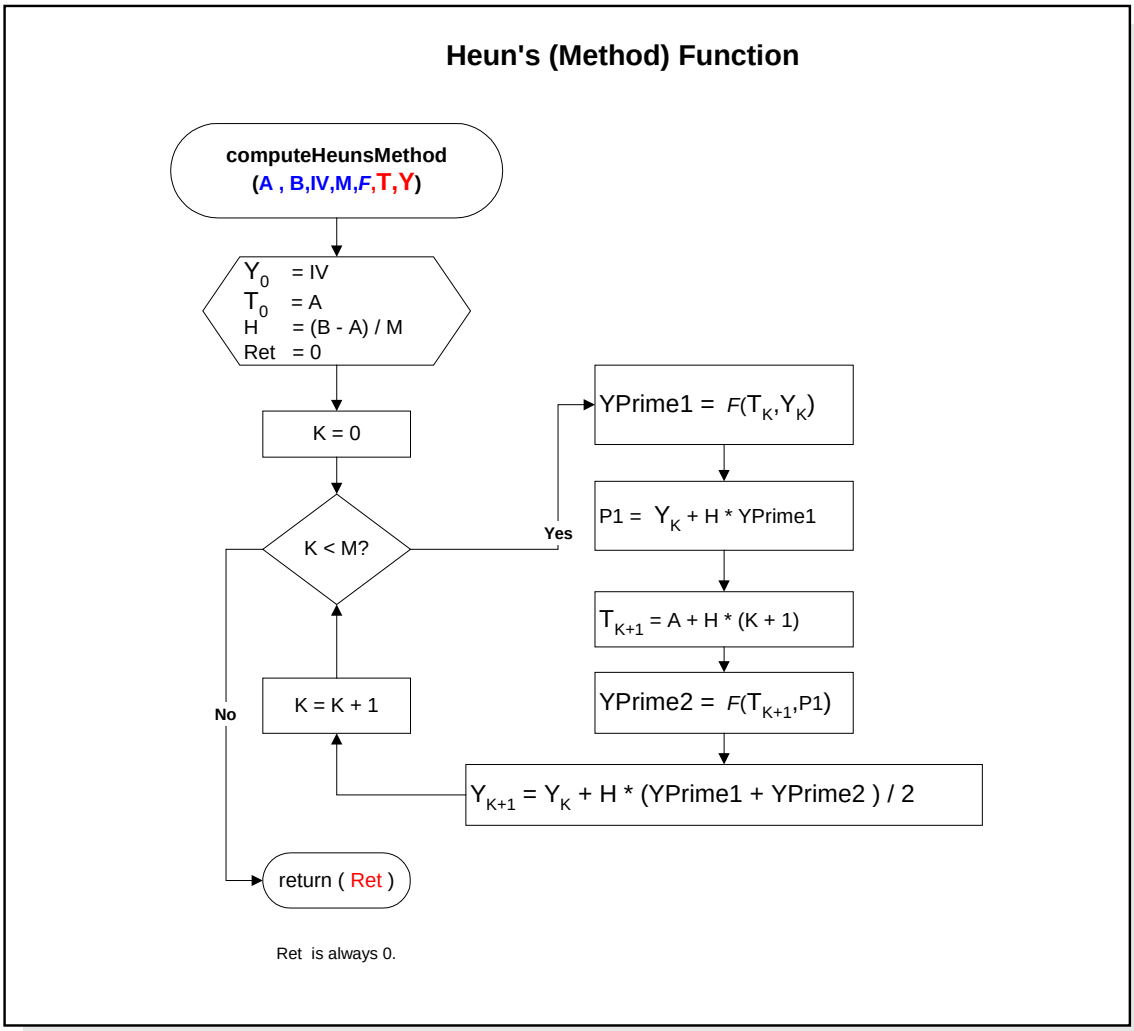
More discussion in obtaining for the F.G.E is found on page 439 of the textbook.

Algorithm 9.2 (Heun's Method). To approximate the solution of the initial value problem $y' = f(t, y)$ with $y(a) = y0$ over $[a, b]$ by computing

$$y_{k+1} = y_k + \frac{h}{2}\left[f(t_k, y_k) + f(t_{k+1}, y_k + hf(t_k, y_k)) \right] \quad \text{for } k = 0,1,....M-1.$$

Program Flow




Heun's (Method) Function
computeHeunsMethod
(A , B,IV,M,F,T,Y)
Y0 = IV
T0 = A
H = (B - A) / M
Ret = 0
K = 0
K < M?
Yes
No
YPrime1 = F(TK,YK)
P1 = YK + H * YPrime1
TK+1 = A + H * (K + 1)
YPrime2 = F(TK+1,P1)
YK+1 = YK + H * (YPrime1 + YPrime2) / 2
K = K + 1
return (Ret)
Ret is always 0.

Program Source Code

```
#include  <memory.h>
#include  <stdio.h>
#include  <conio.h>
#include  "HeunsFunc.h"
/* The function y' = f(t,y) */
double  functionOfTY(double nT,double nY);
int main(int argc, char* argv[])
{
       int nStatus = 0;
       int  i;
       double*  pTPoints = NULL;
       double*  pYPoints = NULL;
       int      nInterval;
       double   nLowerLimit, nUpperLimit, nInitYValue;
       printf("\n ***  Heuns's Method *** \n");
       printf("\n y' =  (t - y)/2 \n\n");

       /*  This is the input stage */
       printf(" Enter the lower limit: ");
       scanf("%lf",&nLowerLimit);    fflush(stdin);
       printf(" Enter the upper limit: ");
       scanf("%lf",&nUpperLimit);  fflush(stdin);
       /*  Ask for the interval*/
       do{
              printf(" Enter the interval M  [ > 0]: ");
              scanf("%d",&nInterval); fflush(stdin);
              /* make sure that the user has the correct input */
       }while(nInterval <= 0);
       printf(" Enter the initial value y(0): ");
       scanf("%lf",&nInitYValue);    fflush(stdin);
       pTPoints = (double*) malloc((nInterval + 1) * sizeof(pTPoints[0]));
       pYPoints = (double*) malloc((nInterval + 1) * sizeof(pYPoints[0]));
       if(!pTPoints || !pYPoints)    nStatus = -1;
       if(nStatus == 0)
       {
       computeHeunsMethod(nLowerLimit,nUpperLimit,nInitYValue,nInterval,functionOfTY,pTPo
ints,pYPoints);
              printf("\n Result:");
              printf("\n -------------------------- ");
              printf("\n   t(k)     \t     y(k)     ");
              for(i = 0; i <= nInterval; i++ )
                     printf("\n %2.8lf \t  %2.8lf",pTPoints[i],pYPoints[i]);
              printf("\n -------------------------- ");
       }
       /* Report any errors */
       switch(nStatus)
       {
       case -1:
              printf("\nMemory allocation error!!");
              break;
       default:
              break;
       }
       /* Free the allocation */
       if(pTPoints) free(pTPoints);
       if(pYPoints) free(pYPoints);
       printf("\nPress any key to continue....");
       getch();
       return 0;
}
/* Evaluates the function y' = f(t,y) */
double  functionOfTY(double nT,double nY)
{      return ((nT - nY) / 2); }
```

```
#include  "HeunsFunc.h"

/* Function name       : computeHeunsMethod
* Description         : Computes the initial value problem y' = f(t,y).
* Parameter(s)        :
*     nLowerLimit         [in]  The lower limit.
*     nUpperLimit         [in]  The upper limit.
*     nInitYValue         [in]  The initial value Y(0).
*     nInterval           [in]  The interval M.
*     pFuncOfTY           [in]  The function y' = f(t,y).
*     *pTPoints           [out] Stores the values of t(k).
*     *pYPoints           [out] Stores the values of y(k).
* Return              :
*    int           - 0  Success.
*/
int _cdecl
computeHeunsMethod(double nLowerLimit,double nUpperLimit,double nInitYValue,int
nInterval,double (*pFuncOfTY)(double t,double y),double *pTPoints,double *pYPoints)
{
    int nRet = 0;

    int     i;
    double  nWeightH;
    double  nYPrime1;   /* the value evaluated by f(t(i),y(i)) */
    double  nYPrime2;   /* the value evaluated by f(t(i+1),nP1) */
    double  nP1;        /* The y(i) +hf(t(i),y(i)) */

    /* Clear the pTPoints && pYPoints. */
    memset(pTPoints,0,(nInterval + 1) * sizeof(pTPoints[0]));
    memset(pYPoints,0,(nInterval + 1) * sizeof(pTPoints[0]));
    nWeightH = (nUpperLimit - nLowerLimit) / (double)nInterval;

    /* Set the lower limit is the initial value of t */
    pTPoints[0] = nLowerLimit;
    pYPoints[0] = nInitYValue;
    /* Start the iteration using the Euler's Method */
    for(i = 0; i < nInterval; i++)
    {

        nYPrime1 = pFuncOfTY(pTPoints[i],pYPoints[i]);
        nP1      = pYPoints[i] + nWeightH * nYPrime1;

        pTPoints[i+1] = nLowerLimit + nWeightH * (i+1);

        nYPrime2 = pFuncOfTY(pTPoints[i+1],nP1);

        pYPoints[i+1] = pYPoints[i] + nWeightH * (nYPrime1 + nYPrime2 ) / 2;

    }

    return nRet;
}
```

Program Output

```
***  Heuns's Method ***

y' =  (t - y)/2

Enter the lower limit: 0
Enter the upper limit: 3
Enter the interval M  [ > 0]: 24
Enter the initial value y(0): 1

Result:
---------------------------
  t(k)              y(k)
0.00000000        1.00000000
0.12500000        0.94335938
0.25000000        0.89771652
0.37500000        0.86240556
0.50000000        0.83680093
0.62500000        0.82031493
0.75000000        0.81239547
0.87500000        0.81252387
1.00000000        0.82021286
1.12500000        0.83500466
1.25000000        0.85646922
1.37500000        0.88420253
1.50000000        0.91782503
1.62500000        0.95698016
1.75000000        1.00133292
1.87500000        1.05056862
2.00000000        1.10439162
2.12500000        1.16252416
2.25000000        1.22470531
2.37500000        1.29068995
2.50000000        1.36024778
2.62500000        1.43316247
2.75000000        1.50923076
2.87500000        1.58826171
3.00000000        1.67007594
---------------------------
Press any key to continue....
```

Taylor's Method of Order 4

Discussion

The Taylor series method has been commonly used in deriving polynomials that are used in obtaining numerically approximated values. One of which is the Newton's method for solving the root of nonlinear equations and systems. Its usage also extends in solving for an I.V.P of a differential equation. In fact it can be devised to have any specified degree of accuracy as long as one can obtain the derivative of the desired order.

The general step for Taylor's method of order N to approximate the numerical solution to the I.V.P. $y'(t) = f(t, y)$ over $[t_0, t_M]$ on each subinterval $[t_k, t_{k+1}]$ is:

$$y_{k+1} = y_k + d_1 h + \frac{d_2 h^2}{2!} + \frac{d_3 h^3}{3!} + \ldots + \frac{d_N h^N}{N!}, \tag{1}$$

where $d_j = y^{(j)}(t_k)$ for j=1,2,....N at each step k=0,1,.....M-1. Thus, Taylor's method of order N = 4 would have:

$$y_{k+1} = y_k + d_1 h + \frac{d_2 h^2}{2!} + \frac{d_3 h^3}{3!} + \frac{d_4 h^4}{4!}. \tag{2}$$

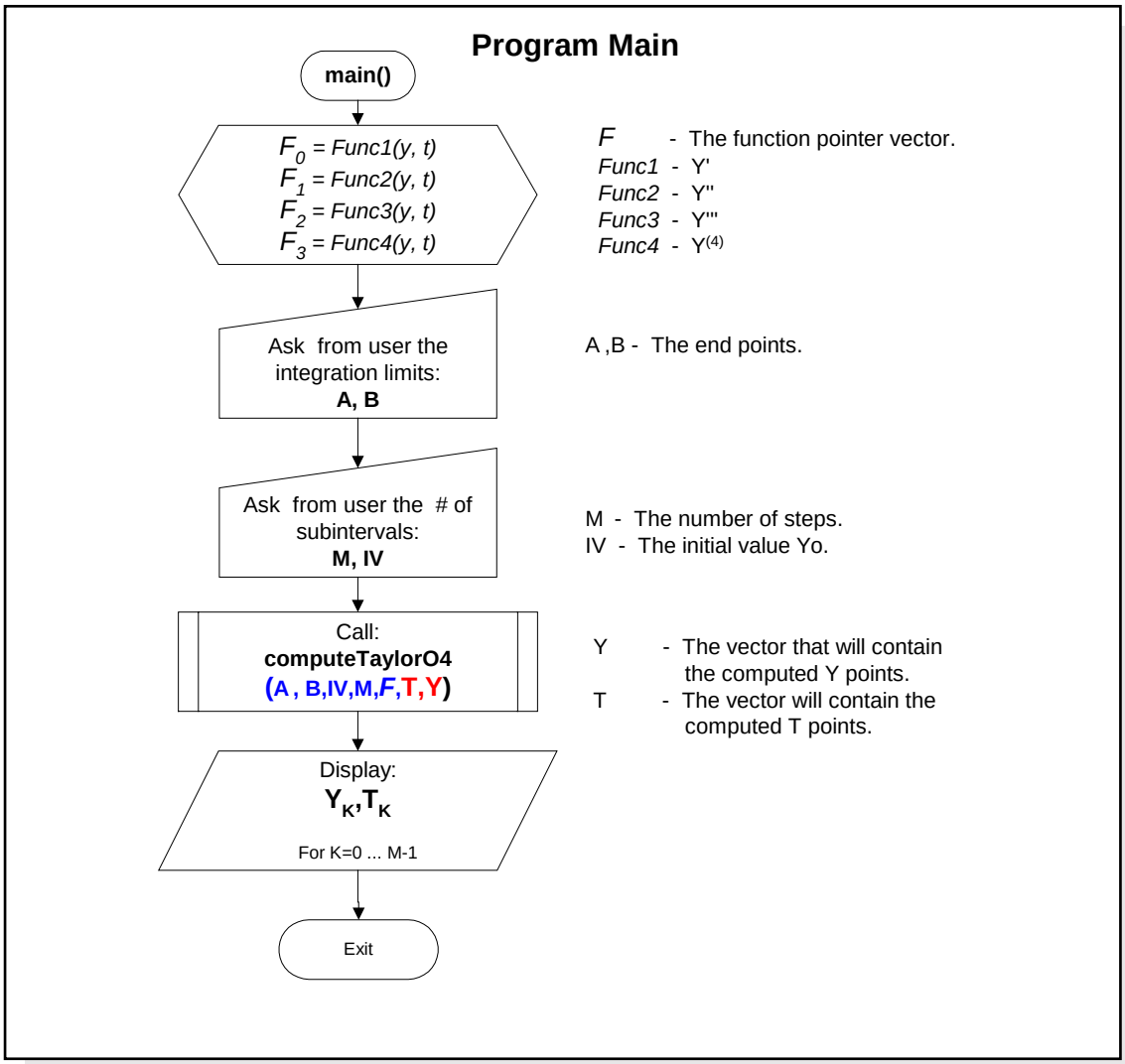
For this method, the final global error (F.G.E), at the end of the interval is defined by:

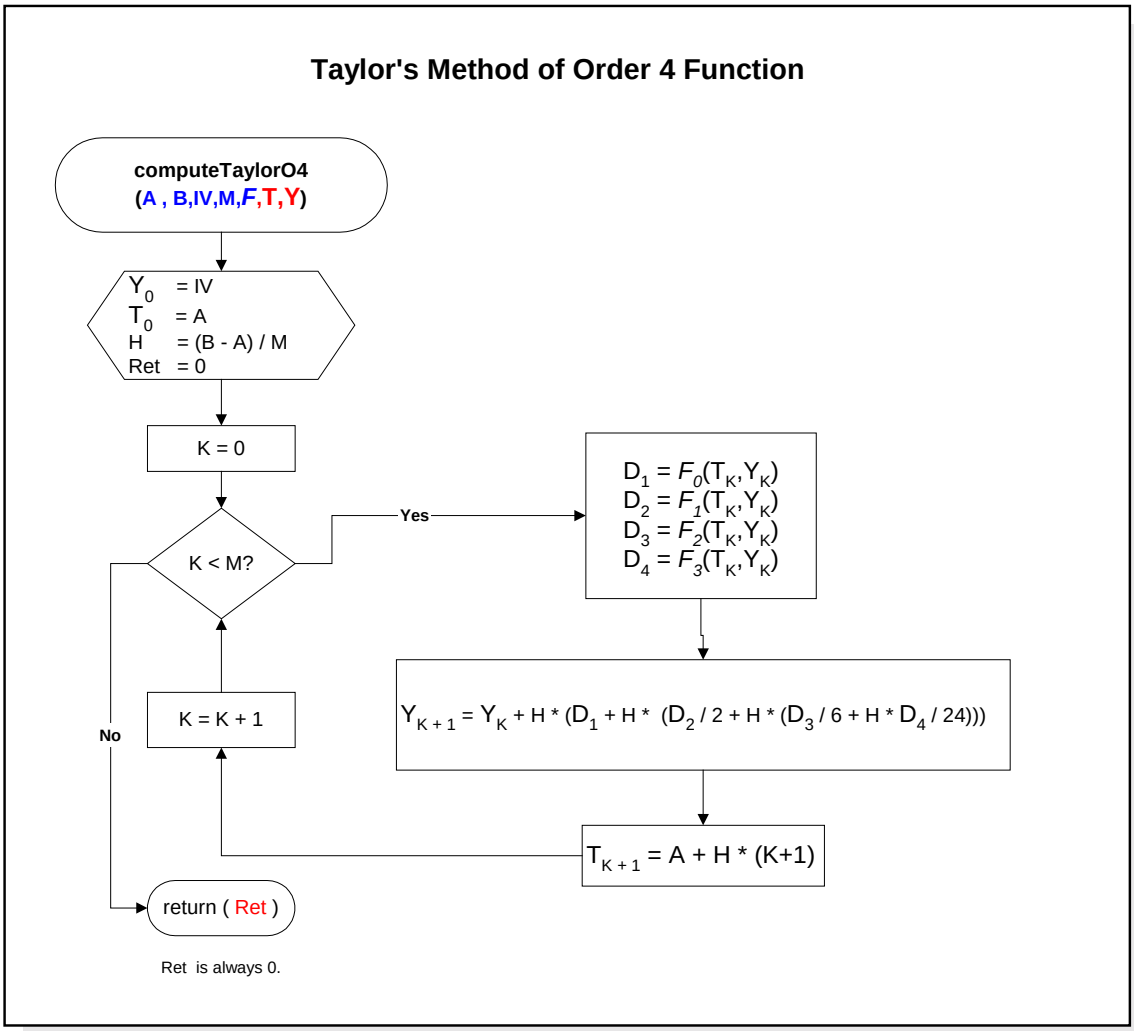
$$E(y(b), h) = \left| y(b) - yM \right| = O(h^N). \tag{3}$$

Explicit discussions in obtaining the F.G.E can be found on page 445 of the textbook.

Algorithm 9.3 (Taylor's Method of Order 4). To approximate the solution of the initial value problem $y' = f(t, y)$ with $y(a) = y_0$ over $[a, b]$ by evaluating y'', y''', and y'''' and using the Taylor polynomial at each step.

Program Flow




Taylor's Method of Order 4 Function
computeTaylorO4
(A , B,IV,M,F,T,Y)
Y0 = IV
T0 = A
H = (B - A) / M
Ret = 0
K = 0
K < M?
Yes
No
D1 = F0(TK,YK)
D2 = F1(TK,YK)
D3 = F2(TK,YK)
D4 = F3(TK,YK)
YK + 1 = YK + H * (D1 + H * (D2 / 2 + H * (D3 / 6 + H * D4 / 24)))
TK + 1 = A + H * (K+1)
K = K + 1
return (Ret)
Ret is always 0.

Program Source Code

```
#include  <memory.h>
#include  <stdio.h>
#include  <conio.h>
#include  "TaylorO4Func.h"

double  funcTY1(double nT,double nY); /* The function y'    */
double  funcTY2(double nT,double nY); /* The function y''   */
double  funcTY3(double nT,double nY); /* The function y'''  */
double  funcTY4(double nT,double nY); /* The function y(4)  */

int main(int argc, char* argv[])
{
        int nStatus = 0;
        int  i;
        double*  pTPoints = NULL;
        double*  pYPoints = NULL;
        int      nInterval;
        double   nLowerLimit,nUpperLimit, nInitYValue;
        FuncTY   funcTY[4];

        funcTY[0] = funcTY1; funcTY[1] = funcTY2; funcTY[2] = funcTY3;      funcTY[3] =
funcTY4;

        printf("\n ***  Taylor's Method of Order 4 *** \n");
        printf("\n y'   =  (t - y)/2        \n");
        printf("\n y''  =  (2 - t + y)/ 4   \n");
        printf("\n y''' =  (-2 + t - y)/ 8  \n");
        printf("\n y(4) =  (2 - t + y)/ 16  \n\n");

        /*  This is the input stage */
        printf(" Enter the lower limit: ");
        scanf("%lf",&nLowerLimit);    fflush(stdin);
        printf(" Enter the upper limit: ");
        scanf("%lf",&nUpperLimit);    fflush(stdin);

        /*  Ask for the interval*/
        do{
                printf(" Enter the interval M  [ > 0]: ");
                scanf("%d",&nInterval);       fflush(stdin);
        }while(nInterval <= 0);

        /*  Ask for the initial value y(0)*/
        printf(" Enter the initial value y(0): ");
        scanf("%lf",&nInitYValue);    fflush(stdin);

        pTPoints = (double*) malloc((nInterval + 1) * sizeof(pTPoints[0]));
        pYPoints = (double*) malloc((nInterval + 1) * sizeof(pYPoints[0]));

        if(!pTPoints || !pYPoints)
                nStatus = -1;

        if(nStatus == 0)
        {

        computeTaylorO4(nLowerLimit,nUpperLimit,nInitYValue,nInterval,funcTY,pTPoints,pYPo
ints);

                printf("\n Result:");
                printf("\n ------------------------- ");
                printf("\n   t(k)     \t     y(k)     ");
```

```
                for(i = 0; i <= nInterval; i++ )
                        printf("\n %2.8lf \t  %2.8lf",pTPoints[i],pYPoints[i]);

                printf("\n ------------------------- ");
        }

        /* Report any errors */
        switch(nStatus)
        {
        case -1:
                printf("\nMemory allocation error!!");
                break;
        default:
                break;
        }

        /* Free the allocation */
        if(pTPoints) free(pTPoints);
        if(pYPoints) free(pYPoints);
        printf("\nPress any key to continue....");
        getch();
        return 0;
}

/* Evaluates the function y'  */
double  funcTY1(double nT,double nY)
{       return ((nT - nY) / 2); }

/* Evaluates the function y'' */
double  funcTY2(double nT,double nY)
{       return ((2 - nT + nY)/4); }

/* Evaluates the function y'''*/
double  funcTY3(double nT,double nY)
{       return ((-2 + nT - nY) / 8); }

/* Evaluates the function y(4) */
double  funcTY4(double nT,double nY)
{       return ((2 - nT + nY)/16); }
```

```
#include  "TaylorO4Func.h"

/* Function name      : computeTaylorO4
* Description        : Computes the initial value problem using Taylor Order 4.
* Parameter(s)       :
*     nLowerLimit          [in]  The lower limit.
*     nUpperLimit          [in]  The upper limit.
*     nInitYValue          [in]  The initial value Y(0).
*     nInterval            [in]  The interval M.
*     funcOfTY[]           [in]  The array of function y',y'',y''',y(4).
*     *pTPoints            [out] Stores the values of t(k).
*     *pYPoints            [out] Stores the values of y(k).
* Return             :
*    int           - 0  Success.
*/
int _cdecl
computeTaylorO4(double nLowerLimit,double nUpperLimit,double nInitYValue,int
nInterval,FuncTY funcOfTY[],double *pTPoints,double *pYPoints)
{
    int nRet = 0;

    int     i;
    double  nWeightH;

    double  nD1;  /* Result of y'   */
    double  nD2;  /* Result of y''  */
    double  nD3;  /* Result of y''' */
    double  nD4;  /* Result of y(4) */

    /* Clear the pTPoints && pYPoints. */
    memset(pTPoints,0,(nInterval + 1) * sizeof(pTPoints[0]));
    memset(pYPoints,0,(nInterval + 1) * sizeof(pTPoints[0]));
    nWeightH = (nUpperLimit - nLowerLimit) / (double)nInterval;
    /* Set the lower limit is the initial value of t */
    pTPoints[0] = nLowerLimit;
    pYPoints[0] = nInitYValue;
    /* Start the iteration using the Euler's Method */
    for(i = 0; i < nInterval; i++)
    {
        nD1 = funcOfTY[0](pTPoints[i],pYPoints[i]);  /* y'   */
        nD2 = funcOfTY[1](pTPoints[i],pYPoints[i]);  /* y''  */
        nD3 = funcOfTY[2](pTPoints[i],pYPoints[i]);  /* y''' */
        nD4 = funcOfTY[3](pTPoints[i],pYPoints[i]);  /* y(2) */

        pYPoints[i + 1] = pYPoints[i] +
                          nWeightH * (nD1 +
                          nWeightH * (nD2 / 2 + nWeightH * (nD3 / 6 +
                          nWeightH * nD4 / 24)));

        pTPoints[i + 1] = pTPoints[0] + nWeightH * (i+1);

    }

    return nRet;
}
```

Program Output

```
***  Taylor's Method of Order 4 ***

y'   =  (t - y)/2

y''  =  (2 - t + y)/ 4

y''' =  (-2 + t - y)/ 8

y(4) =  (2 - t + y)/ 16

Enter the lower limit: 0
Enter the upper limit: 3
Enter the interval M  [ > 0]: 24
Enter the initial value y(0): 1

Result:
--------------------------
  t(k)              y(k)
0.00000000       1.00000000
0.12500000       0.94323921
0.25000000       0.89749075
0.37500000       0.86208742
0.50000000       0.83640243
0.62500000       0.81984698
0.75000000       0.81186794
0.87500000       0.81194569
1.00000000       0.81959210
1.12500000       0.83434860
1.25000000       0.85578442
1.37500000       0.88349487
1.50000000       0.91709980
1.62500000       0.95624208
1.75000000       1.00058621
1.87500000       1.04981703
2.00000000       1.10363847
2.12500000       1.16177241
2.25000000       1.22395755
2.37500000       1.28994845
2.50000000       1.35951453
2.62500000       1.43243919
2.75000000       1.50851893
2.87500000       1.58756259
3.00000000       1.66939061
--------------------------
Press any key to continue....
```

Runge-Kutta of Order 4

Discussion

Runge-Kutta of order 4 simulates the accuracy of Taylor's method of order 4. Even though the former is slower compared to the latter due to the several functions that need to be evaluated in each step, Runge-Kutta benefits anyone who uses it. This is because it doesn't require one to compute for the higher derivatives of the desired order, which is a must in the Taylor's method.

RK4 is based on computing y_{k+1} as follows:

$$y_{k+1} = y_k + w_1 k_1 + w_2 k_2 + w_3 k_3 + w_4 k_4 \tag{1}$$

where k_1, k_2, k_3, and $k4$ has the forms:

$$\begin{aligned} k_1 &= hf(t_k, y_k), \\ k_2 &= hf(t_k + a_1 b, y_k + b_1 k_1), \\ k_3 &= hf(t_k + a_2 b, y_k + b_2 k_1 + b_3 k_2), \\ k_4 &= hf(t_k + a_3 b, y_k + b_4 k_1 + b_5 k_2 + b_6 k_3), \end{aligned} \tag{2}$$

The equations and the values for the unknowns in (2) are shown on p. 451 of the textbook. With an initial point (t_0, y_0), one can generate the sequence of approximated values using:

$$y_{k+1} = y_k + h\frac{(f_1 + 2f_2 + 2f_3 + f_4)}{6} \tag{3}$$

where,

$$\begin{aligned} f_1 &= f(t_k, y_k), \\ f_2 &= f(t_k + \frac{h}{2}, y_k + \frac{h}{2} f_1), \\ f_3 &= f(t_k + \frac{h}{2}, y_k + \frac{h}{2} f_2), \\ f_4 &= f(t_k + h, y_k + hf_3). \end{aligned} \tag{4}$$

The development of RK4 is far too complicated for a novice in numerical methods that even the textbook does not cover its details. But it was shown that it applied the principles of the Simpson's rule from numerical integration with step size h/2 to obtain the approximation to the integral:

$$\int_{t_0}^{t_1} f(t, y(t))dt \approx \frac{h}{6}(f(t_0, y(t_0)) + 4f(t_{1/2}, y(t_{1/2})) + f(t_1, y(t_1))). \tag{5}$$

And by relating each term in (3) against (5) we will have:

$$\begin{aligned} f(t_0, y(t_0)) &= f_1, \\ f(t_1, y(t_1)) &\approx f_4, \\ f(t_{1/2}, y(t_{1/2})) &\approx \frac{f_2 + f_3}{2}, \end{aligned} \tag{6}$$

Then (3) can be rewritten as:

$$y_{k+1} = y_k + \frac{h}{6}\left[f_1 + \frac{4(f_2 + f_3)}{2} + f_4 \right]. \tag{7}$$

More details are discussed on pp. 452-453 of the textbook.

Since this process utilizes the Simpson's rule, then the error term for the said method can be applied in getting the order of the accumulated error. Given that this is a simulation to the Taylor's method of order 4, it is but natural that the order of the F.G.E. will be:

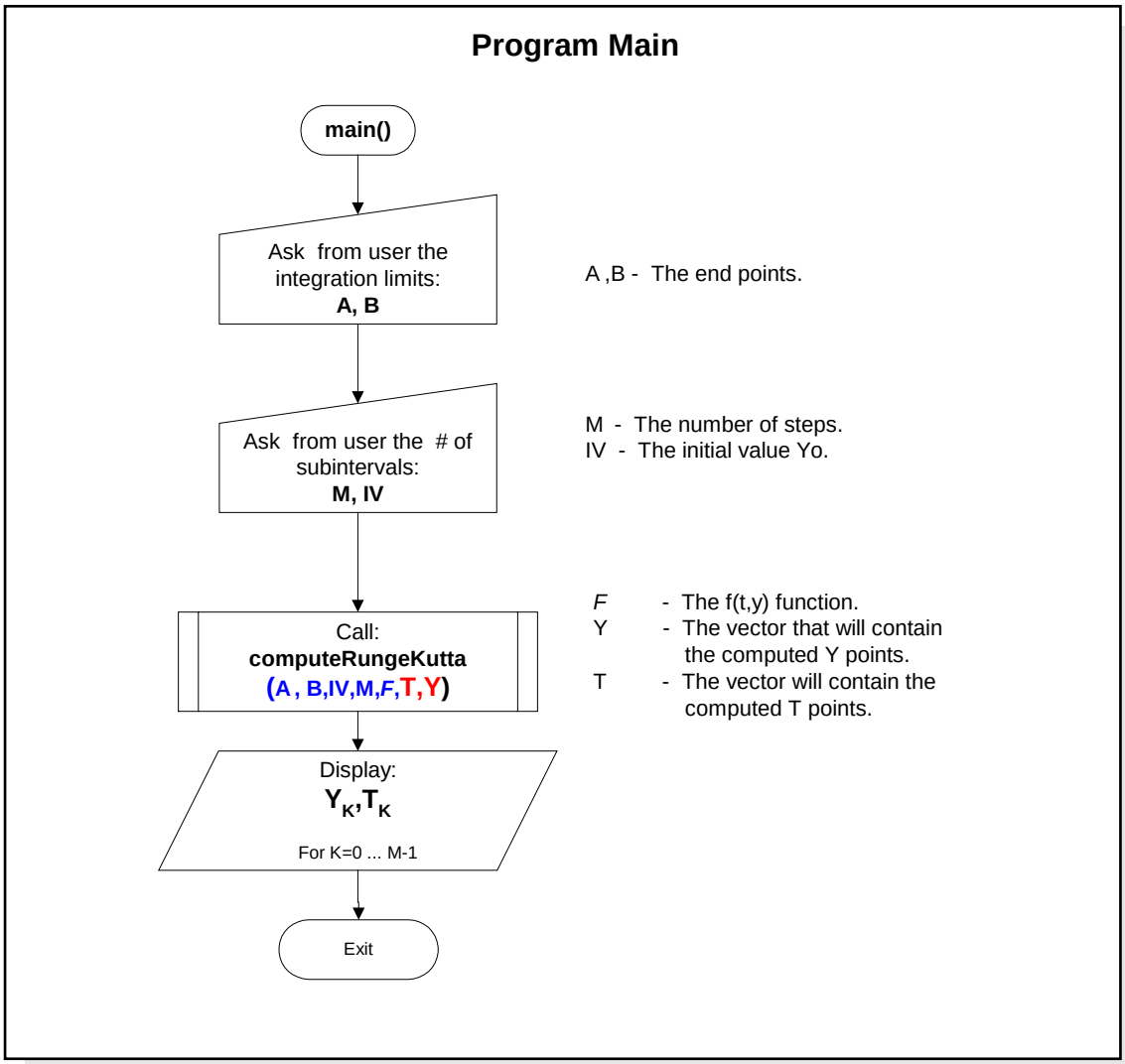
$$E(y(b), h) = |y(b) - yM| = O(h^4). \tag{8}$$

Details for obtaining the F.G.E are discussed on pp. 453-454 of the textbook.

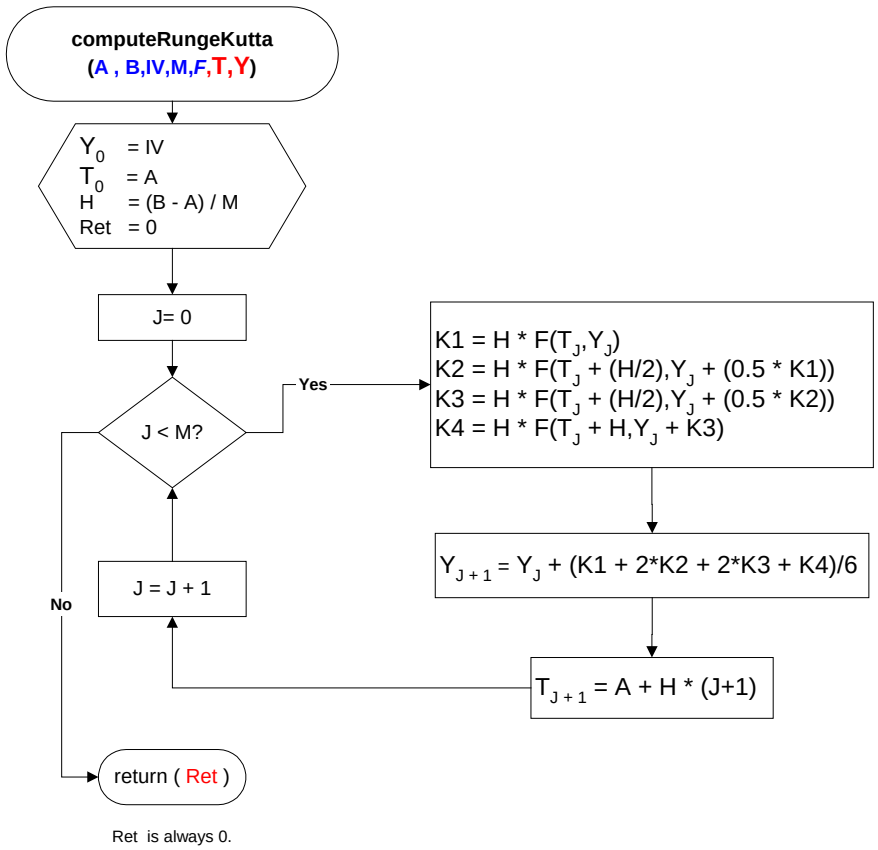
Algorithm 9.4 (Runge-Kutta Method of Order 4). To approximate the solution of the initial value problem $y' = f(t, y)$ with $y(a) = y_0$ over $[a, b]$ by using the formula

$$y_{k+1} = y_k + \frac{h}{6}[K_1 + 2K_2 + 2K_3 + K_4].$$

Program Flow



Runge-Kutta Method of Order 4 Function



Ret is always 0.

Program Source Code

```
#include  <memory.h>
#include  <stdio.h>
#include  <conio.h>
#include  "RungeKuttaFunc.h"
double  funcTY(double nT,double nY); /* The function y'   */
int main(int argc, char* argv[])
{
    int nStatus = 0;
    int  i;
    double*  pTPoints = NULL;
    double*  pYPoints = NULL;
    int      nInterval;
    double   nLowerLimit;
    double   nUpperLimit;
    double   nInitYValue;
    printf("\n ***  Runge-Kutta Method of Order 4 *** \n");
    printf("\n y'   =  (t - y)/2        \n\n");
    /*  This is the input stage */
    printf(" Enter the lower limit: ");
    scanf("%lf",&nLowerLimit);   fflush(stdin);
    printf(" Enter the upper limit: ");
    scanf("%lf",&nUpperLimit);   fflush(stdin);
    do{
        printf(" Enter the interval M  [ > 0]: ");
        scanf("%d",&nInterval);    fflush(stdin);
    }while(nInterval <= 0);
    /*  Ask for the initial value y(0)*/
    printf(" Enter the initial value y(0): ");
    scanf("%lf",&nInitYValue);   fflush(stdin);
    pTPoints = (double*) malloc((nInterval + 1) * sizeof(pTPoints[0]));
    pYPoints = (double*) malloc((nInterval + 1) * sizeof(pYPoints[0]));
    if(!pTPoints || !pYPoints)    nStatus = -1;
    if(nStatus == 0)
    {
computeRungeKutta(nLowerLimit,nUpperLimit,nInitYValue,nInterval,funcTY,pTPoints,pYPoints)
;
        printf("\n Result:");
        printf("\n ------------------------- ");
        printf("\n   t(k)     \t     y(k)     ");
        for(i = 0; i <= nInterval; i++ )
            printf("\n %2.8lf \t  %2.8lf",pTPoints[i],pYPoints[i]);

        printf("\n ------------------------- ");
    }
    /* Report any errors */
    switch(nStatus)
    {
    case -1:
        printf("\nMemory allocation error!!");
        break;
    default:
        break;
    }
    /* Free the allocation */
    if(pTPoints)    free(pTPoints);
    if(pYPoints)    free(pYPoints);
    printf("\nPress any key to continue....");
    getch();
    return 0;
}
/* Evaluates the function y'  */
double  funcTY(double nT,double nY)
{    return ((nT - nY) / 2); }
```

```
#include  "RungeKuttaFunc.h"

/* Function name      : computeRungeKutta
* Description        : Computes the initial value problem y' = f(t,y). using Rung-Kutta
of Order 4.
* Parameter(s)       :
*     nLowerLimit     [in]  The lower limit.
*     nUpperLimit     [in]  The upper limit.
*     nInitYValue     [in]  The initial value Y(0).
*     nInterval       [in]  The interval M.
*     pFuncOfTY       [in]  The function y' = f(t,y).
*     *pTPoints       [out] Stores the values of t(k).
*     *pYPoints       [out] Stores the values of y(k).
*
* Return             :
*    int           - 0  Success.
*/
int _cdecl
computeRungeKutta(double nLowerLimit,double nUpperLimit,double nInitYValue,int
nInterval,double (*pFuncOfTY)(double,double),double *pTPoints,double *pYPoints)
{
    int nRet = 0;

    int     i;
    double  nWeightH;
    double  nK1;  /* Result of f(t,y) @ ti     */
    double  nK2;  /* Result of f(t,y) @ ti.5   */
    double  nK3;  /* Result of f(t,y) @ ti+.5  */
    double  nK4;  /* Result of f(t,y) @ ti+1   */

    /* Clear the pTPoints && pYPoints. */
    memset(pTPoints,0,(nInterval + 1) * sizeof(pTPoints[0]));
    memset(pYPoints,0,(nInterval + 1) * sizeof(pTPoints[0]));

    nWeightH = (nUpperLimit - nLowerLimit) / (double)nInterval;
    /* Set the lower limit is the initial value of t */
    pTPoints[0] = nLowerLimit;
    pYPoints[0] = nInitYValue;
    /* Start the iteration using the Euler's Method */
    for(i = 0; i < nInterval; i++)
    {
        nK1 = nWeightH * pFuncOfTY(pTPoints[i],pYPoints[i]);
/* f(t,y) @ ti       */
        nK2 = nWeightH * pFuncOfTY(pTPoints[i] + (nWeightH/2),pYPoints[i] + (0.5 * nK1));
/* f(t,y) @ ti + .5  */
        nK3 = nWeightH * pFuncOfTY(pTPoints[i] + (nWeightH/2),pYPoints[i] + (0.5 * nK2));
/* f(t,y) @ ti + .5  */
        nK4 = nWeightH * pFuncOfTY(pTPoints[i] + nWeightH,pYPoints[i] + nK3);
/* f(t,y) @ ti+1     */

        pYPoints[i + 1] = pYPoints[i] + (nK1 + 2*nK2 + 2*nK3 + nK4)/6;
        pTPoints[i + 1] = pTPoints[0] + nWeightH * (i+1);

    }

    return nRet;
}
```

Program Output

```
***  Runge-Kutta Method of Order 4 ***

y'    =  (t - y)/2

Enter the lower limit: 0
Enter the upper limit: 3
Enter the interval M  [ > 0]: 24
Enter the initial value y(0): 1

Result:
--------------------------
  t(k)              y(k)
0.00000000        1.00000000
0.12500000        0.94323921
0.25000000        0.89749075
0.37500000        0.86208742
0.50000000        0.83640243
0.62500000        0.81984698
0.75000000        0.81186794
0.87500000        0.81194569
1.00000000        0.81959210
1.12500000        0.83434860
1.25000000        0.85578442
1.37500000        0.88349487
1.50000000        0.91709980
1.62500000        0.95624208
1.75000000        1.00058621
1.87500000        1.04981703
2.00000000        1.10363847
2.12500000        1.16177241
2.25000000        1.22395755
2.37500000        1.28994845
2.50000000        1.35951453
2.62500000        1.43243919
2.75000000        1.50851893
2.87500000        1.58756259
3.00000000        1.66939061
--------------------------
Press any key to continue....
```

Adams-Bashforth-Moulton Method

Discussion

Another way in solving for I.V.P which involves the use of not one but four initial points in obtaining a new point, is the **Adams-Bashforth-Moulton method**. The preceding methods of Euler, Heun, Taylor, and Runge-Kutta are termed as **single-step** since they only require single previous information in computing for the next one. That is, in order for one to get the value of y_{k+1}, one must have the value of y_k beforehand. However in the Adams-Bashforth-Moulton method**,** one is required to have the values for y_{k-3}, y_{k-2}, y_{k-1}, and y_k in order to obtain y_{k+1}. This is why it is called a four-step method.

The principle of Adams-Bashforth-Moulton is based on the fundamental theorem of calculus and is a little more similar to Heun's method as it also has two parts, the Adams-Bashforth predictor and the Adams-Moulton corrector.

$$p_{k+1} = y_k + \frac{h}{24}(-9 f_{k-3} + 37 f_{k-2} - 59 f_{k-1} + 55 f_k). \tag{1}$$

$$y_{k+1} = y_k + \frac{h}{24}(f_{k-2} - 5 f_{k-1} - 19 f_k + 9 f_{k+1}). \tag{2}$$

Both of these equations were obtained from the Lagrange polynomial approximation where (1) extrapolates to obtain the value for p_{k+1} while (2) interpolates to have the corrected value which is y_{k+1}. Details on the predictor and corrector can be found on page 464 of the textbook.

The L.T.E's for both (1) and (2) are of order $\boldsymbol{O(h^5)}$ which are:

$$y(t_{k+1}) - p_{k+1} = \frac{251}{720} y^{(5)}(c_{k+1}) h^5 \quad \text{for the predictor and} \tag{3}$$

$$y(t_{k+1}) - y_{k+1} = \frac{-19}{720} y^{(5)}(d_{k+1}) h^5 \quad \text{for the corrector.} \tag{4}$$

With a small value for h and with $y^{(5)}(t)$ nearly constant over the interval, then the terms involving $y^{(5)}(t)$ in (3) and (4) can be eliminated and the resulting formula will be:

$$y(t_{k+1}) - y_{k+1} \approx \frac{-19}{270}(y_{k+1} - p_{k+1}) \tag{5}$$

which can be used to obtain the approximate error estimate from p_{k+1}, y_{k+1} that doesn't use $y^{(5)}(t)$. One may benefit the use of (5) by adjusting the step size whenever the accuracy doesn't meet the prescribed error. Details on error estimation and correction can be found on pp. 465-466 of the textbook.

Actual implementation of this algorithm doesn't need one to provide the four initial points. This has been modified to create the three additional points using the Runge-Kutta method from a single initial point y_0 before the four-step method starts.

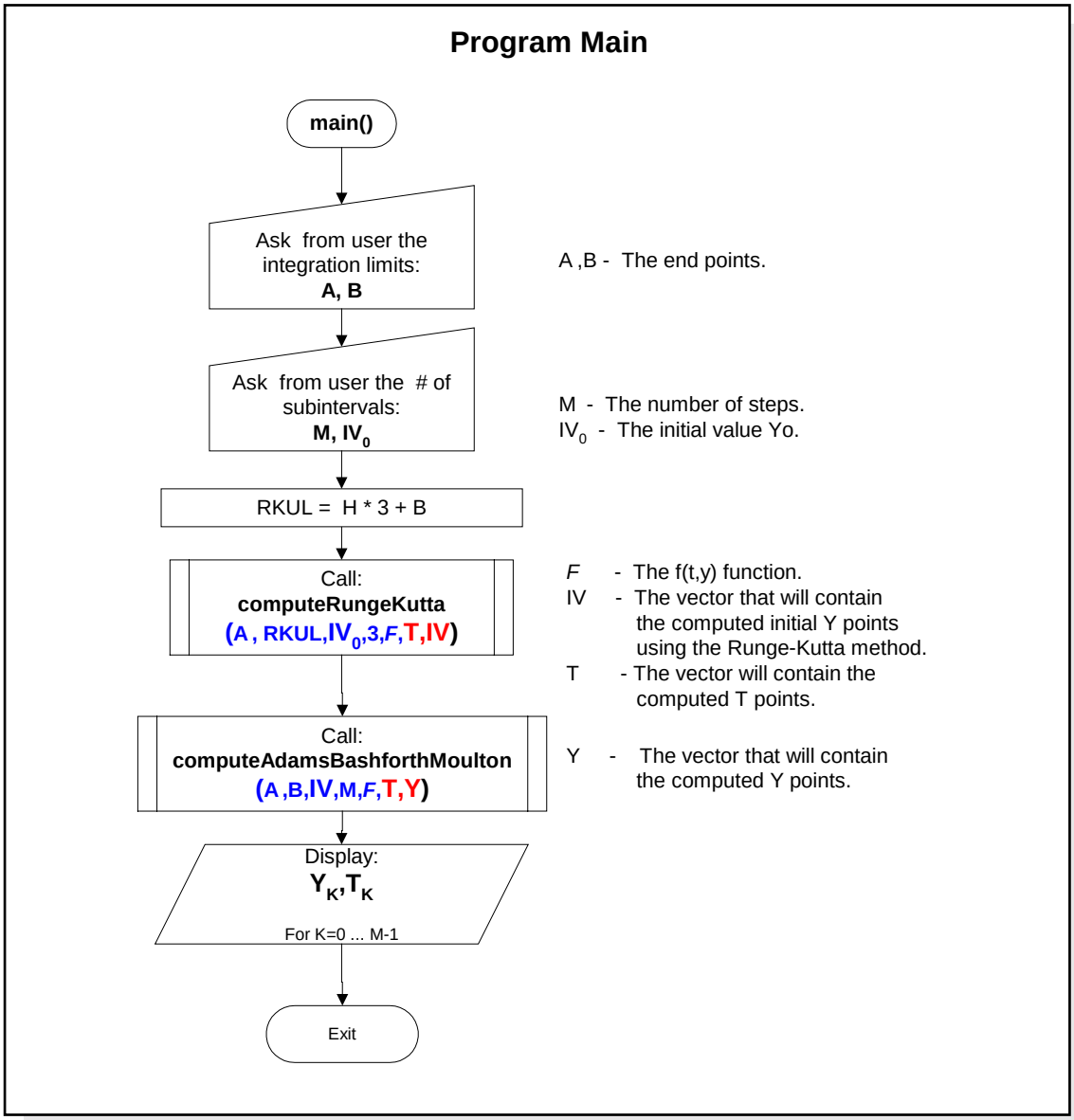
Algorithm 9.6 (Adams-Bashforth-Moulton Method). To approximate the solution of the initial value problem $y' = f(t, y)$ with $y(a) = y_0$ over $[a, b]$ by using predictor

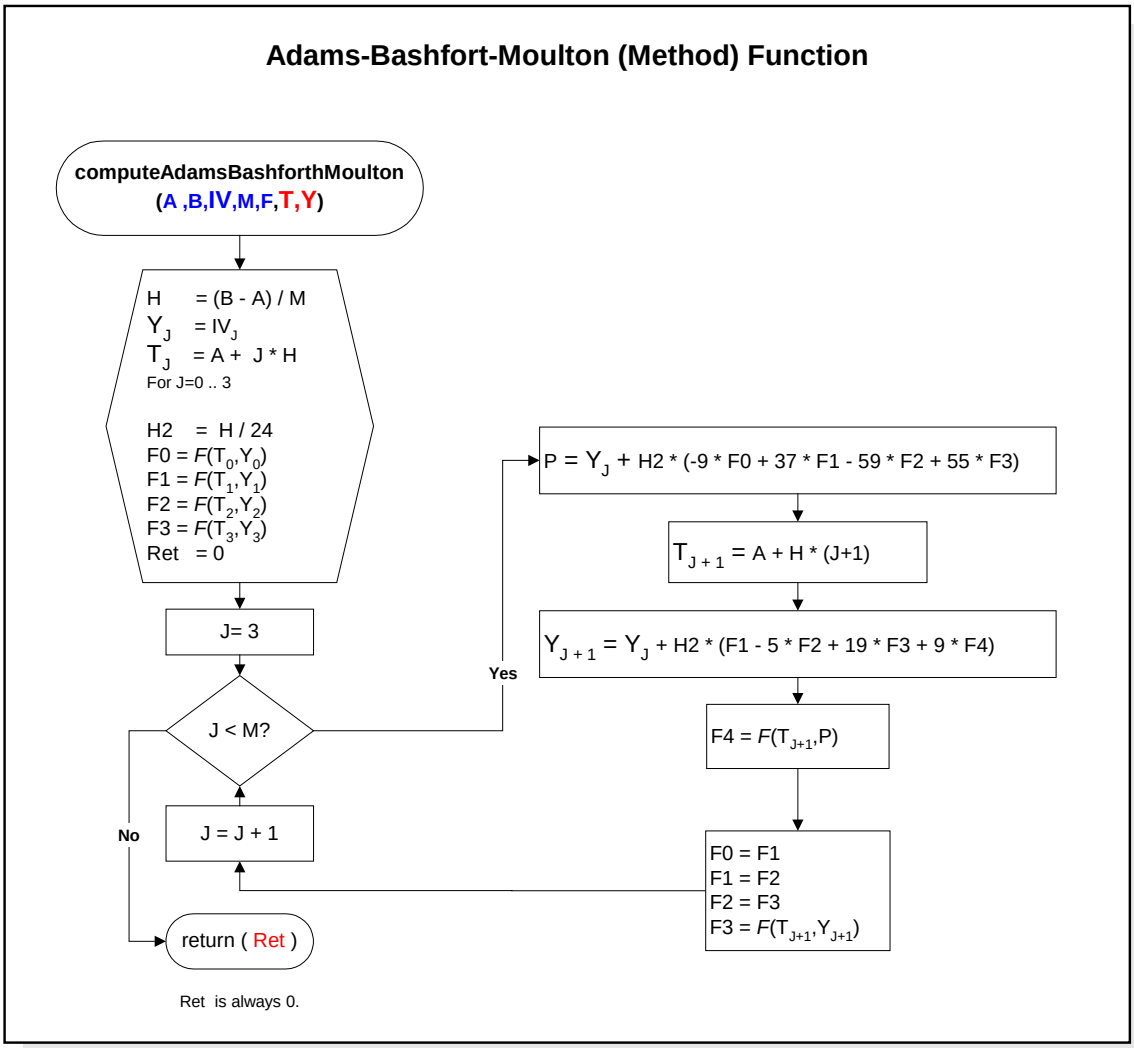
$$p_{k+1} = y_k + \frac{h}{24}(-9 f_{k-3} + 37 f_{k-2} - 59 f_{k-1} + 55 f_k).$$

and the corrector

$$y_{k+1} = y_k + \frac{h}{24}(f_{k-2} - 5 f_{k-1} - 19 f_k + 9 f_{k+1}).$$

Program Flow




Adams-Bashfort-Moulton (Method) Function
computeAdamsBashforthMoulton
(A ,B,IV,M,F,T,Y)
H = (B - A) / M
Y_J = IV_J
T_J = A + J * H
For J=0 .. 3
H2 = H / 24
F0 = F(T_0,Y_0)
F1 = F(T_1,Y_1)
F2 = F(T_2,Y_2)
F3 = F(T_3,Y_3)
Ret = 0
J= 3
J < M?
Yes
No
J = J + 1
return (Ret)
Ret is always 0.
P = Y_J + H2 * (-9 * F0 + 37 * F1 - 59 * F2 + 55 * F3)
T_{J+1} = A + H * (J+1)
Y_{J+1} = Y_J + H2 * (F1 - 5 * F2 + 19 * F3 + 9 * F4)
F4 = F(T_{J+1},P)
F0 = F1
F1 = F2
F2 = F3
F3 = F(T_{J+1},Y_{J+1})

Program Source Code

```
#include  <memory.h>
#include  <stdio.h>
#include  <conio.h>
#include  "AdamsFunc.h"
#include  "RungeKuttaFunc.h"

double  funcTY(double nT,double nY); /* The function y'   */

int main(int argc, char* argv[])
{
       int nStatus = 0;
       int  i;
       double*  pTPoints = NULL;
       double*  pYPoints = NULL;
       int      nInterval;
       double   nLowerLimit,nUpperLimit,nRungeKuttaULimit;
       double   nInitYValue[4],nInitTValue[4],nWeightH;

       printf("\n ***  Adams-Bashforth-Moulton Method *** \n");
       printf("\n y'   =  (t - y)/2        \n\n");
       /*  This is the input stage */
       printf(" Enter the lower limit: ");
       scanf("%lf",&nLowerLimit);    fflush(stdin);
       printf(" Enter the upper limit: ");
       scanf("%lf",&nUpperLimit);    fflush(stdin);
       /*  Ask for the interval*/
       do{
              printf(" Enter the interval M  [ > 3]: ");
              scanf("%d",&nInterval); fflush(stdin);
       }while(nInterval <= 3);

       nWeightH = (nUpperLimit - nLowerLimit) / (double)nInterval;
       nRungeKuttaULimit = nWeightH * (double)3 + nLowerLimit;
       printf(" Enter the initial value y(0): ");
       scanf("%lf",&nInitYValue[0]); fflush(stdin);

       computeRungeKutta(nLowerLimit,nRungeKuttaULimit,nInitYValue[0],3,funcTY,nInitTValu
e,nInitYValue);

       printf(" The computed initial values using RK4: ");
       printf("\n -------------------------- ");
       printf("\n   t(k)     \t     y(k)     ");

       for(i = 0; i <= 3; i++)
              printf("\n %2.8lf \t  %2.8lf",nInitTValue[i],nInitYValue[i]);
       printf("\n -------------------------- ");

       pTPoints = (double*) malloc((nInterval + 1) * sizeof(pTPoints[0]));
       pYPoints = (double*) malloc((nInterval + 1) * sizeof(pYPoints[0]));

       if(!pTPoints || !pYPoints)    nStatus = -1;

       if(nStatus == 0)
       {

       computeAdamsBashforthMoulton(nLowerLimit,nUpperLimit,nInitYValue,nInterval,funcTY,
pTPoints,pYPoints);
              printf("\n Result:");
```

```
                printf("\n ------------------------- ");
                printf("\n   t(k)     \t     y(k)     ");
                for(i = 0; i <= nInterval; i++ )
                        printf("\n %2.8lf \t  %2.8lf",pTPoints[i],pYPoints[i]);
                printf("\n ------------------------- ");
        }

        /* Report any errors */
        switch(nStatus)
        {
        case -1:
                printf("\nMemory allocation error!!");
                break;
        default:
                break;
        }

        /* Free the allocation */
        if(pTPoints) free(pTPoints);
        if(pYPoints) free(pYPoints);
        printf("\nPress any key to continue....");
        getch();
        return 0;
}

/* Evaluates the function y'  */
double  funcTY(double nT,double nY)
{       return ((nT - nY) / 2); }
```

```
#include  "AdamsFunc.h"
/* Function name      : computeAdamsBashforthMoulton
* Description        : Computes the initial value problem y' = f(t,y). using
AdamsBashforthMoulton
* Parameter(s)       :
*     nLowerLimit         [in]  The lower limit.
*     nUpperLimit         [in]  The upper limit.
*     nInitYValue[]       [in]  The initial value Y(0) - Y(3).
*     nInterval           [in]  The interval M.
*     pFuncOfTY           [in]  The function y' = f(t,y).
*     *pTPoints           [out] Stores the values of t(k).
*     *pYPoints           [out] Stores the values of y(k).
* Return             :
*    int           - 0  Success.
*/
int _cdecl
computeAdamsBashforthMoulton(double nLowerLimit,double nUpperLimit,double
nInitYValue[],int nInterval,double (*pFuncOfTY)(double,double),double *pTPoints,double
*pYPoints)
{
    int nRet = 0;
    int     i;
    double  nWeightH,nWeightH2,nPredicted;
    double  nF0;  /* Result of f(ti,yi)      */
    double  nF1;  /* Result of f(ti+1,yi+1)  */
    double  nF2;  /* Result of f(ti+2,yi+2)  */
    double  nF3;  /* Result of f(ti+3,yi+3)  */
    double  nF4;  /* Result of f(ti+4,P)     */

    /* Clear the pTPoints && pYPoints. */
    memset(pTPoints,0,(nInterval + 1) * sizeof(pTPoints[0]));
    memset(pYPoints,0,(nInterval + 1) * sizeof(pTPoints[0]));
    nWeightH = (nUpperLimit - nLowerLimit) / (double)nInterval;
    for(i = 0; i <= 3; i++)
    {
        pTPoints[i] = nLowerLimit + nWeightH * i;
        pYPoints[i] = nInitYValue[i];
    }
    /* Get the respective results */
    nF0 = pFuncOfTY(pTPoints[0],pYPoints[0]);
    nF1 = pFuncOfTY(pTPoints[1],pYPoints[1]);
    nF2 = pFuncOfTY(pTPoints[2],pYPoints[2]);
    nF3 = pFuncOfTY(pTPoints[3],pYPoints[3]);
    nWeightH2 = nWeightH/24;
    /* Start the iteration using the Euler's Method */
    for(i = 3; i < nInterval; i++)
    {
        /* Compute the predicted value */
        nPredicted = pYPoints[i] + nWeightH2 * (-9 * nF0 + 37 * nF1 - 59 * nF2 + 55 *
nF3);
        pTPoints[i+1] = nLowerLimit + nWeightH * (i + 1);

        nF4 = pFuncOfTY(pTPoints[i+1],nPredicted);
        /* Compute the Y(i+1) */
        pYPoints[i + 1] = pYPoints[i] + nWeightH2 * (nF1 - 5 * nF2 + 19 * nF3 + 9 * nF4);

        /* Roll the values down */
        nF0 = nF1; nF1 = nF2;  nF2 = nF3;  nF3 = pFuncOfTY(pTPoints[i+1],pYPoints[i+1]);;

    }

    return nRet;
}
```

Program Output

```
***  Adams-Bashforth-Moulton Method ***

y'    =  (t - y)/2

Enter the lower limit: 0
Enter the upper limit: 3
Enter the interval M  [ > 3]: 24
Enter the initial value y(0): 1
The computed initial values using RK4:
-------------------------
  t(k)              y(k)
0.00000000       1.00000000
0.12500000       0.94323921
0.25000000       0.89749075
0.37500000       0.86208742
-------------------------
Result:
-------------------------
  t(k)              y(k)
0.00000000       1.00000000
0.12500000       0.94323921
0.25000000       0.89749075
0.37500000       0.86208742
0.50000000       0.83640232
0.62500000       0.81984678
0.75000000       0.81186767
0.87500000       0.81194535
1.00000000       0.81959170
1.12500000       0.83434815
1.25000000       0.85578392
1.37500000       0.88349434
1.50000000       0.91709924
1.62500000       0.95624149
1.75000000       1.00058560
1.87500000       1.04981640
2.00000000       1.10363784
2.12500000       1.16177176
2.25000000       1.22395690
2.37500000       1.28994780
2.50000000       1.35951389
2.62500000       1.43243854
2.75000000       1.50851829
2.87500000       1.58756196
3.00000000       1.66938999
-------------------------
Press any key to continue....
```

Milne-Simpson Method

Discussion

Like the Adams-Bashforth-Moulton method, Milne-Simpson is another multi-step process requiring the use of four initial points to obtain the value of the new point. It also utilizes the predictor-corrector scheme and is additionally based on the fundamental theorem of calculus.

The formula for the Milne predictor, which uses the Lagrange approximation and is integrated over the interval $[t_{k-3}, t_{k+1}]$ is:

$$p_{k+1} = y_{k-3} + \frac{4h}{3}(2f_{k-2} - f_{k-1} + 2f_k). \tag{1}$$

The formula for the corrector, which uses the Simpson's rule integrated over the interval $[t_{k-1}, t_{k+1}]$ is:

$$y_{k+1} = y_{k-1} + \frac{h}{3}(f_{k-1} + 4f_k + f_{k+1}). \tag{2}$$

Details on the predictor and corrector can be found on p. 467 of the textbook.

The L.T.E's for both (1) and (2) are of order $\boldsymbol{O(h^5)}$ which are:

$$y(t_{k+1}) - p_{k+1} = \frac{28}{90}y^{(5)}(c_{k+1})h^5 \quad \text{for the predictor and} \tag{3}$$

$$y(t_{k+1}) - y_{k+1} = \frac{-1}{90}y^{(5)}(d_{k+1})h^5 \quad \text{for the corrector.} \tag{4}$$

With a small value for h and with $y^{(5)}(t)$ (delete: is) nearly constant over the interval, then the terms involving $y^{(5)}(t)$ in (3) and (4) can be eliminated and the resulting formula is:

$$y(t_{k+1}) - y_{k+1} \approx \frac{28}{29}(y_{k+1} - p_{k+1}) \tag{5}$$

which can be used to obtain the error estimate for the predictor from the computed values p_{k+1}, y_{k+1} that doesn't use $y^{(5)}(t)$. To take advantage of (5), one can use it to improve the predicted value which will be called the modifier m_{k+1} :

$$m_{k+1} = p_{k+1} + 28\frac{y_k - p_k}{29}. \tag{6}$$

This modified value can then be used in place of p_{k+1} in (2) forming a new equation:

$$y_{k+1} = y_{k-1} + \frac{h}{3}(f_{k-1} + 4f_k + f(t_{k+1}, m_{k+1})). \tag{7}$$

Like the Adams-Bashforth-Moulton, actual implementation of this algorithm doesn't need one to provide the four initial points. This has been modified to create the three additional points using the Runge-Kutta method from a single initial point y_0 before the four-step method begins.

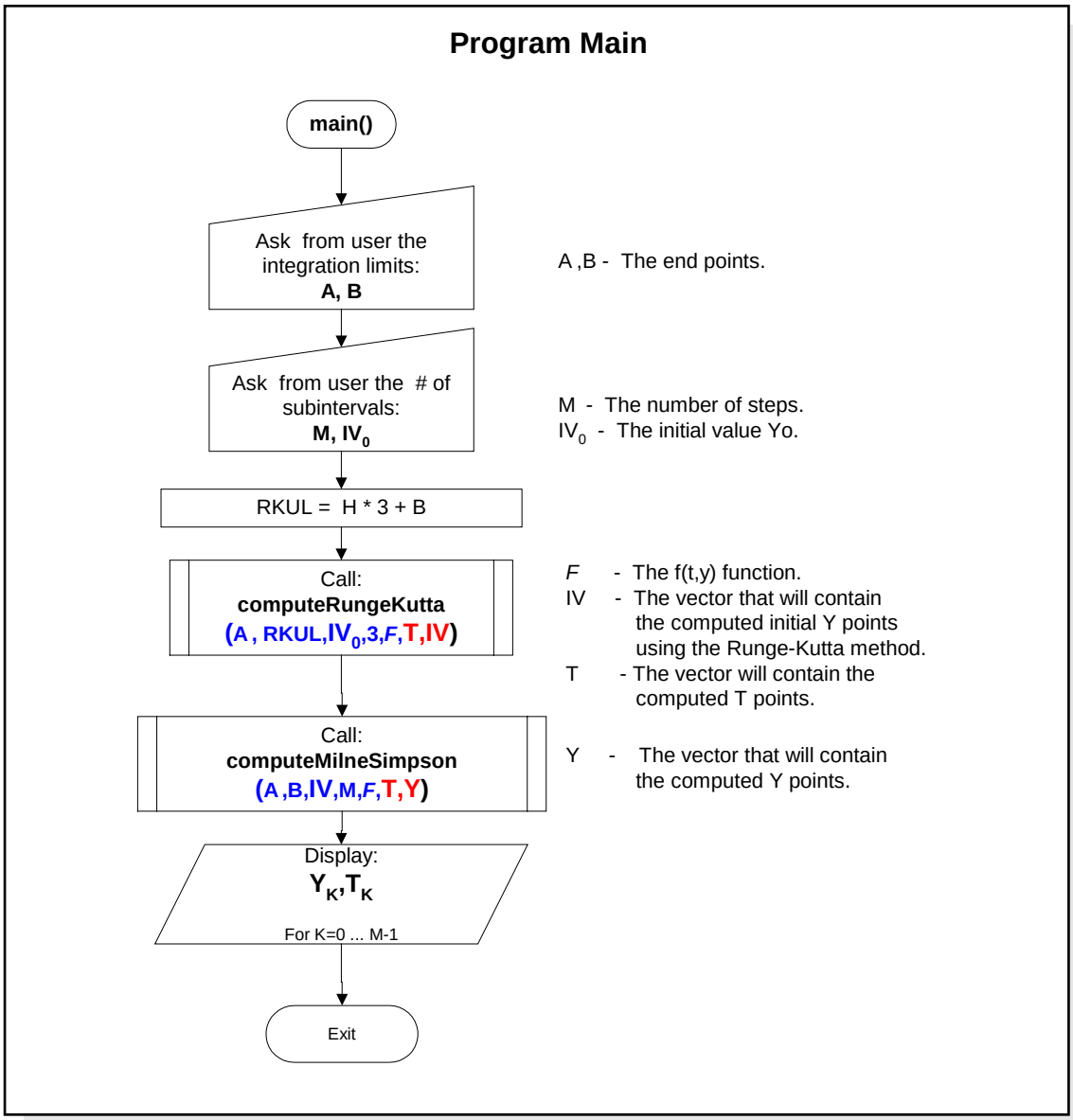
Algorithm 9.6 (Milne-Simpson Method). To approximate the solution of the initial value problem $y' = f(t, y)$ with $y(a) = y_0$ over $[a, b]$ by using the predictor

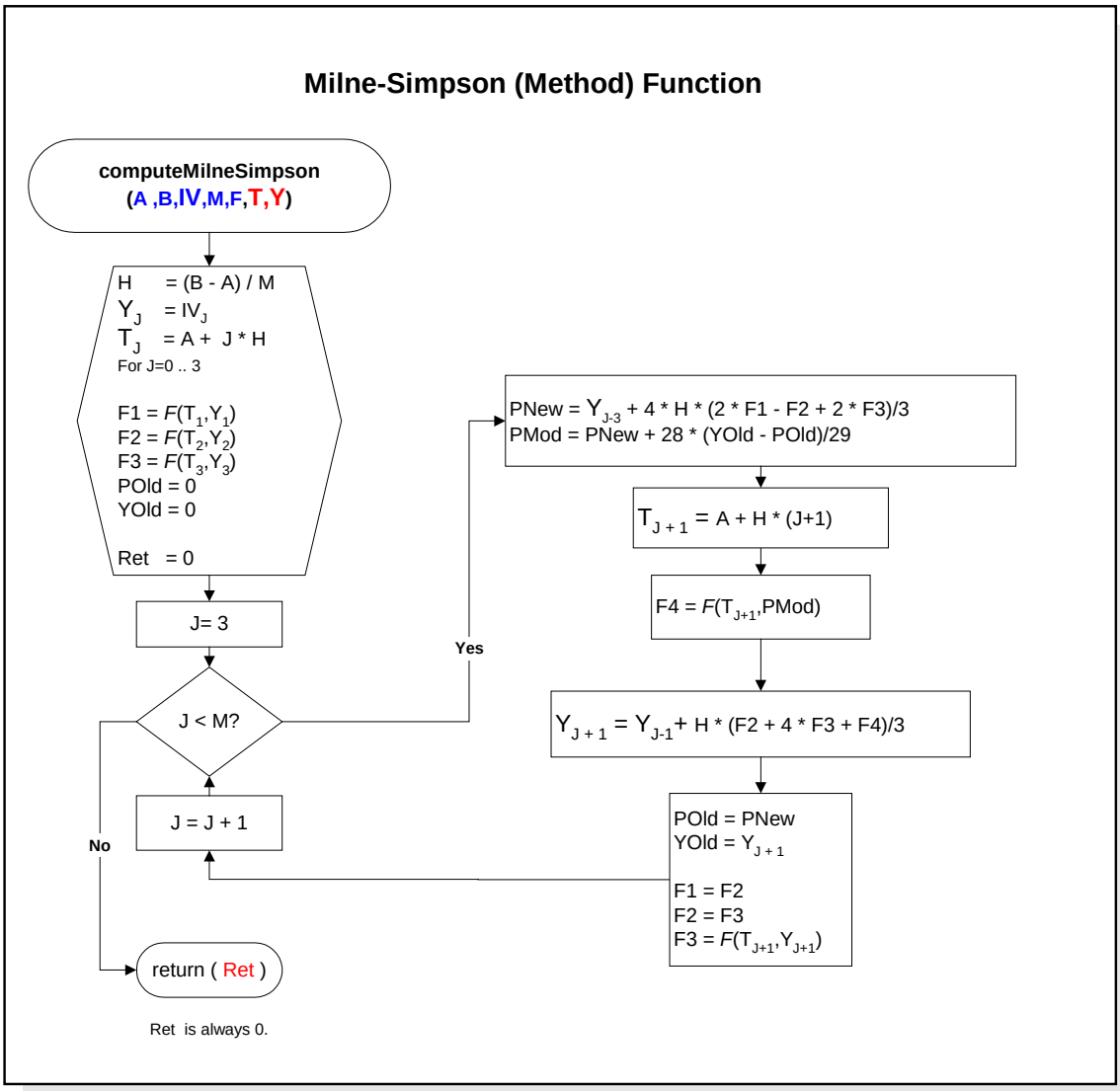
$$p_{k+1} = y_{k-3} + \frac{4h}{3}[2f_{k-2} - f_{k-1} + 2f_k]$$

and the corrector

$$y_{k+1} = y_{k-1} + \frac{h}{3}[f_{k-1} + 4f_k + f_{k+1}].$$

Program Flow




Milne-Simpson (Method) Function
computeMilneSimpson
(A ,B,IV,M,F,T,Y)
H = (B - A) / M
Y_J = IV_J
T_J = A + J * H
For J=0 .. 3
F1 = F(T_1,Y_1)
F2 = F(T_2,Y_2)
F3 = F(T_3,Y_3)
POld = 0
YOld = 0
Ret = 0
J= 3
J < M?
Yes
No
PNew = Y_{J-3} + 4 * H * (2 * F1 - F2 + 2 * F3)/3
PMod = PNew + 28 * (YOld - POld)/29
T_{J+1} = A + H * (J+1)
F4 = F(T_{J+1},PMod)
Y_{J+1} = Y_{J-1} + H * (F2 + 4 * F3 + F4)/3
POld = PNew
YOld = Y_{J+1}
F1 = F2
F2 = F3
F3 = F(T_{J+1},Y_{J+1})
J = J + 1
return (Ret)
Ret is always 0.

Program Source Code

```
#include  <memory.h>
#include  <stdio.h>
#include  <conio.h>
#include  "MilneSimpsonFunc.h"
#include  "RungeKuttaFunc.h"

double  funcTY(double nT,double nY); /* The function y'   */
int main(int argc, char* argv[])
{
    int nStatus = 0;
    int  i;
    double*  pTPoints = NULL;
    double*  pYPoints = NULL;
    int      nInterval;
    double   nLowerLimit,nUpperLimit,nRungeKuttaULimit,nInitYValue[4],nInitTValue[4];
    double   nWeightH;
    printf("\n ***  Milne-Simpson Method *** \n");
    printf("\n y'   =  (t - y)/2        \n\n");
    /*  This is the input stage */
    printf(" Enter the lower limit: ");
    scanf("%lf",&nLowerLimit);  fflush(stdin);
    printf(" Enter the upper limit: ");
    scanf("%lf",&nUpperLimit);  fflush(stdin);
    /*  Ask for the interval*/
    do{
        printf(" Enter the interval M  [ > 3]: ");
        scanf("%d",&nInterval);  fflush(stdin);
    }while(nInterval <= 3);
    nWeightH = (nUpperLimit - nLowerLimit) / (double)nInterval;
    nRungeKuttaULimit = nWeightH * (double)3 + nLowerLimit;
    printf(" Enter the initial value y(0): ");
    scanf("%lf",&nInitYValue[0]);  fflush(stdin);


computeRungeKutta(nLowerLimit,nRungeKuttaULimit,nInitYValue[0],3,funcTY,nInitTValue,nInit
YValue);
    printf(" The computed initial values using RK4: ");
    printf("\n ------------------------- ");
    printf("\n   t(k)     \t     y(k)     ");

    for(i = 0; i <= 3; i++)
          printf("\n %2.8lf \t  %2.8lf",nInitTValue[i],nInitYValue[i]);
    printf("\n ------------------------- ");
    pTPoints = (double*) malloc((nInterval + 1) * sizeof(pTPoints[0]));
    pYPoints = (double*) malloc((nInterval + 1) * sizeof(pYPoints[0]));

    if(!pTPoints || !pYPoints)       nStatus = -1;

    if(nStatus == 0)
    {

computeMilneSimpson(nLowerLimit,nUpperLimit,nInitYValue,nInterval,funcTY,pTPoints,pYPoint
s);
        printf("\n Result:");
        printf("\n ------------------------- ");
        printf("\n   t(k)     \t     y(k)     ");

        for(i = 0; i <= nInterval; i++ )
              printf("\n %2.8lf \t  %2.8lf",pTPoints[i],pYPoints[i]);
         printf("\n ------------------------- ");
    }
```

```
    /* Report any errors */
    switch(nStatus)
    {
    case -1:
        printf("\nMemory allocation error!!");
        break;
    default:
        break;
    }
    /* Free the allocation */
    if(pTPoints)   free(pTPoints);
    if(pYPoints)   free(pYPoints);

    printf("\nPress any key to continue....");
    getch();
    return 0;
}

/* Evaluates the function y'  */
double  funcTY(double nT,double nY)
{   return ((nT - nY) / 2); }
```

```
#include  "MilneSimpsonFunc.h"
/* Function name       : computeMilneSimpson
* Description          : Computes the initial value problem y' = f(t,y). using Milne -
Simpson Method
* Parameter(s)        :
*     nLowerLimit          [in]  The lower limit.
*     nUpperLimit          [in]  The upper limit.
*     nInitYValue[]        [in]  The initial value Y(0) - Y(3).
*     nInterval            [in]  The interval M.
*     pFuncOfTY            [in]  The function y' = f(t,y).
*     *pTPoints            [out] Stores the values of t(k).
*     *pYPoints            [out] Stores the values of y(k).
* Return              :
*    int           - 0  Success.
*/
int _cdecl
computeMilneSimpson(double nLowerLimit,double nUpperLimit,double nInitYValue[],int
nInterval,double (*pFuncOfTY)(double,double),double *pTPoints,double *pYPoints)
{
    int nRet = 0;
    int     i;
    double  nWeightH;
    double  nPredictedOld;
    double  nPredictedNew;
    double  nPredictedMod;
    double  nYOld;
    double  nF1;  /* Result of f(ti+1,yi+1)  */
    double  nF2;  /* Result of f(ti+2,yi+2)  */
    double  nF3;  /* Result of f(ti+3,yi+3)  */
    double  nF4;  /* Result of f(ti+4,P)     */
    /* Clear the pTPoints && pYPoints. */
    memset(pTPoints,0,(nInterval + 1) * sizeof(pTPoints[0]));
    memset(pYPoints,0,(nInterval + 1) * sizeof(pTPoints[0]));
    nWeightH = (nUpperLimit - nLowerLimit) / (double)nInterval;
    for(i = 0; i <= 3; i++)
    {
        pTPoints[i] = nLowerLimit + nWeightH * i;
        pYPoints[i] = nInitYValue[i];
    }
    /* Get the respective results */
    nF1 = pFuncOfTY(pTPoints[1],pYPoints[1]);
    nF2 = pFuncOfTY(pTPoints[2],pYPoints[2]);
    nF3 = pFuncOfTY(pTPoints[3],pYPoints[3]);
    nPredictedOld = 0;
    nYOld  = 0;
    /* Start the iteration using the Euler's Method */
    for(i = 3; i < nInterval; i++)
    {
        /* Compute the predicted value */
        nPredictedNew = pYPoints[i-3] + 4 * nWeightH * (2 * nF1 - nF2 + 2 * nF3)/3;
        nPredictedMod = nPredictedNew + 28 * (nYOld - nPredictedOld)/29;
        pTPoints[i+1] = nLowerLimit + nWeightH * (i + 1);
        nF4 = pFuncOfTY(pTPoints[i+1],nPredictedMod);
        /* Compute the Y(i+1) */
        pYPoints[i + 1] = pYPoints[i-1] + nWeightH * (nF2 + 4 * nF3 + nF4)/3;
        nPredictedOld = nPredictedNew;
        nYOld = pYPoints[i + 1];
        /* Roll the values down */
        nF1 = nF2;  nF2 = nF3;  nF3 = pFuncOfTY(pTPoints[i+1],pYPoints[i+1]);
    }

    return nRet;
}
```

Program Output

```
***  Milne-Simpson Method ***

y'    =  (t - y)/2

Enter the lower limit: 0
Enter the upper limit: 3
Enter the interval M  [ > 3]: 24
Enter the initial value y(0): 1
The computed initial values using RK4:
---------------------------
  t(k)              y(k)
0.00000000       1.00000000
0.12500000       0.94323921
0.25000000       0.89749075
0.37500000       0.86208742
---------------------------
Result:
---------------------------
  t(k)              y(k)
0.00000000       1.00000000
0.12500000       0.94323921
0.25000000       0.89749075
0.37500000       0.86208742
0.50000000       0.83640234
0.62500000       0.81984692
0.75000000       0.81186781
0.87500000       0.81194560
1.00000000       0.81959193
1.12500000       0.83434848
1.25000000       0.85578422
1.37500000       0.88349472
1.50000000       0.91709958
1.62500000       0.95624191
1.75000000       1.00058597
1.87500000       1.04981686
2.00000000       1.10363823
2.12500000       1.16177223
2.25000000       1.22395731
2.37500000       1.28994828
2.50000000       1.35951429
2.62500000       1.43243902
2.75000000       1.50851869
2.87500000       1.58756243
3.00000000       1.66939038
---------------------------
Press any key to continue....
```

Finite-Difference Solution for the Wave Equation

Discussion

A **partial differential equation** is a relation between the partial derivatives of an unknown function and the independent variables. The order of the highest derivative is called the **order** of the equation.

Since this is the first topic that involves the study of the finite-difference methods which are based on formulas for approximating the first and the second order derivative of a function, one must first classify the three types of equation which are covered by these topics (Algo 10.1 – 10.4).

A PDE of the form:

$$A\Phi_{xx} + B\Phi_{xy} + C\Phi_{yy} = F(x, y, \Phi, \Phi_x, \Phi_y) \tag{1}$$

where *A, B, C* are constants is called quasilinear and has three types:

Type	Coefficient	Typical Form	Name
Hyperbolic	$B^2 - 4AC > 0$	$u_{tt} - c^2 u_{xx} = 0$	Wave Equation
Parabolic	$B^2 - 4AC = 0$	$u_t - c^2 u_{xx} = 0$	Heat Equation
Elliptic	$B^2 - 4AC < 0$	$u_{xx} + u_{yy} = 0$	Laplace Equation

whose names arise by analogy with the curve

$$ax^2 + 2bxy + cy^2 = f, \tag{2}$$

which represents a hyperbola, a parabola, and an ellipse depending on whether $b^2 - 4ac$ is positive, zero or negative, respectively.

A one-dimensional model for a vibrating string will be used to illustrate the solution to hyperbolic equations using the **finite-difference solution for the wave equation** by considering the wave equation:

$$u_{tt}(x,t) = c^2 u_{xx}(x,t) \quad \text{for } 0 < x < a \text{ and } 0 < t < b, \tag{3}$$

with boundary conditions

$$u(0,t) = 0 \text{ and } u(a,t) = 0 \quad \text{for } 0 \le t \le b,$$
$$u(x,0) = f(x) \quad \text{for } 0 \le x \le a,$$
$$u_t(x,0) = g(x) \quad \text{for } 0 < x < a,$$

(4)

The wave equation serves as a model in solving for the displacement u of a vibrating elastic string with fixed ends at $x = 0$ and $x = a$. Since this is a difference equation, one starts the process by creating a grid partitioned into *n-1* by *m-1* rectangles with sides h and k corresponding to Δx and Δt, respectively. Boundary conditions given in (4) will then be assigned to the grid. Since two starting rows are needed before the third row can be generated, the first row will be assigned with the *f(x)* in (4) while the second will be assigned with:

$$u_{i,2} = (1-r^2) f_i + k g_i + \frac{r^2}{2}(f_{i+1} + f_{i-1}) \qquad \text{for } i = 2,3...n-1. \tag{5}$$

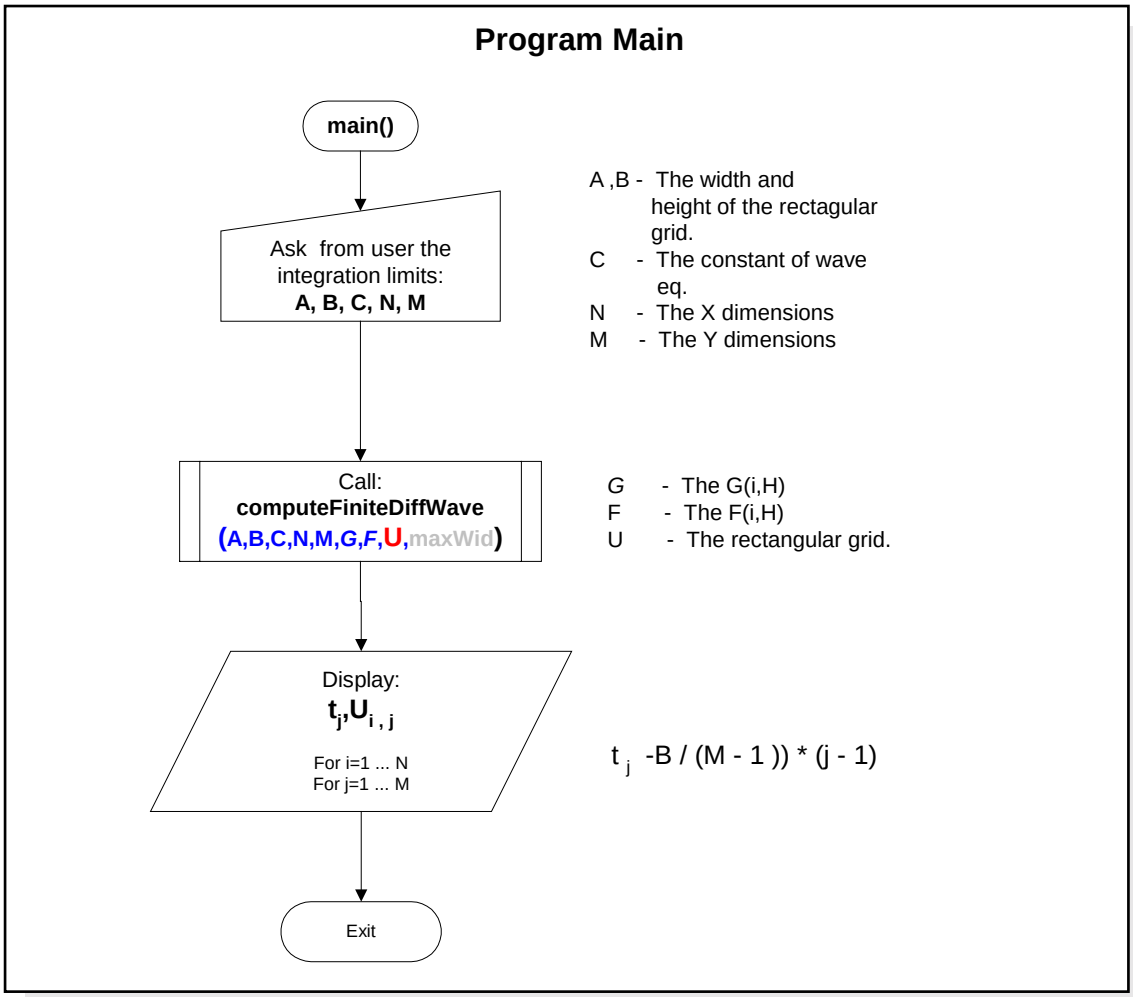
Finally, the new rows will be generated using:

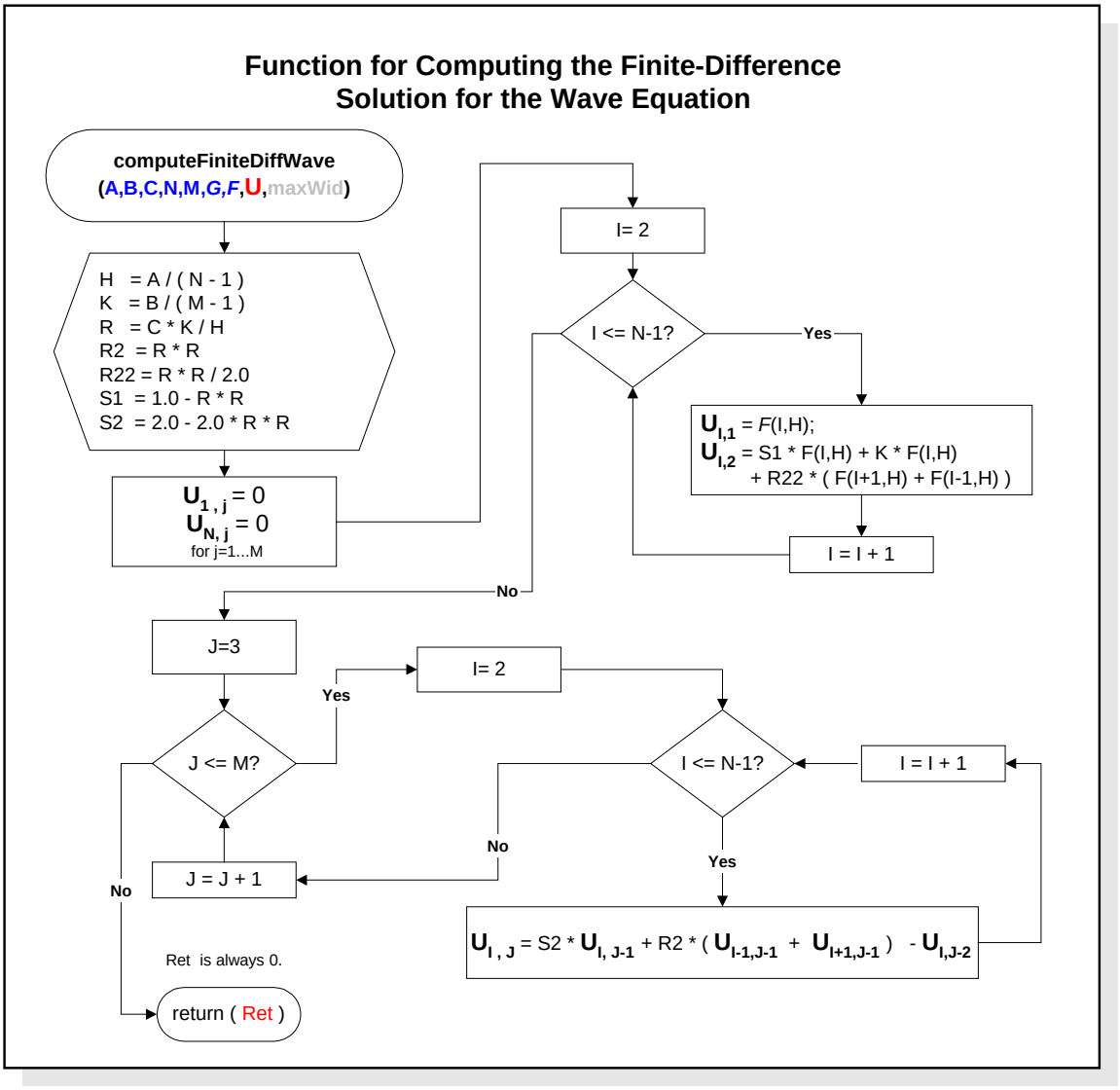
$$u_{i,j+1} = (2-2r^2)u_{i,j} + r^2(u_{i+1,j} + u_{i-1,j}) - u_{i,j-1} \qquad \text{for } i = 2,3...n-1 \tag{6}$$

where $r = ck/h$. More details can be found on pp. 501-506 of the textbook.

Algorithm 10.1 (Finite-Difference Solution for the Wave Equation). To approximate the solution $u_{tt}(x,t) - c^2 u_{xx}(x,t)$ over $\mathrm{R} = \{(\mathrm{x},\mathrm{t}) : 0 \le x \le, 0 \le t \le b\}$ with $u(0,t)=0$, $u(a,t)=0$ *for* $0 \le t \le b$ *and* $u(x,0)=f(x)$, $u_t(x,0)=g(x)$ *for* $0 \le x \le a$.

Program Flow




Function for Computing the Finite-Difference Solution for the Wave Equation
computeFiniteDiffWave
(A,B,C,N,M,G,F,U,maxWid)
H = A / (N - 1)
K = B / (M - 1)
R = C * K / H
R2 = R * R
R22 = R * R / 2.0
S1 = 1.0 - R * R
S2 = 2.0 - 2.0 * R * R
U1 , j = 0
UN, j = 0
for j=1...M
I= 2
I <= N-1?
Yes
UI,1 = F(I,H);
UI,2 = S1 * F(I,H) + K * F(I,H)
+ R22 * (F(I+1,H) + F(I-1,H))
I = I + 1
No
J=3
J <= M?
Yes
I= 2
I <= N-1?
I = I + 1
No
Yes
J = J + 1
No
UI , J = S2 * UI, J-1 + R2 * (UI-1,J-1 + UI+1,J-1) - UI,J-2
Ret is always 0.
return (Ret)

Program Source Code

```
#include <stdio.h>
#include <conio.h>
#include "FiniteDiffFuncW.h"
double  gFunc(int i,double H);
double  fFunc(int i,double H);
int main(int argc, char* argv[])
{
    double grid[50][50];
    double wid_A, ht_B;          /* in : Width and height of Rectangle  */
    double cons_C;               /* in : Wave equation const.           */
    int xdim_N, ydim_M;          /* dx and dy  */
    int i,j;
    printf("\n  ***  Finite-Difference Solution for Wave Equation *** \n\n");
    /*  Ask input from user */
    printf("Please enter rectangular grid width  A: ");
    scanf("%lf",&wid_A);     fflush(stdin);
    printf("Please enter rectangular grid height B: ");
    scanf("%lf",&ht_B);      fflush(stdin);
    printf("Please enter wave eq. constant C: ");
    scanf("%lf",&cons_C);  fflush(stdin);
    printf("Please enter grid x dimension N: ");
    scanf("%d",&xdim_N);  fflush(stdin);
    printf("Please enter grid y dimension M: ");
    scanf("%d",&ydim_M);  fflush(stdin);

    /* Compute the finite difference */
    computeFiniteDiffWave(wid_A,ht_B,cons_C,xdim_N,ydim_M,gFunc,fFunc,&grid[0][0],50);

    /* Display the output */
    i = 1;
    printf("\n=================================================================\n");
    do{
        int nRow;
        printf(" t[j] \t");
        for (nRow = 0; (nRow + i) <= xdim_N && nRow < 4; nRow++ )
                printf("   x[%d]  \t",(nRow + i));
        printf("\n-----------------------------------------------------------------\n");

        for ( j = 1; j <= ydim_M; j++ )
        {
            printf("%2.4lf\t", (ht_B / ( ydim_M - 1 )) * (j - 1));

            for (nRow = 0; (nRow + i) <= xdim_N && nRow < 4; nRow++ )
                printf("%2.6lf\t", GRID(&grid[0][0],(nRow + i),j,50));

            printf("\n");
        }

        i+=4;
        printf("\n=================================================================\n");

    }while(i <= xdim_N);
    printf("\nPress any key to continue..");
    getch();
    return 0;
}

double  gFunc(int i,double H)
{   /* g(x) = 0 */   return 0; }
double  fFunc(int i,double H)
{    double arg;    arg = H * (i - 1);
     return ( sin ( 3.1415926 * arg) + sin ( 2 * 3.1415926 * arg));
}
```

```
#include "FiniteDiffFuncW.h"

/* Function name      : computeFiniteDiffWave
* Description        : Computes the Finite-Difference of the Wave Equation.
* Parameter(s)       :
*     wid_A          [in]  The width  of the rectangular grid.
*     ht_B           [in]  The height of the rectangular grid.
*     cons_C         [in]  The constant  of wave eq..
*     xdim_N         [in]  The dimension in X.
*     ydim_M         [in]  The dimension in Y.
*     pGFunc      [in]  g(x)
*     pFFunc      [in]  f(x)
*     gridU         [out] The grid.
*     maxWid         [in]  The max width of the grid.
* Return             :
*    0   - Success
*/
int
computeFiniteDiffWave(double wid_A,double ht_B,double cons_C,int xdim_N,int
ydim_M,double(*pGFunc)(int,double),double(*pFFunc)(int,double),double* gridU,int maxWid)
{
    int nRet = 0;
    double    dx_H;       /* The delta X */
    double    dy_K;       /* The delta Y */
    double    ratio_R;    /* The  ratio R */
    double    ratio_R2;   /* R^2   */
    double    ratio_R22;  /* R^2/2 */
    double    term_S1;    /* 1 - R^2 */
    double    term_S2;    /* 2 - 2R^2 */
    int i,j;
    /* Compute step sizes */
    dx_H      = wid_A / (xdim_N - 1);
    dy_K      = ht_B / ( ydim_M - 1 );
    ratio_R   = cons_C * dy_K / dx_H;
    ratio_R2  = ratio_R * ratio_R;
    ratio_R22 = ratio_R * ratio_R / 2.0;
    term_S1   = 1.0 - ratio_R * ratio_R;
    term_S2   = 2.0 - 2.0 * ratio_R * ratio_R;
    /* Apply boundary conditions */
    for ( j = 1; j <= ydim_M; j++ )
    {
        GRID(gridU,1,j,maxWid)      = 0;
        GRID(gridU,xdim_N,j,maxWid) = 0;
    }
    /* Compute the starting values: first and second rows */
    for ( i = 2; i <= xdim_N - 1 ; i++ )
    {
        GRID(gridU,i,1,maxWid) = pFFunc(i,dx_H);
        GRID(gridU,i,2,maxWid) = term_S1 * pFFunc(i,dx_H) + dy_K * pGFunc(i,dx_H) +
ratio_R22 * ( pFFunc(i+1,dx_H) + pFFunc(i-1,dx_H));
    }
    /* Generate new waves */
    for ( j = 3; j <= ydim_M; j++ )
    {
        for ( i = 2; i <= xdim_N - 1; i++ )
        {
            GRID(gridU,i,j,maxWid) = term_S2 * GRID(gridU,i,j-1,maxWid) + ratio_R2
                * ( GRID(gridU,i-1,j-1,maxWid) + GRID(gridU,i+1,j-1,maxWid)) -
GRID(gridU,i,j-2,maxWid);
        }
    }
    return nRet;
}
```

Program Output

```
  ***  Finite-Difference Solution for Wave Equation ***

Please enter rectangular grid width  A: 1
Please enter rectangular grid height B: .5
Please enter wave eq. constant C: 2
Please enter grid x dimension N: 11
Please enter grid y dimension M: 11

================================================================
 t[j]      x[1]            x[2]            x[3]            x[4]
----------------------------------------------------------------
0.0000  0.000000        0.896802        1.538842        1.760074
0.0500  0.000000        0.769421        1.328438        1.538842
0.1000  0.000000        0.431636        0.769421        0.948401
0.1500  0.000000        0.000000        0.051599        0.181636
0.2000  0.000000        -0.380037       -0.587785       -0.519421
0.2500  0.000000        -0.587785       -0.951056       -0.951057
0.3000  0.000000        -0.571020       -0.951057       -1.019421
0.3500  0.000000        -0.363271       -0.639384       -0.769421
0.4000  0.000000        -0.068364       -0.181636       -0.360616
0.4500  0.000000        0.181636        0.210404        -0.000000
0.5000  0.000000        0.278768        0.363271        0.142039

================================================================
 t[j]      x[5]            x[6]            x[7]            x[8]
----------------------------------------------------------------
0.0000  1.538842        1.000000        0.363271        -0.142039
0.0500  1.380037        0.951057        0.428980        0.000000
0.1000  0.951057        0.809017        0.587785        0.360616
0.1500  0.377381        0.587785        0.740653        0.769421
0.2000  -0.181636       0.309017        0.769421        1.019421
0.2500  -0.587785       -0.000000       0.587785        0.951057
0.3000  -0.769421       -0.309017       0.181636        0.519421
0.3500  -0.740653       -0.587785       -0.377381       -0.181636
0.4000  -0.587785       -0.809017       -0.951057       -0.948401
0.4500  -0.428980       -0.951057       -1.380037       -1.538842
0.5000  -0.363271       -1.000000       -1.538842       -1.760074

================================================================
 t[j]      x[9]            x[10]           x[11]
----------------------------------------------------------------
0.0000  -0.363271       -0.278768       0.000000
0.0500  -0.210404       -0.181636       0.000000
0.1000  0.181636        0.068364        0.000000
0.1500  0.639384        0.363271        0.000000
0.2000  0.951057        0.571020        0.000000
0.2500  0.951056        0.587785        0.000000
0.3000  0.587785        0.380037        0.000000
0.3500  -0.051599       -0.000000       0.000000
0.4000  -0.769421       -0.431636       0.000000
0.4500  -1.328438       -0.769421       0.000000
0.5000  -1.538842       -0.896802       0.000000

================================================================

Press any key to continue..
```

Forward-Difference Method for the Heat Equation

Discussion

For parabolic differential equations that will be solved using the **forward-difference method for the heat equation**, one may consider using a one-dimensional heat equation:

$$u_t(x,t) = c^2 u_{xx}(x,t) \quad for\ 0 < x < a\ and\ 0 < t < b, \tag{1}$$

with initial condition

$$u\ (x,0) = f(x) \quad for\ t = 0\ and\ 0 \le x \le a \tag{2}$$

and the boundary conditions

$$u(0,t) = g_1(t) \equiv c_1 \quad for\ x = 0\ and \quad 0 \le t \le b\ ,$$

$$u(a,t) = g_2(t) \equiv c_2 \quad for\ x = a\ and \quad 0 \le t \le b. \tag{3}$$

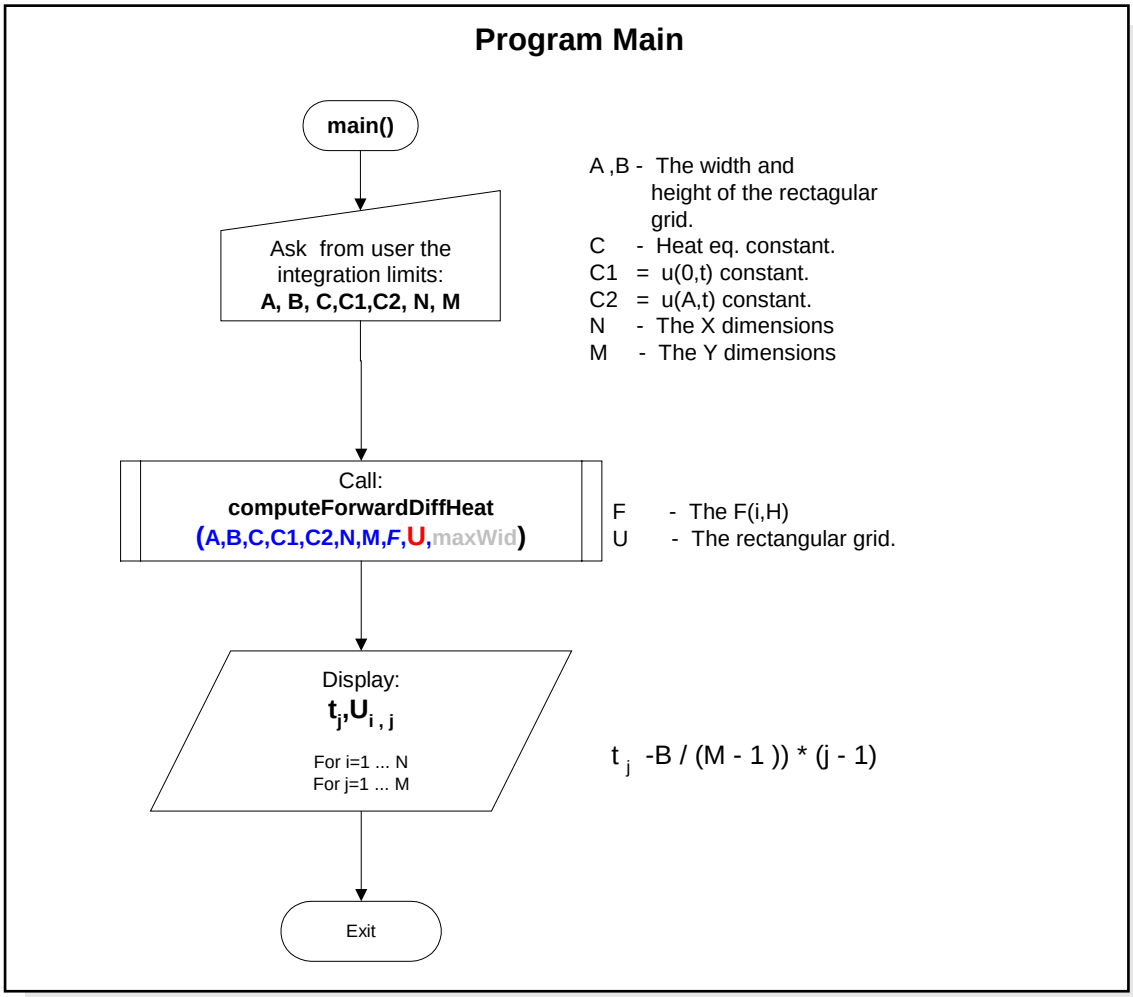
The heat equation serves as a model in solving for the temperature in an insulated rod with ends held at constant temperature c_1 and c_2 and the initial temperature distributed along the rod being $f(x)$. Since this is a difference equation, one starts the process by creating a grid partitioned into $n-1$ by $m-1$ rectangles with sides h and k corresponding to Δx and Δt respectively. Boundary conditions given in (4) will then be assigned to the grid. After which, the initial condition $f(x)$ in (2) will be assigned to the first row. Finally, the new waves will be generated using:

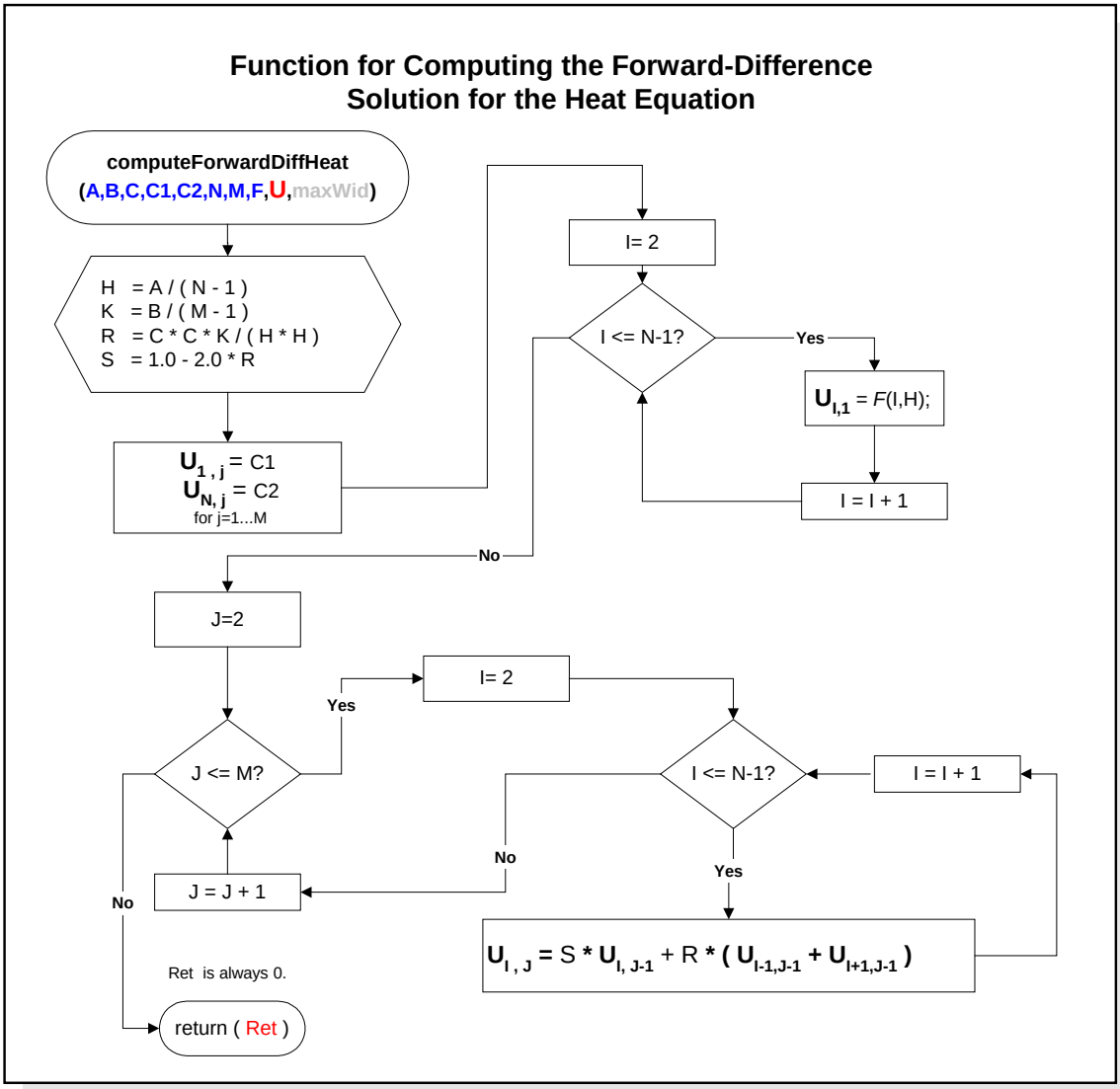
$$u_{i,j+1} = (1-2r)u_{i,j} + r(u_{i-1,j} + u_{i+1,j}) \quad for\ i = 2,3...n-1 \tag{4}$$

where $r = c^2k/h^2$. More details can be found on pp. 509-513 of the textbook.

Algorithm 10.2 (Forward-Difference Method for the Heat Equation). To approximate the solution of $u_t(x,t) - c^2 u_{xx}(x,t)$ over $\mathrm{R} = \{(\mathrm{x},\mathrm{t}) : 0 \le x \le, 0 \le t \le b\}$ with $u(x,0) = f(x)$ *for* $0 \le x \le a$ *and* $u(0,t) = c_1$, $u(a,t) = c_2$ *for* $0 \le t \le b$.

Program Flow




Function for Computing the Forward-Difference Solution for the Heat Equation
computeForwardDiffHeat (A,B,C,C1,C2,N,M,F,U,maxWid)
H = A / (N - 1)
K = B / (M - 1)
R = C * C * K / (H * H)
S = 1.0 - 2.0 * R
U1 , j = C1
UN, j = C2
for j=1...M
I= 2
I <= N-1?
Yes
UI,1 = F(I,H);
I = I + 1
No
J=2
I= 2
Yes
J <= M?
I <= N-1?
I = I + 1
No
Yes
No
J = J + 1
UI , J = S * UI, J-1 + R * (UI-1,J-1 + UI+1,J-1)
Ret is always 0.
return (Ret)

Program Source Code

```
#include <stdio.h>
#include <conio.h>
#include "ForwardiffFuncH.h"
double  fFunc(int i,double H);
int main(int argc, char* argv[])
{
    double grid[50][50];
     double wid_A, ht_B;             /* in : Width and height of Rectangle  */
    double cons_C,cons_C1,cons_C2;  /* in : Heat equation const.           */
    int xdim_N, ydim_M;             /* dx and dy  */
    int i,j;
    printf("\n  ***  Forward-Difference Solution for the Heat Equation *** \n\n");
    /*  Ask input from user */
    printf("Please enter rectangular grid width  A   : ");
    scanf("%lf",&wid_A);     fflush(stdin);
    printf("Please enter rectangular grid height B   : ");
    scanf("%lf",&ht_B);      fflush(stdin);
    printf("Please enter constant C of heat eq.: ");
       scanf("%lf",&cons_C);  fflush(stdin);
    printf("Please enter constant C1 = u(0,t): ");
       scanf("%lf",&cons_C1);  fflush(stdin);
    printf("Please enter constant C2 = u(A,t): ");
       scanf("%lf",&cons_C2);  fflush(stdin);
    printf("Please enter grid x dimension N: ");
    scanf("%d",&xdim_N);  fflush(stdin);
    printf("Please enter grid y dimension M: ");
    scanf("%d",&ydim_M);  fflush(stdin);
    /* Compute the forward difference for heat equation */
computeForwardDiffHeat(wid_A,ht_B,cons_C,cons_C1,cons_C2,xdim_N,ydim_M,fFunc,&grid[0][0],
50);
    /* Display the output */
    i = 1;
    printf("\n=================================================================\n");
    do{
        int nRow;
        printf(" t[j] \t");
        for (nRow = 0; (nRow + i) <= xdim_N && nRow < 4; nRow++ )
                printf("   x[%d]  \t",(nRow + i));
        printf("\n-----------------------------------------------------------------\n");

        for ( j = 1; j <= ydim_M; j++ )
        {
            printf("%2.4lf\t", (ht_B / ( ydim_M - 1 )) * (j - 1));

            for (nRow = 0; (nRow + i) <= xdim_N && nRow < 4; nRow++ )
                printf("%2.6lf\t", GRID(&grid[0][0],(nRow + i),j,50));

            printf("\n");
        }
        i+=4;
        printf("\n=================================================================\n");
    }while(i <= xdim_N);
    printf("\nPress any key to continue..");
    getch();
    return 0;
}
double  fFunc(int i,double H)
{
    double arg;
    arg = H * (i - 1);
    return ( 4.0 * arg - 4.0 * arg * arg);
}
```

```
#include "ForwardiffFuncH.h"

/* Function name      : computeForwardDiffHeat
* Description        : Computes the Forward-Difference for the Heat Equation.
* Parameter(s)       :
*     wid_A         [in]  The width  of the rectangular grid.
*     ht_B          [in]  The height of the rectangular grid.
*     cons_C        [in]  constant of heat eq..
*     cons_C1       [in]  constant C1 = u(0,t).
*     cons_C2       [in]  constant C2 = u(A,t).
*     xdim_N        [in]  The dimension in X.
*     ydim_M        [in]  The dimension in Y.
*     pFFunc        [in]  f(x)
*     gridU         [out] The grid.
*     maxWid         [in]  The max width of the grid.
* Return             :
*    0   - Success
*/
int
computeForwardDiffHeat(double wid_A,double ht_B,double cons_C,double cons_C1,double
cons_C2,int xdim_N,int ydim_M,double(*pFFunc)(int,double),double* gridU,int maxWid)
{
    int nRet = 0;

    double    dx_H;       /* The delta X */
    double    dy_K;       /* The delta Y */
    double    ratio_R;    /* The  ratio R */
    double    term_S;     /* 1 - 2*R */

    int i,j;

    /* Compute step sizes */
    dx_H   = wid_A / ( xdim_N - 1 );
    dy_K   = ht_B / ( ydim_M - 1 );
    ratio_R   = cons_C * cons_C * dy_K / ( dx_H * dx_H );
    term_S   = 1.0 - 2.0 * ratio_R;

    /* Assign boundary conditions */
    for ( j = 1; j <= ydim_M; j++ )
    {
        GRID(gridU,1,j,maxWid) = cons_C1;
        GRID(gridU,xdim_N,j,maxWid) = cons_C2;
    }

    /* Fill the first row */
    for ( i = 2; i <= xdim_N - 1 ; i++ ) GRID(gridU,i,1,maxWid) = pFFunc(i,dx_H);

    /* Generate new waves */
    for ( j = 2; j <= ydim_M; j++ )
    {
        for ( i = 2; i <= xdim_N - 1; i++ )
        {
            GRID(gridU,i,j,maxWid) = term_S * GRID(gridU,i,j-1,maxWid) + ratio_R * (
GRID(gridU,i-1,j-1,maxWid) + GRID(gridU,i+1,j-1,maxWid) );
        }
    }

    return nRet;
}
```

Program Output

```
  ***  Forward-Difference Solution for the Heat Equation ***

Please enter rectangular grid width  A    : 1
Please enter rectangular grid height B    : .2
Please enter constant C of heat eq.: 1
Please enter constant C1 = u(0,t): 0
Please enter constant C2 = u(A,t): 0
Please enter grid x dimension N: 6
Please enter grid y dimension M: 11

================================================================
 t[j]      x[1]          x[2]          x[3]          x[4]
----------------------------------------------------------------
0.0000  0.000000      0.640000      0.960000      0.960000
0.0200  0.000000      0.480000      0.800000      0.800000
0.0400  0.000000      0.400000      0.640000      0.640000
0.0600  0.000000      0.320000      0.520000      0.520000
0.0800  0.000000      0.260000      0.420000      0.420000
0.1000  0.000000      0.210000      0.340000      0.340000
0.1200  0.000000      0.170000      0.275000      0.275000
0.1400  0.000000      0.137500      0.222500      0.222500
0.1600  0.000000      0.111250      0.180000      0.180000
0.1800  0.000000      0.090000      0.145625      0.145625
0.2000  0.000000      0.072813      0.117813      0.117813

================================================================
 t[j]      x[5]          x[6]
----------------------------------------------------------------
0.0000  0.640000      0.000000
0.0200  0.480000      0.000000
0.0400  0.400000      0.000000
0.0600  0.320000      0.000000
0.0800  0.260000      0.000000
0.1000  0.210000      0.000000
0.1200  0.170000      0.000000
0.1400  0.137500      0.000000
0.1600  0.111250      0.000000
0.1800  0.090000      0.000000
0.2000  0.072813      0.000000

================================================================

Press any key to continue..
```

Crank-Nicholson Method for the Heat Equation

Discussion

For a one-dimensional heat equation:

$$u_t(x,t) = c^2 u_{xx}(x,t) \qquad for\ 0 < x < a\ and\ 0 < t < b, \tag{1}$$

with initial condition

$$\begin{aligned} u\ (x,0) &= f(x) \qquad for\ t = 0\ and\ 0 \le x \le a, \\ u\ (x,0) &= f(x) \qquad for\ t = 0\ and\ 0 \le x \le a, \end{aligned} \tag{2}$$

and the boundary conditions

$$\begin{aligned} u(0,t) &= g_1(t) \equiv c_1 \quad for\ x = 0\ and\ \ 0 \le t \le b\ , \\ u(a,t) &= g_2(t) \equiv c_2 \quad for\ x = a\ and\ \ 0 \le t \le b, \end{aligned} \tag{3}$$

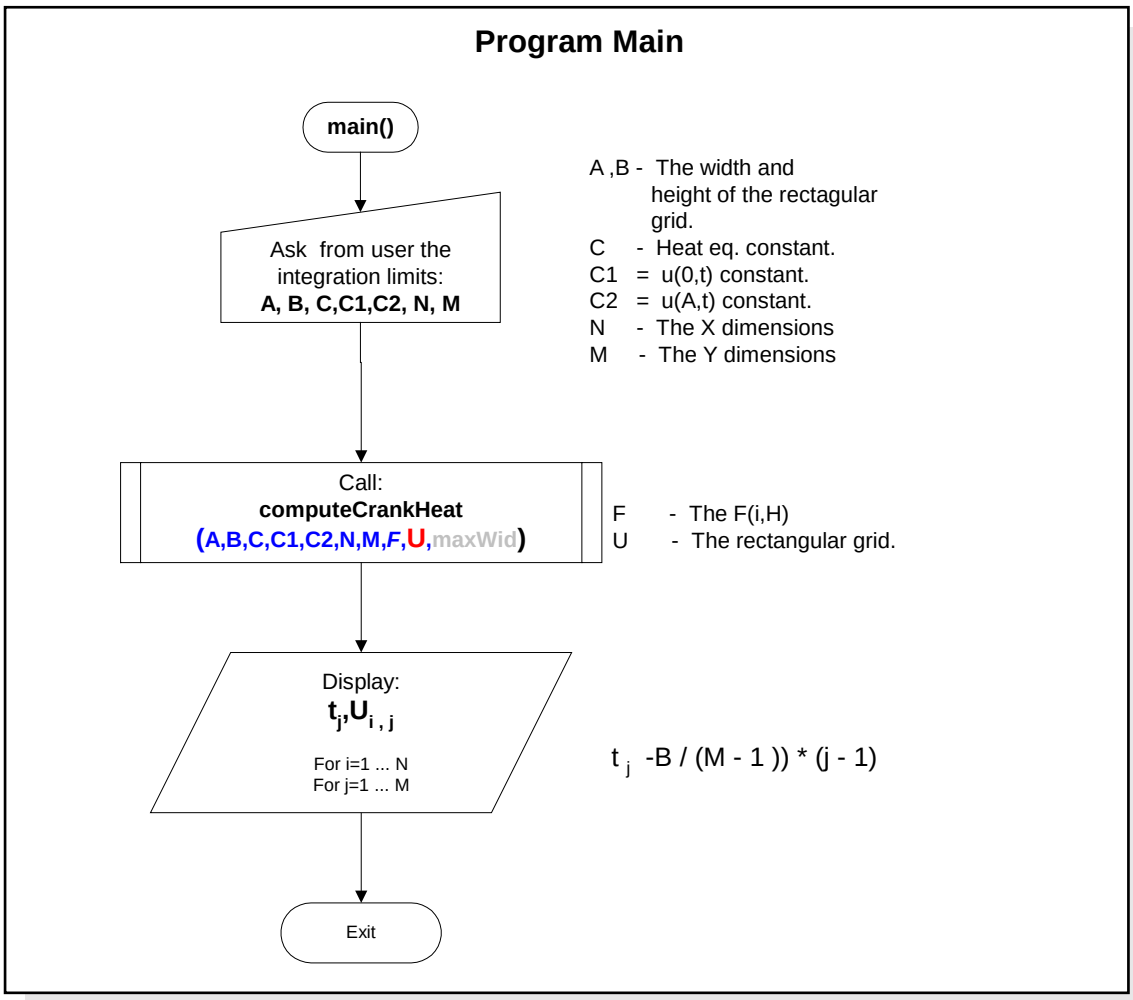
another method invented by John Crank and Phyllis Nicholson, can be used in approximating the partial derivative in (1). Using a finite difference centered at the time value $t_{j+1/2}$, the derived formula:

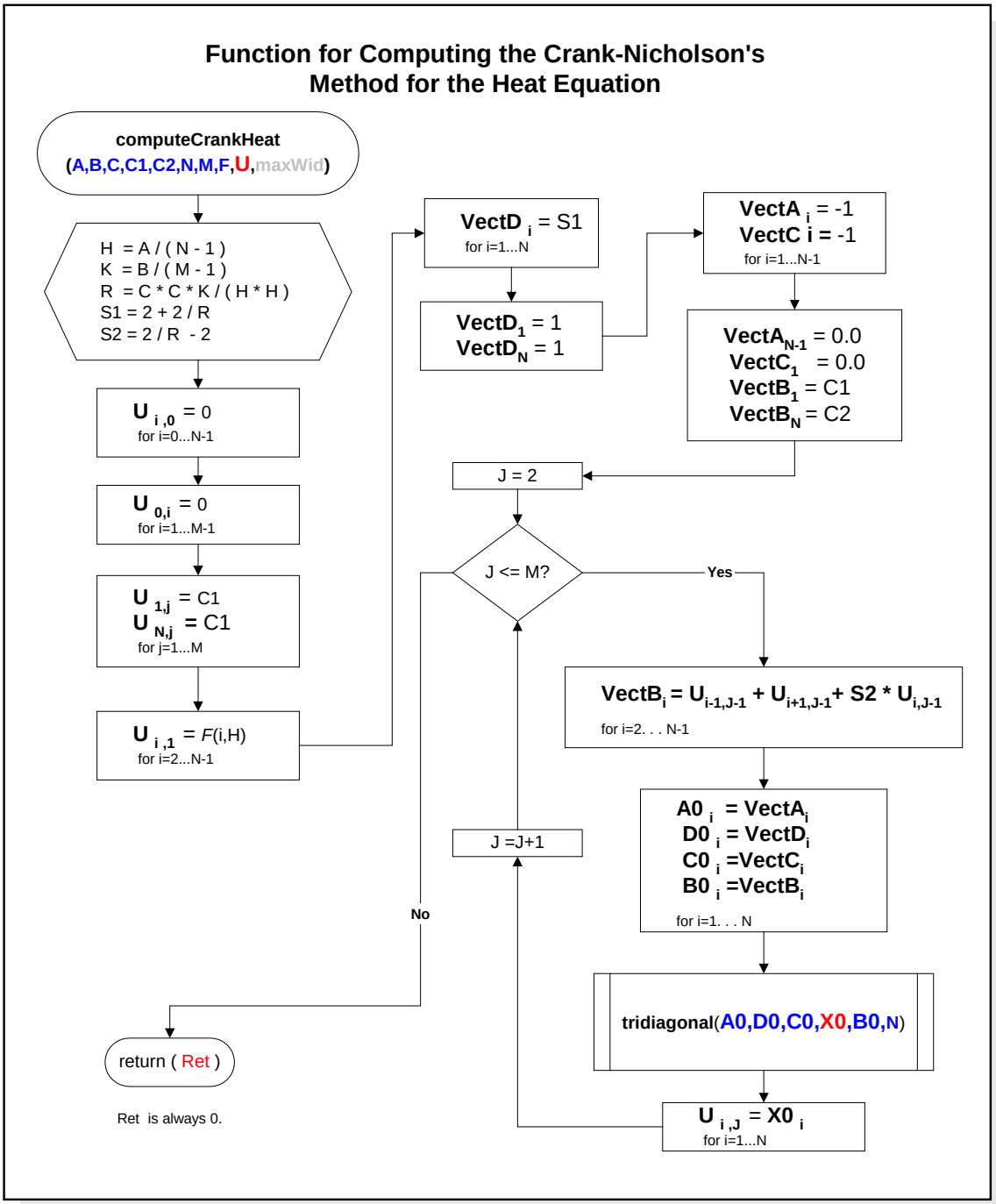
$$-ru_{i-1,j+1} + (2+2r)u_{i,j+1} - ru_{i+1,j+1} = (2-2r)u_{i,j} + r(u_{i-1,j} + u_{i+1,j}) \tag{4}$$

(Details for obtaining (4) can be found on pp. 513-514 of the textbook.) is used in generating new rows using a tridiagonal linear system A**X**=B.

Algorithm 10.3 (Crank-Nicholson Method for the Heat Equation). To approximate the solution of $u_t(x,t) - c^2 u_{xx}(x,t)$ over $R = \{(x,t):\ 0 \le x \le,\ 0 \le t \le b\}$ with $u(x,0) = f(x)$ *for* $0 \le x \le a$ *and* $u(0,t) = c_1$, $u(a,t) = c_2$ *for* $0 \le t \le b$.

Program Flow




Function for Computing the Crank-Nicholson's Method for the Heat Equation
computeCrankHeat
(A,B,C,C1,C2,N,M,F,U,maxWid)
H = A / (N - 1)
K = B / (M - 1)
R = C * C * K / (H * H)
S1 = 2 + 2 / R
S2 = 2 / R - 2
U i ,0 = 0
for i=0...N-1
U 0,i = 0
for i=1...M-1
U 1,j = C1
U N,j = C1
for j=1...M
U i ,1 = F(i,H)
for i=2...N-1
VectD i = S1
for i=1...N
VectD1 = 1
VectDN = 1
VectA i = -1
VectC i = -1
for i=1...N-1
VectAN-1 = 0.0
VectC1 = 0.0
VectB1 = C1
VectBN = C2
J = 2
J <= M?
Yes
No
VectBi = Ui-1,J-1 + Ui+1,J-1 + S2 * Ui,J-1
for i=2. . . N-1
A0 i = VectAi
D0 i = VectDi
C0 i = VectCi
B0 i = VectBi
for i=1. . . N
J =J+1
tridiagonal(A0,D0,C0,X0,B0,N)
U i ,J = X0 i
for i=1...N
return (Ret)
Ret is always 0.

Program Source Code

```
#include <stdio.h>
#include <conio.h>
#include "CrankHeatFunc.h"
double  fFunc(int i,double H);
int main(int argc, char* argv[])
{
    double grid[50][50];
    double wid_A, ht_B;             /* in : Width and height of Rectangle  */
    double cons_C,cons_C1,cons_C2;  /* in : Heat equation const.           */
    int xdim_N, ydim_M;             /* dx and dy  */
    int i,j;
    printf("\n  ***  Crank-Nicholson Method for the Heat Equation *** \n\n");
    /*  Ask input from user */
    printf("Please enter rectangular grid width  A   : ");
    scanf("%lf",&wid_A);     fflush(stdin);
    printf("Please enter rectangular grid height B   : ");
    scanf("%lf",&ht_B);      fflush(stdin);
    printf("Please enter constant C of heat eq.: ");
    scanf("%lf",&cons_C);  fflush(stdin);
    printf("Please enter constant C1 = u(0,t): ");
    scanf("%lf",&cons_C1);  fflush(stdin);
    printf("Please enter constant C2 = u(A,t): ");
    scanf("%lf",&cons_C2);  fflush(stdin);
    printf("Please enter grid x dimension N: ");
    scanf("%d",&xdim_N);  fflush(stdin);
    printf("Please enter grid y dimension M: ");
    scanf("%d",&ydim_M);  fflush(stdin);
    /* Compute the crank-nicholsons method for heat equation */
computeCrankHeat(wid_A,ht_B,cons_C,cons_C1,cons_C2,xdim_N,ydim_M,fFunc,&grid[0][0],50);

    /* Display the output */
    i = 1;
    printf("\n=================================================================\n");
    do{
        int nRow;
        printf(" t[j] \t");
        for (nRow = 0; (nRow + i) <= xdim_N && nRow < 4; nRow++ )
                printf("   x[%d]  \t",(nRow + i));
        printf("\n-----------------------------------------------------------------\n");

        for ( j = 1; j <= ydim_M; j++ )
        {
            printf("%2.4lf\t", (ht_B / ( ydim_M - 1 )) * (j - 1));

            for (nRow = 0; (nRow + i) <= xdim_N && nRow < 4; nRow++ )
                printf("%2.6lf\t", GRID(&grid[0][0],(nRow + i),j,50));
            printf("\n");
        }
        i+=4;
        printf("\n=================================================================\n");

    }while(i <= xdim_N);
    printf("\nPress any key to continue..");
    getch();
    return 0;
}

double  fFunc(int i,double H)
{
    double arg;
    arg = H * (i - 1);
    return (  sin( 3.1415926535 * arg ) + sin( 3.0 * 3.1415926535 * arg ) );
}
```

```
#include "CrankHeatFunc.h"
#define  MAX_VECT_SIZE  51

/* Function name      : computeCrankHeat
* Description        : Computes the Crank-Nicholson's method for the Heat Equation.
* Parameter(s)       :
*     wid_A         [in]  The width  of the rectangular grid.
*     ht_B          [in]  The height of the rectangular grid.
*     cons_C        [in]  constant of heat eq..
*     cons_C1       [in]  constant cons_C1 = u(0,t).
*     cons_C2       [in]  constant cons_C2 = u(wid_A,t).
*     xdim_N        [in]  The dimension in X.
*     ydim_M        [in]  The dimension in Y.
*     pFFunc        [in]  f(x)
*     gridU         [out] The grid.
*     maxWid         [in]  The max width of the grid.
* Return             :
*    0   - Success
*/
int
computeCrankHeat(double wid_A,double ht_B,double cons_C,double cons_C1,double cons_C2,int
xdim_N,int ydim_M,double(*pFFunc)(int,double),double* gridU,int maxWid)
{
    int nRet = 0;

    double    dx_H;       /* The delta X */
    double    dy_K;       /* The delta Y */
    double    ratio_R;    /* The  ratio ratio_R */
    double    term_S1;     /* 2 + 2/ratio_R */
       double    term_S2;     /* 2/ratio_R - 2*/

       double VectA[MAX_VECT_SIZE], VectB[MAX_VECT_SIZE]; /* diagonal and off-diagonal
*/
    double VectC[MAX_VECT_SIZE], VectD[MAX_VECT_SIZE]; /* elements of matrix A (see book)
*/
    double A0[MAX_VECT_SIZE], B0[MAX_VECT_SIZE]; /* backup of the VectA, VectB, VectC
*/
    double C0[MAX_VECT_SIZE], D0[MAX_VECT_SIZE]; /* and VectD
*/
    double X0[MAX_VECT_SIZE];          /* Solution vector for tridiagonal syst. */

    int i,j;

    /* Compute step sizes */
    dx_H  = wid_A / ( xdim_N - 1 );
       dy_K  = ht_B / ( ydim_M - 1 );

    /* Compute ratios and constants */
    ratio_R  = cons_C * cons_C * dy_K / ( dx_H * dx_H );
    term_S1 = 2.0 + 2.0 / ratio_R;
    term_S2 = 2.0 / ratio_R  - 2.0;

       for ( i = 0; i < xdim_N; i++ ) GRID(gridU,i,0,maxWid) = 0.0;
    for ( i = 1; i < ydim_M; i++ ) GRID(gridU,0,i,maxWid) = 0.0;

    /* Boundary conditions */
    for ( j = 1; j <= ydim_M; j++ )
    {
          GRID(gridU,1,j,maxWid) = cons_C1;
          GRID(gridU,xdim_N,j,maxWid) = cons_C2;
    }
```

```
    /* First row  */
    for ( i = 2; i <= xdim_N - 1 ; i++ ) GRID(gridU,i,1,maxWid) = pFFunc(i,dx_H);

    /* Form the diagonal elements of the matrix wid_A */
    for ( i = 1; i <= xdim_N ; i++ ) VectD[i] = term_S1;
    VectD[1] = 1.0;
    VectD[xdim_N] = 1.0;

    /* Form the off diagonal Elements of the matrix wid_A */

    for ( i = 1; i <= xdim_N - 1 ; i++ )
    {
        VectA[i] = -1.0;
        VectC[i] = -1.0;
    }

    VectA[xdim_N-1] = 0.0;
    VectC[1]   = 0.0;

    VectB[1] = cons_C1;
    VectB[xdim_N] = cons_C2;

    /* Construct successive rows in the grid */

    for ( j = 2; j <= ydim_M; j++ )
    {
        for ( i = 2; i <= xdim_N - 1; i++ )
        {
            VectB[i] = GRID(gridU,i-1,j-1,maxWid) + GRID(gridU,i+1,j-1,maxWid) + term_S2 *
GRID(gridU,i,j-1,maxWid);
        }

        /* Make a copy of the vetor since it will be
                modified by triSytem
            */
        for ( i = 1; i <= xdim_N; i++ )
        {
            A0[i] = VectA[i];
            B0[i] = VectB[i];
            C0[i] = VectC[i];
            D0[i] = VectD[i];
        }


            /* Solve the tri diagonal-system */
            tridiagonal(A0,D0,C0,X0,B0,xdim_N);

            /* Next row in grid */

        for ( i = 1; i <= xdim_N; i++ ) GRID(gridU,i,j,maxWid) = X0[i];

    } /* End of outer for-loop*/


    return nRet;
}
```

```
/* Function name       : triadiagonal
* Description         : Solves the triadiagonal system
* Parameter(s)        :
*       a[]            [in]   The sub diagonal vector.
*       d[]            [in]   The diagonal vector.
*       c[]            [in]   The super diagonal vector.
*       x[]            [out]  The unknown vector.
*       b[]            [in]   The constant vector.
*       n                 [in]   The number of elements.
* Return              :
*    int  0 - Success.
*        -1 - Failed.
*
*-------------------------------------------------------------------------------*/
int
tridiagonal(double a[],double d[],double c[],double x[],double b[],int n)
{
       int nRet = 0;

       int k;

       for(k = 2; k <= n; k++)
       {
              if(!d[k-1])
              {
                     /* Signal failure and stop */
                     nRet = -1;
                     break;
              }
              else
              {
                     double m;

                     m = a[k-1]/d[k-1];

                     d[k] = d[k] - m*c[k-1];
                     b[k] = b[k] - m*b[k-1];
              }
       }

       if(nRet == 0)
       {
              if(!d[n])
              {
                     /* Signal failure and stop */
                     nRet = -1;
              }
              else
              {
                     x[n] = b[n]/d[n];

                     for(k = n-1; k >= 1; k--)
                     {
                            x[k] = (b[k] - c[k] * x[k+1]) / d[k];
                     }

              }
       }

       return nRet;
}
```

Program Output

```
  ***  Crank-Nicholson Method for the Heat Equation ***

Please enter rectangular grid width  A    : 1
Please enter rectangular grid height B    : .01
Please enter constant C of heat eq.: 1
Please enter constant C1 = u(0,t): 0
Please enter constant C2 = u(A,t): 0
Please enter grid x dimension N: 11
Please enter grid y dimension M: 11

=================================================================
 t[j]      x[1]            x[2]            x[3]            x[4]
-----------------------------------------------------------------
0.0000  0.000000        1.118034        1.538842        1.118034
0.0010  0.000000        1.050967        1.457812        1.085686
0.0020  0.000000        0.989001        1.382801        1.055351
0.0030  0.000000        0.931734        1.313336        1.026877
0.0040  0.000000        0.878797        1.248981        1.000121
0.0050  0.000000        0.829848        1.189335        0.974952
0.0060  0.000000        0.784574        1.134030        0.951249
0.0070  0.000000        0.742686        1.082725        0.928902
0.0080  0.000000        0.703917        1.035107        0.907809
0.0090  0.000000        0.668023        0.990889        0.887875
0.0100  0.000000        0.634777        0.949804        0.869013

=================================================================
 t[j]      x[5]            x[6]            x[7]            x[8]
-----------------------------------------------------------------
0.0000  0.363271        0.000000        0.363271        1.118034
0.0010  0.400547        0.069438        0.400547        1.085686
0.0020  0.434229        0.132702        0.434229        1.055351
0.0030  0.464606        0.190286        0.464606        1.026877
0.0040  0.491947        0.242648        0.491947        1.000121
0.0050  0.516499        0.290207        0.516499        0.974952
0.0060  0.538488        0.333350        0.538488        0.951249
0.0070  0.558124        0.372433        0.558124        0.928902
0.0080  0.575598        0.407784        0.575598        0.907809
0.0090  0.591088        0.439703        0.591088        0.887875
0.0100  0.604757        0.468471        0.604757        0.869013

=================================================================
 t[j]      x[9]            x[10]           x[11]
-----------------------------------------------------------------
0.0000  1.538842        1.118034        0.000000
0.0010  1.457812        1.050967        0.000000
0.0020  1.382801        0.989001        0.000000
0.0030  1.313336        0.931734        0.000000
0.0040  1.248981        0.878797        0.000000
0.0050  1.189335        0.829848        0.000000
0.0060  1.134030        0.784574        0.000000
0.0070  1.082725        0.742686        0.000000
0.0080  1.035107        0.703917        0.000000
0.0090  0.990889        0.668023        0.000000
0.0100  0.949804        0.634777        0.000000

=================================================================

Press any key to continue..
```

Dirichlet Method for Laplace's Equation

Discussion

A Laplace's equation

$$\nabla^2 u = 0 \tag{1}$$

in cases where boundary values of *u* at all points of the sides of a rectangular region R which are subdivided into *n-1 x m-1* squares with side *h* as given, one maybe able to obtain the solution to PDE using the Laplace difference equation:

$$u_{i+1,j} + u_{i-1,j} + u_{i,j+1} + u_{i,j-1} - 4u_{i,j} = 0 \tag{2}$$

Using (2), one may be able to construct a system of linear of *n-1* equations and *n-1* unknowns, which are shown on page 571 of the textbook. But a problem of storage may arise when one wishes to obtain a better approximation of the solution because it requires a finer mesh that would lead to a greater computational storage as the number of partitions will be increased. In order to avoid this problem, one may opt to use the iterative method. Such process requires the same number of equations all throughout the process as long as a starting value (initial value) will be supplied and (2) must be rewritten in the iterative form:

$$u_{i,j} = u_{i,j} + r_{i,j} \tag{3}$$

where:

$$r_{i,j} = \frac{u_{i+1,j} + u_{i-1,j} + u_{i,j+1} + u_{i,j-1} - 4u_{i,j}}{4} \tag{4}$$

for $2 \le i \le n\text{-}1$ and for $2 \le i \le m\text{-}1$.

One iteration consists of sweeping formula (3) throughout all of the interior points of the grid. Successive iterations sweep the interior of the grid with the Laplace iterative operator (3) until the residual term $r_{i,j}$ on the right side of equation (3) is "reduced to zero". The speed of convergence for reducing all of the residuals $\{r_{i,j}\}$ to zero is increased by using the method called successive overrelaxation (SOR). Such method uses the iteration formula:

$$u_{i,j} = u_{i,j} + \omega\left(\frac{u_{i+1,j} + u_{i-1,j} + u_{i,j+1} + u_{i,j-1} - 4u_{i,j}}{4}\right) \tag{5}$$

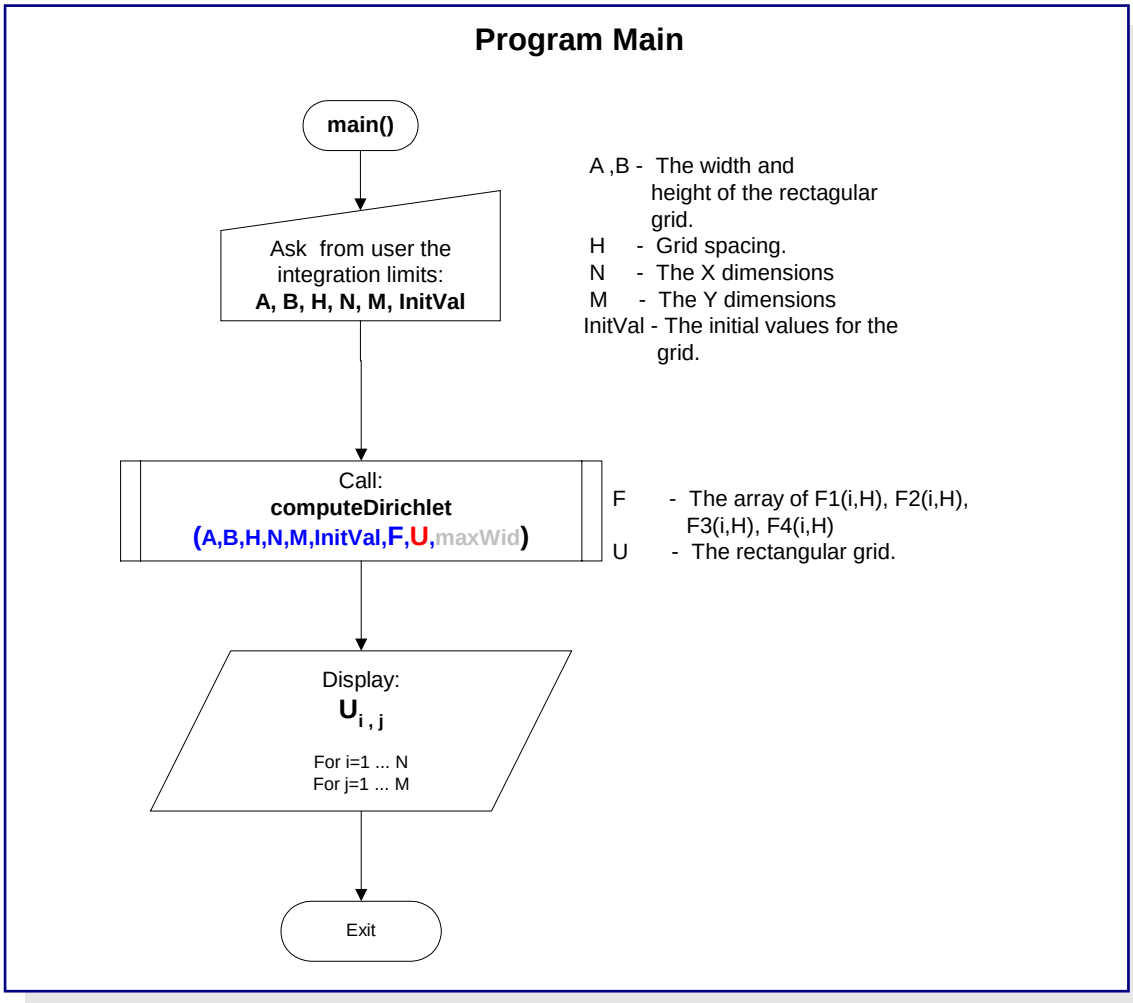
or

$$u_{i,j} = u_{i,j} + \omega\, r_{i,j}$$

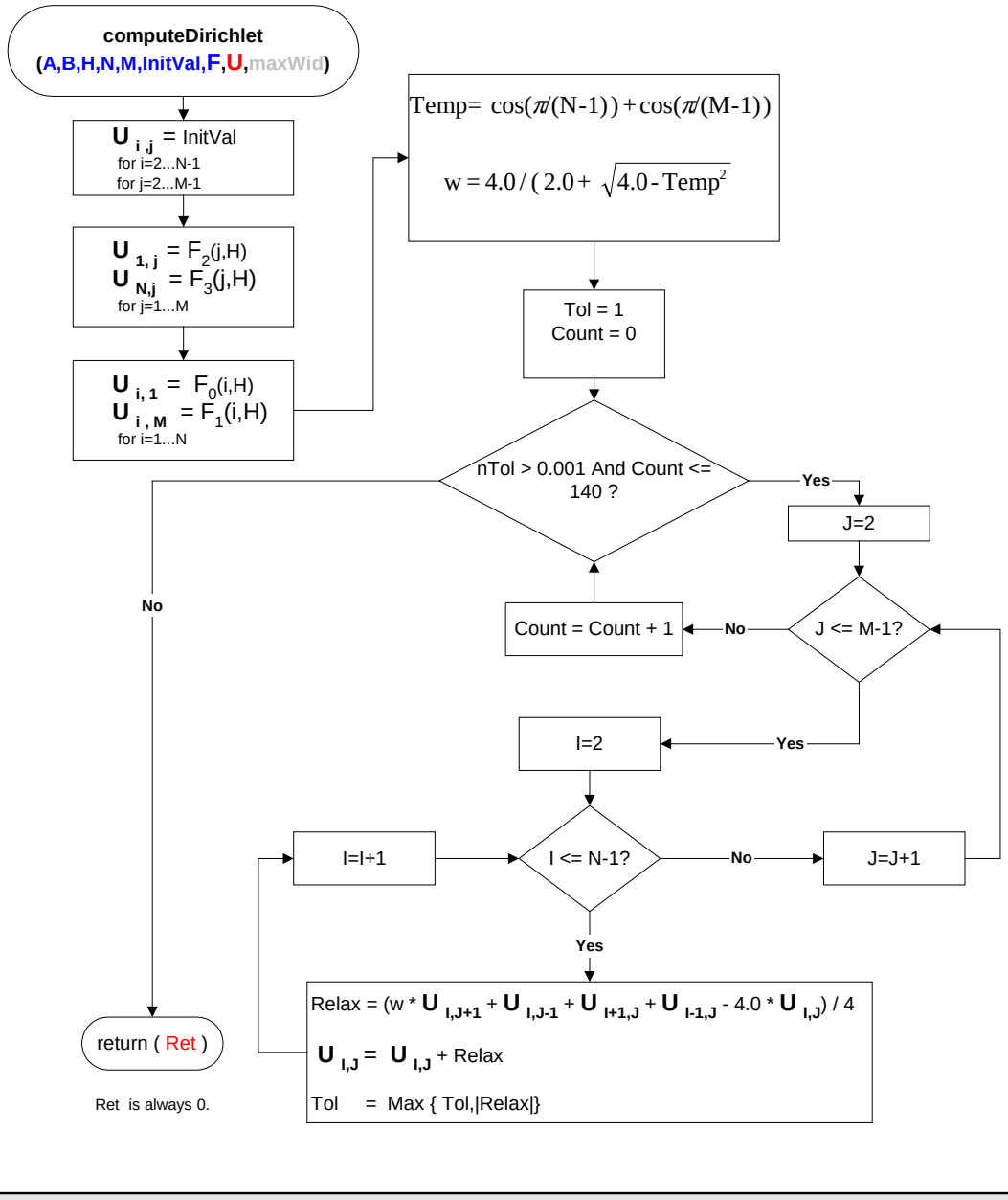
which will be used in place of (3). More information on the SOR can be found on pp. 576-577 of the textbook.

Algorithm 10.4 (Dirichlet Method for Laplace's Equation). To approximate the solution of $u_{xx}(x, y) + u_{yy}(x, y) = 0$ over R = $\{(x, y): 0 \le x \le a, 0 \le y \le b\}$ with u$(x, 0) = f_1(x)$, $u(x,b) = f_2(x)$ for $0 \le$ x $\le$ a anf $u(0, y) = f_3(y)$, $u(a, y) = f_4(y)$ for $0 \le y \le b$. It is assumed that $\Delta x = \Delta y = h$ and that integer n and m exist so that $a = nh$ *and* $b = mh$.

Program Flow



Function for Computing the Dirichlet Method for Laplace's Equation



Program Source Code

```
#include <stdio.h>
#include <conio.h>
#include "DirichletFunc.h"


double  fFunc1(int i,double H);
double  fFunc2(int i,double H);
double  fFunc3(int i,double H);
double  fFunc4(int i,double H);

int main(int argc, char* argv[])
{
    double grid[50][50];

    double wid_A, ht_B;             /* in : Width and height of Rectangle  */
    double space_H;                 /* The grid spacing */
    int xdim_N, ydim_M;             /* dx and dy  */
       double initVal;                  /* The initial value */
    int i,j;
       FUNCF   arrFunc[4];              /* The array of function that will hold F1-F4 */

    printf("\n  ***  Dirichlet Method for Laplace's Equation *** \n\n");

    /*  Ask input from user */
    printf("Please enter rectangular grid width  A   : ");
    scanf("%lf",&wid_A);     fflush(stdin);
    printf("Please enter rectangular grid height B   : ");
    scanf("%lf",&ht_B);      fflush(stdin);
    printf("Please grid spacing H: ");
       scanf("%lf",&space_H);  fflush(stdin);
    printf("Please enter grid x dimension N: ");
    scanf("%d",&xdim_N);  fflush(stdin);
    printf("Please enter grid y dimension M: ");
    scanf("%d",&ydim_M);  fflush(stdin);
    printf("Please initial value for interior grid: ");
       scanf("%lf",&initVal);  fflush(stdin);

       arrFunc[0] = fFunc1;
       arrFunc[1] = fFunc2;
       arrFunc[2] = fFunc3;
       arrFunc[3] = fFunc4;


    /* Compute the Dirichlet*/
    computeDirichlet(wid_A,ht_B,space_H,xdim_N,ydim_M,initVal,arrFunc,&grid[0][0],50);

    /* Display the output */
    i = 1;
    printf("\n=================================================================\n");
    do{
        int nRow;
        for (nRow = 0; (nRow + i) <= xdim_N && nRow < 4; nRow++ )
               printf("   x[%d]  \t",(nRow + i));
        printf("\n-----------------------------------------------------------------\n");

        for ( j = 1; j <= ydim_M; j++ )
        {
            for (nRow = 0; (nRow + i) <= xdim_N && nRow < 4; nRow++ )
               printf("%2.6lf\t", GRID(&grid[0][0],(nRow + i),j,50));

            printf("\n");
        }
```

```
        i+=4;
        printf("\n=================================================================\n");

    }while(i <= xdim_N);


    printf("\nPress any key to continue..");
    getch();

    return 0;
}


double  fFunc1(int i,double H)
{
       /* Ignore the i and H */
       return 180;
}

double  fFunc2(int i,double H)
{
       /* Ignore the i and H */
       return 20;
}

double  fFunc3(int i,double H)
{
       /* Ignore the i and H */
       return 80;
}

double  fFunc4(int i,double H)
{
       /* Ignore the i and H */
       return 0;
}
```

```
#include "DirichletFunc.h"

/* Function name      : computeDirichlet
* Description        : Computes the Dirichlet's method for Laplace's Equation.
* Parameter(s)       :
*     wid_A         [in]  The width  of the rectangular grid. (This is not used.)
*     ht_B          [in]  The height of the rectangular grid. (This is not used.)
*        space_H             [in]  The grid spacing for x & y.
*     xdim_N        [in]  The dimension in X.
*     ydim_M        [in]  The dimension in Y.
*        initVal             [in]  The initial value for ui,j.
*     arrFunc       [in]  Array of functions containing f1(i,H),f2(i,H),f3(i,H),f3(i,H)
*     gridU         [out] The grid.
*     maxWid        [in]  The max width of the grid.
* Return             :
*    0   - Success
*/
int
computeDirichlet(double wid_A,double ht_B,double space_H,int xdim_N,int ydim_M,double
initVal,FUNCF arrFunc[],double* gridU,int maxWid)
{
    int nRet = 0;

    int i,j;

        const static double PI = 3.14159265358979323846264338327 95;
        double   temp;
        double   w;
        double   Relax;
        double   nTol;    /* Iteration tolerance */
        int      nCount;  /* Iteration counter */

    /* Initialize starting values at the interior points */
    for ( i = 2; i <= xdim_N-1; i++ )
    {
        for ( j = 2; j <= ydim_M-1; j++ )   GRID(gridU,i,j,maxWid) = initVal;
    }

    /* Store boundary values in the solution matrix */
    for ( j = 1; j <= ydim_M ; j++ )
    {
        GRID(gridU,1,j,maxWid) = arrFunc[2](j,space_H);
        GRID(gridU,xdim_N,j,maxWid) = arrFunc[3](j,space_H);
    }

    for ( i = 1; i <= xdim_N ; i++ )
    {
        GRID(gridU,i,1,maxWid) = arrFunc[0](i,space_H);
        GRID(gridU,i,ydim_M,maxWid) = arrFunc[1](i,space_H);
    }

    /* The SQR parameter */
    temp =  cos( PI/(xdim_N-1) ) + cos( PI/(ydim_M-1) );

    w = 4.0 / ( 2.0 + sqrt( 4.0 - temp * temp ) );

    /* Initialize the loop control parameters */

    nTol   = 1.0;
    nCount = 0;
```

```
    while ( (nTol > 0.001) && (nCount <= 140) )
    {
       nTol = 0.0;
       for ( j = 2; j <= ydim_M - 1; j++ )
       {
           for ( i = 2; i <= xdim_N - 1; i++ )
           {
             Relax = w * ( GRID(gridU,i,j+1,maxWid) + GRID(gridU,i,j-1,maxWid) +
GRID(gridU,i+1,j,maxWid) +
                           GRID(gridU,i-1,j,maxWid) - 4.0 * GRID(gridU,i,j,maxWid) ) /
 4.0;
             GRID(gridU,i,j,maxWid) += Relax;
             if( fabs(Relax) > nTol ) nTol = fabs(Relax);
           }
       }

          nCount++;
    }

    return nRet;
}
```

Program Output

```
  ***  Dirichlet Method for Laplace's Equation ***

Please enter rectangular grid width  A   : 4
Please enter rectangular grid height B   : 4
Please grid spacing H: .5
Please enter grid x dimension N: 9
Please enter grid y dimension M: 9
Please initial value for interior grid: 70

================================================================
   x[1]           x[2]           x[3]           x[4]
----------------------------------------------------------------
180.000000     180.000000     180.000000     180.000000
80.000000      125.821215     141.172292     145.413997
80.000000      102.112188     113.453406     116.478673
80.000000      89.173614      94.049937      93.920967
80.000000      80.531915      79.651504      76.399890
80.000000      73.302267      67.624119      62.026741
80.000000      65.052833      55.515890      48.867119
80.000000      51.393090      40.519510      35.169120
20.000000      20.000000      20.000000      20.000000

================================================================
   x[5]           x[6]           x[7]           x[8]
----------------------------------------------------------------
180.000000     180.000000     180.000000     180.000000
144.004639     137.478217     122.642417     88.607001
113.126097     103.265620     84.484377      51.785571
88.755286      77.973718      60.243920      34.050961
70.000283      59.630069      44.466679      24.174356
55.215853      46.079647      33.818352      18.179814
42.756788      35.654309      26.547301      14.726560
31.289853      27.233535      21.989990      14.179136
20.000000      20.000000      20.000000      20.000000

================================================================
   x[9]
----------------------------------------------------------------
180.000000
0.000000
0.000000
0.000000
0.000000
0.000000
0.000000
0.000000
20.000000

================================================================

Press any key to continue..
```