Introduction to
Scientific & Engineering Computing
BIL 102FE (Fortran) Course
for
Week 11

Dr. Ali Can Takinacı

Assistant Professor

in

The Faculty of Naval Architecture and Ocean Engineering

80626

Maslak – Istanbul – Turkey

AN INTRODUCTION TO NUMERICAL METHODS IN F PROGRAMS

The main area of application for F programs is the solution of scientific and technological problems. In other words a process which usually involves the solution of mathematical problems by numerical, as opposed to analytical means.

Numerical calculations

The F language was primarily designed to help in the solution of numerical problems, although it is certainly not limited to that purpose.

Consequently, it is extremely important that the writer and the user of such F programs should be aware of the intrinsic limitations of a computer in this area and the steps that may be taken to improve matters.

As it has already been met, a real number is stored in a computer to about six or seven decimal digits of precision with an exponent range of around -10^{38} to $+10^{38}$.

For example e^{88}=1.651636e38, but e^{89} can not be calculated because this would require an exponent of 10 (it exceeds the limit that the computer allows.

Any attempt to store a number whose exponent is too large, as here, will create a condition known as **overflow** and will normally cause program fail at this stage of the processing.

Obviously, once a calculation has overflowed, any subsequent calculations using this result will also be incorrect.

A similar situation arises with the number such as

$$e^{-868}=4.473779e\text{-}38.$$

This situation is known as underflow, which is less serious than the overflow, since the effect is that the number is too close to zero to be distinguished from zero.

Many computers will report this form of error and store this number as zero.

Or some computer systems do report its occurrence as non-fatal error

- Most computers have provision to store floating-point numbers using one of two precisions, usually referred to as **single-precision** and **double precision** with corresponds hardware registers to perform arithmetic operations on them.

- And also the use of the intrinsic function **selected_real_kind** permits more precise control over the precision and exponent range of floating-point number (real variable).

- But it is always remembered that the mechanism for specifying higher precision or exponent range should not be *used blindly* to attempt to get out of numerical difficulties.

Conditioning and stability

A **well-conditioned** problem is one, which is relatively insensitive to changes in the values of its parameters, so that small changes in these parameters only produce small changes in the output.

An **ill-conditioning** problem is one, which is highly sensitive to changes in its parameters, where small changes in its parameters produce large changes in the output.

An example of an ill-conditioned problem is the pair of simultaneous equations,

$$x + y = 10$$

$$1.002x + y = 0$$

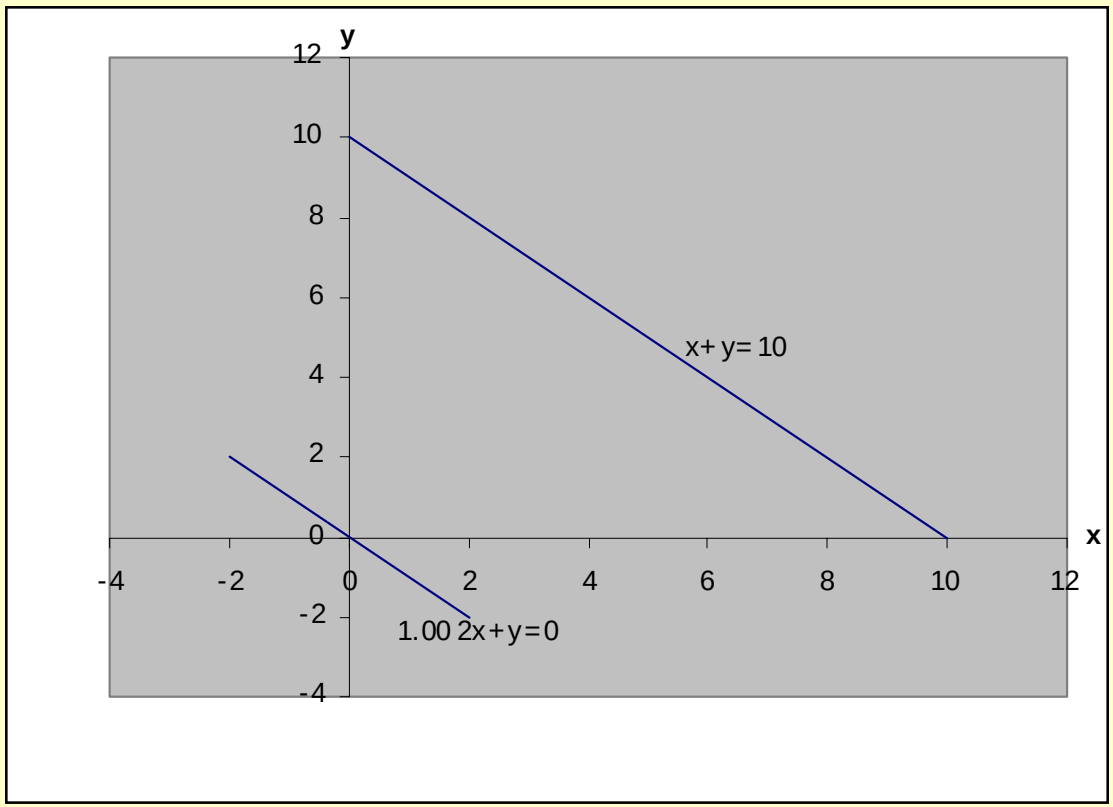
whose solution is clearly x = -5000, y=5010.

However, if some round-off had led to the second equation being expressed as

$$1.001x + y = 0$$

then the solution would have been x = -10000, y = 10010 which is a very great change from the original solution. If the round-off error had led the coefficient of x in the second equation to be 1.00 (to four significant digits) then the problem would have been insoluble.

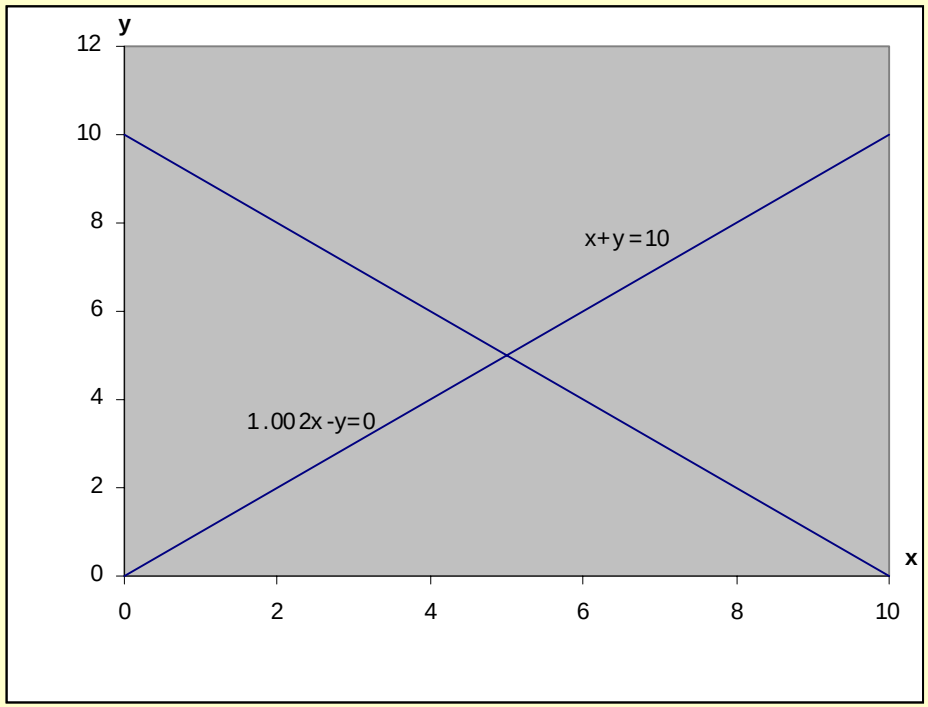
Clearly in this case the reason for this extremely ill-conditioned behaviour is that the two equations represent two straight lines which are almost parallel, and therefore a very small change in the gradient of one will cause a very large movement of their point of intersection.



On the other hand, the two equations

$$x + y = 10$$
$$1.002x - y = 0$$

which have the solution x = 4.995, y=5.002 or x = 5, y=5 respectively. This well-conditioned behaviour is because, in this case the two lines are almost perpendicular to each other.

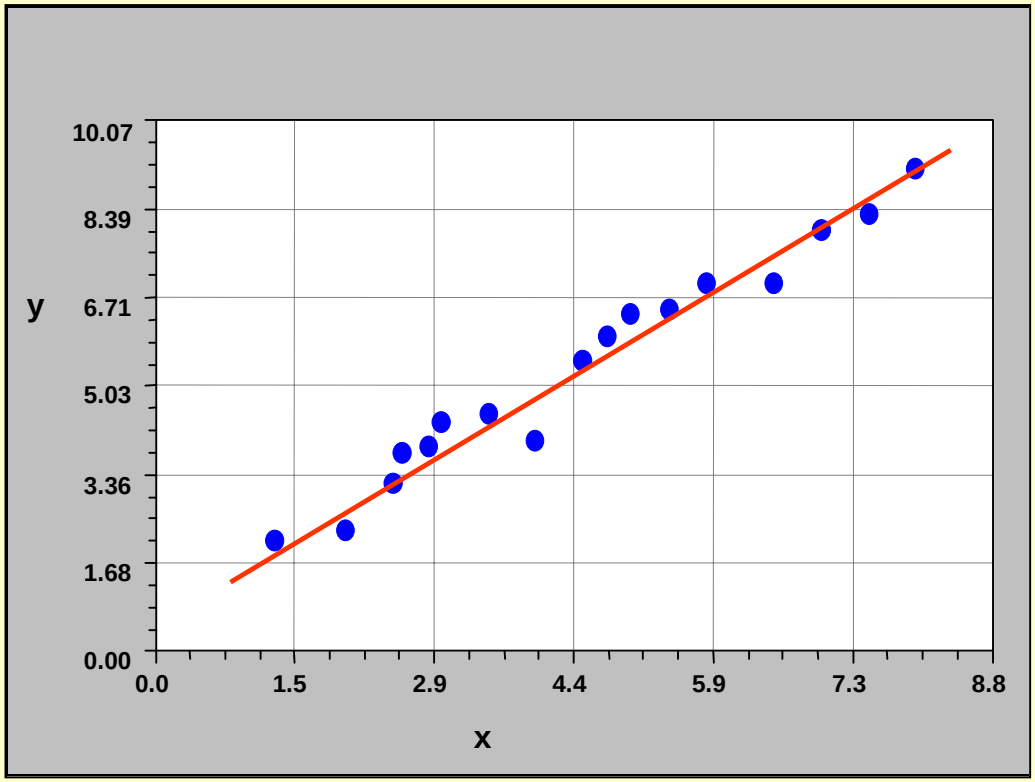


Data fitting by least squares approximation

A frequent situation in experimental sciences is that data have been collected which, it is believed, will satisfy a linear relationship of the form

$$y = ax + b$$

However, due to experimental error, the relationship between the data collected at different times will rarely be identical and can typically be represented graphically as shown in the figure drawn below. Fitting a straight line through the data in such a way as to obtain the fit, which most closely reflects the true relationship, is, therefore, a widespread need. One well-established method is known as the method of least squares.



This method can be applied to any polynomial, or even to more general functions, but for the present but it shall be considered here only the linear case.

The difference between a calculated value and an experimental value y is called residual, and the method of least squares attempts to minimise the sum of the squares of residuals for all the data points.

Some differential calculus, which all are scope of this course, leads to the conclusion that the equation that minimises the square of residuals is when the two coefficients *a* and *b* are defined as follows:

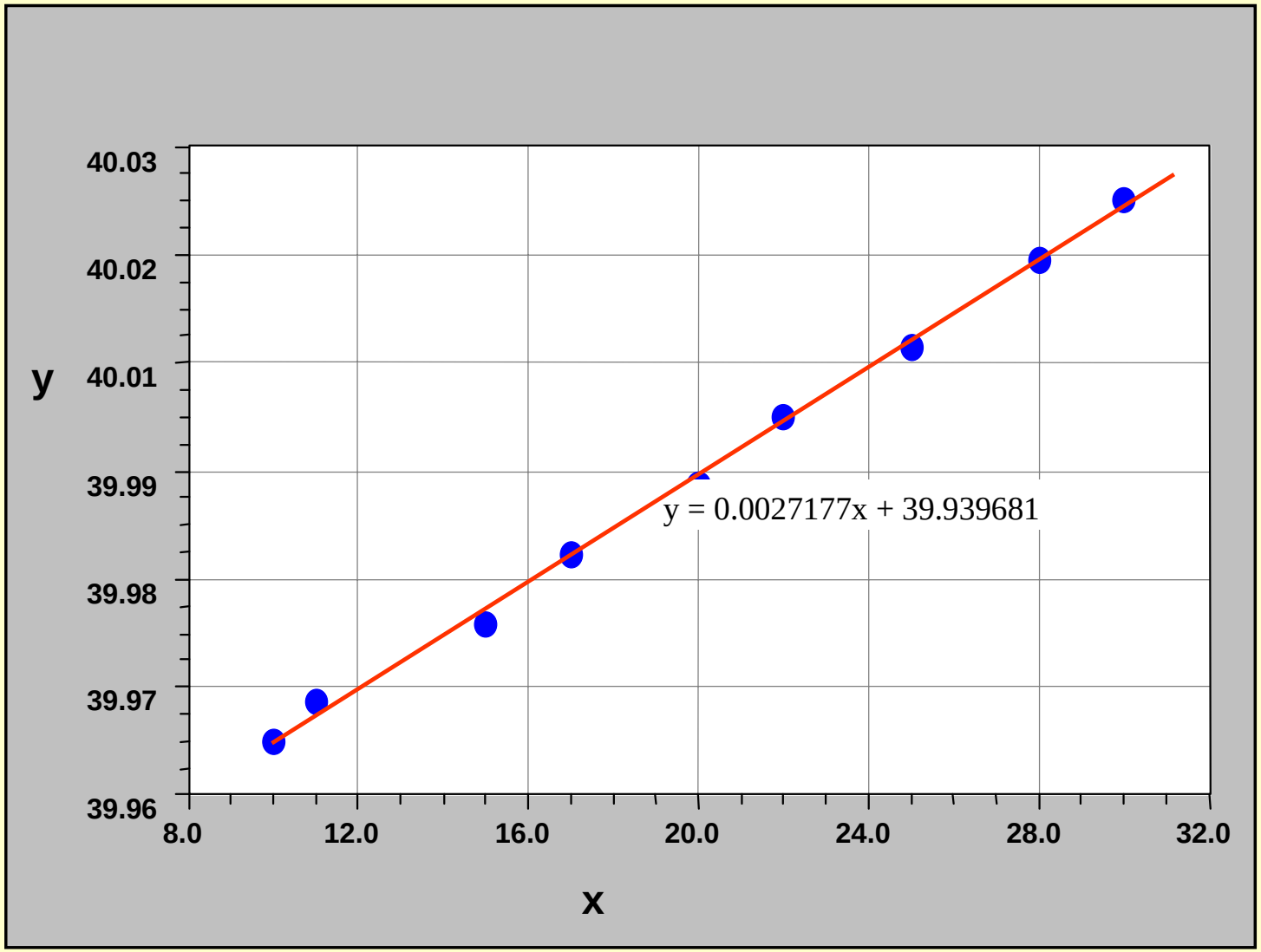
$$a = \frac{\sum x_i \sum y_i - n \sum x_i y_i}{\left(\sum x_i\right)^2 - n \sum x_i^2} \qquad b = \frac{\sum y_i - a \sum x_i}{n}$$

The value of the sum of the squares of the residuals, often referred to as simply the **residual sum**, can be a good guide as to how closely the equation fits the data. If it is a perfect fit, then all data points will lie on the line and the residual sum will be zero.

Example: Figure drawn below shows the result obtained from an experiment to calculate the linear equation of it.

x	y
10	39.967
11	39.971
15	39.979
17	39.986
20	39.993
22	40.000
25	40.007
28	40.016
30	40.022

```
      subroutine least_squares_line(n,a,b,x,y)
!     this subroutine calculates the least squares fit
       line
!     ax + b to x-y data pairs
      real, dimension(n), intent(in) :: x, y
      integer, intent(in) :: n
      real, intent(out) :: a, b
!     local variables
      real :: sum_x, sum_y, sum_xy, sum_x_sq
!     calculate sums
      sum_x = sum(x)
      sum_y = sum(y)
      sum_xy = dot_product(x, y)
      sum_x_sq = dot_product(x, x)
!     calculate coefficients of least squares fit line
      a=(sum_x*sum_y-n*sum_xy)/(sum_x**2 - n*sum_x_sq)
      b = (sum_y - a*sum_x) / n
      end subroutine least_squares_line
```


40.03
40.02
40.01
39.99
39.98
39.97
39.96
y
8.0
12.0
16.0
20.0
24.0
28.0
32.0
x
y = 0.0027177x + 39.939681

Iterative solution of non-linear equations

In this section it shall be started to investigate methods to solve the equation f(x) = 0 *numerically*.

Numerical methods are usually based on calculating an approximation to the true value of a root (or zero) of the equation

$$f(x) = 0$$

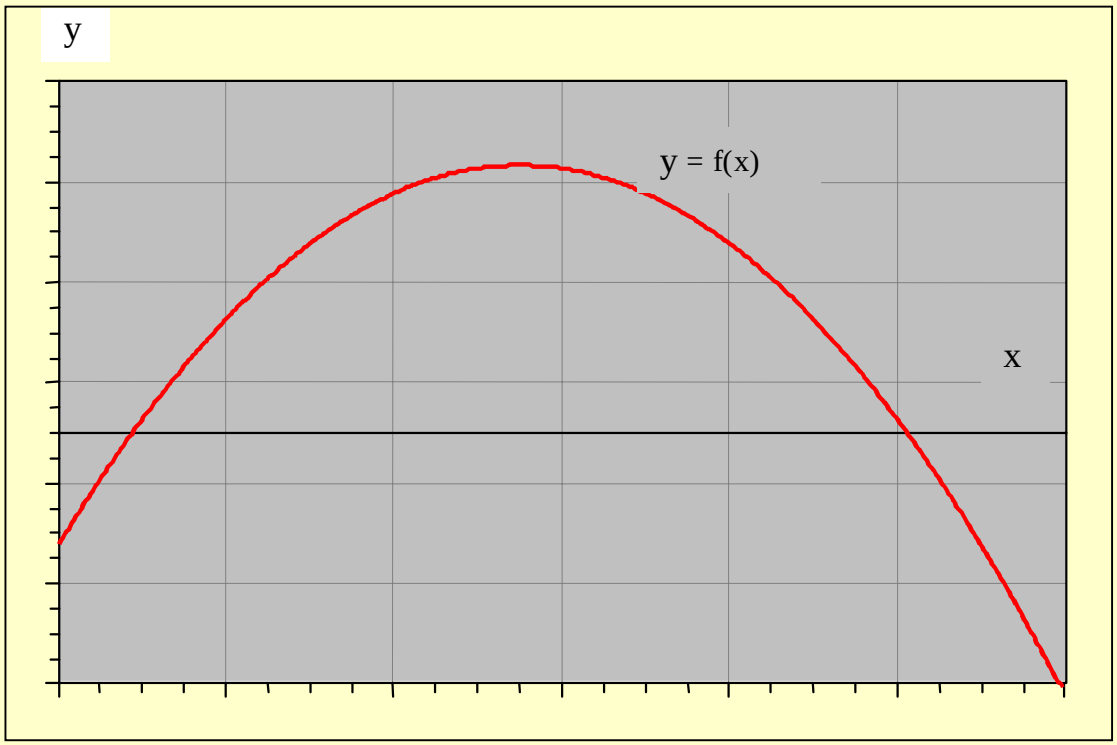
and then successively refining this approximation until further refining would achieve no useful purpose.

Iterative solution of non-linear equations

Numerical methods are usually based on calculating an approximation to the true value of a root (or zero) of the equation

$$f(x) = 0$$

and then successively refining this approximation until further refining would achieve no useful purpose.



Last figure shows the graphical representation of a continuous function $y = f(x)$, and it is clear that the roots of the equation

$f(x) = 0$

are the values of x at which the curve intersect the x-axis. This leads us to a powerful approach to calculating these roots, based on the observation that if $f(x)<0$ and $f(x_j)>0$, then there must be at least one root in the open interval $x_i < x < x_j$. There may be more than one root in the interval.

The **bisection method** uses this fact by then evaluating the value of f(x) at the point midway between x_i and x_j and then repeating the process until the value of x is sufficiently close to the true value of the root.

As in all **iterative methods**, the **convergence criteria** are:

1. The magnitude of the function should be less than ε.

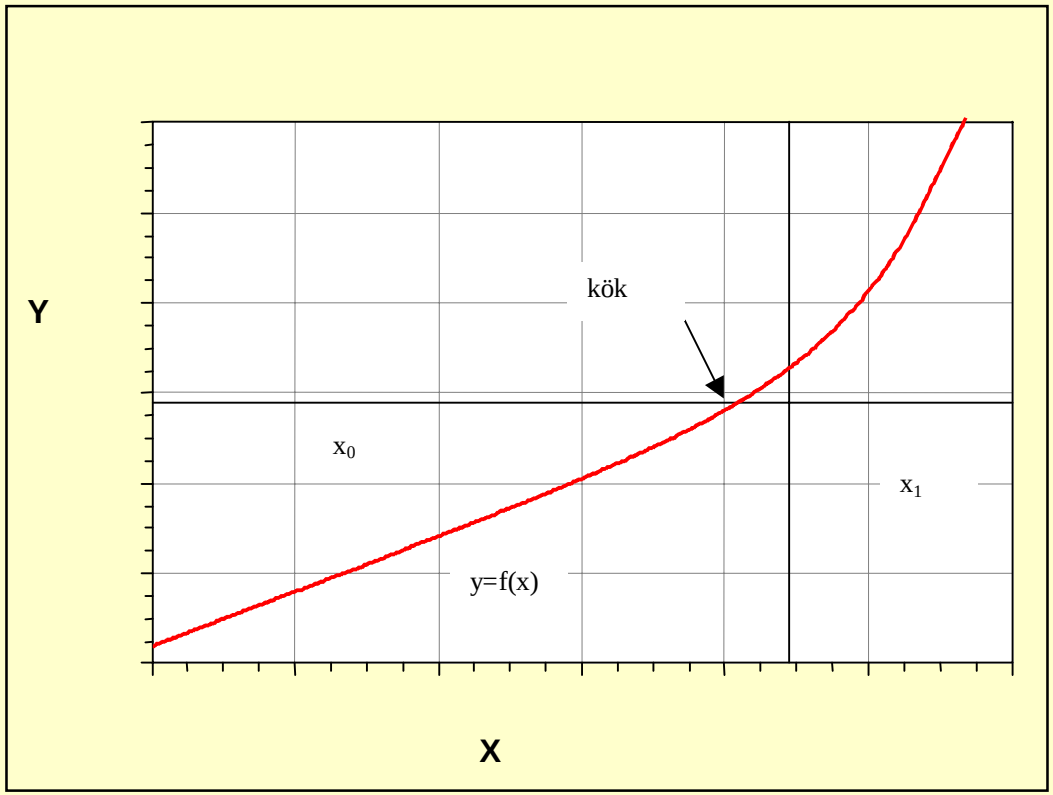
2. The error , where x_t is the true value of the root, should be less than ε.

3. The difference between successive approximations should be less than ε.

Different methods will use different criteria to terminate the iteration.

In the case of bisection method, it is clear that, at each step, the interval, which surrounds the true value of the root is halved. For example, if the two initial values $f(x_0)$ and $f(x_1)$ have opposite signs, the root must lie between them as shown in the figure, and the value of $f(x_2)$ is calculated, where

$$x_2 = \frac{x_1 - x_0}{2}$$



If the sign of $f(x_2)$ is the same as that of $f(x_0)$, then the root must lie in the interval $x_2<x<x_1$, while if it is opposite than the root must lie in the interval $x_0<x<x_2$. In either case the new interval is the half size of the first one. If $f(x_2) = 0$, then, iteration can be stopped since the root is found exactly.

After n iterations the interval containing the root will, therefore, be of size t, where

$$t = \frac{x_1 - x_0}{2^n}$$

The true root must, therefore, differ from any point within this interval by no more than and, in particular must differ from the mid-point of this interval by no more than t/2. Rather suprisingly, therefore, the use of criterion 2 can be stopped the iteration.

Because of the assumption made here, the two initial values, x_0 and x_1, between which the root lies, must be introduced.

Example: Write a program to find the root of the equation f(x)=0 which lies in a specified interval. The program should use a function to define the equation being solved, and the user should input the details of the interval in which the root lies and the accuracy required.

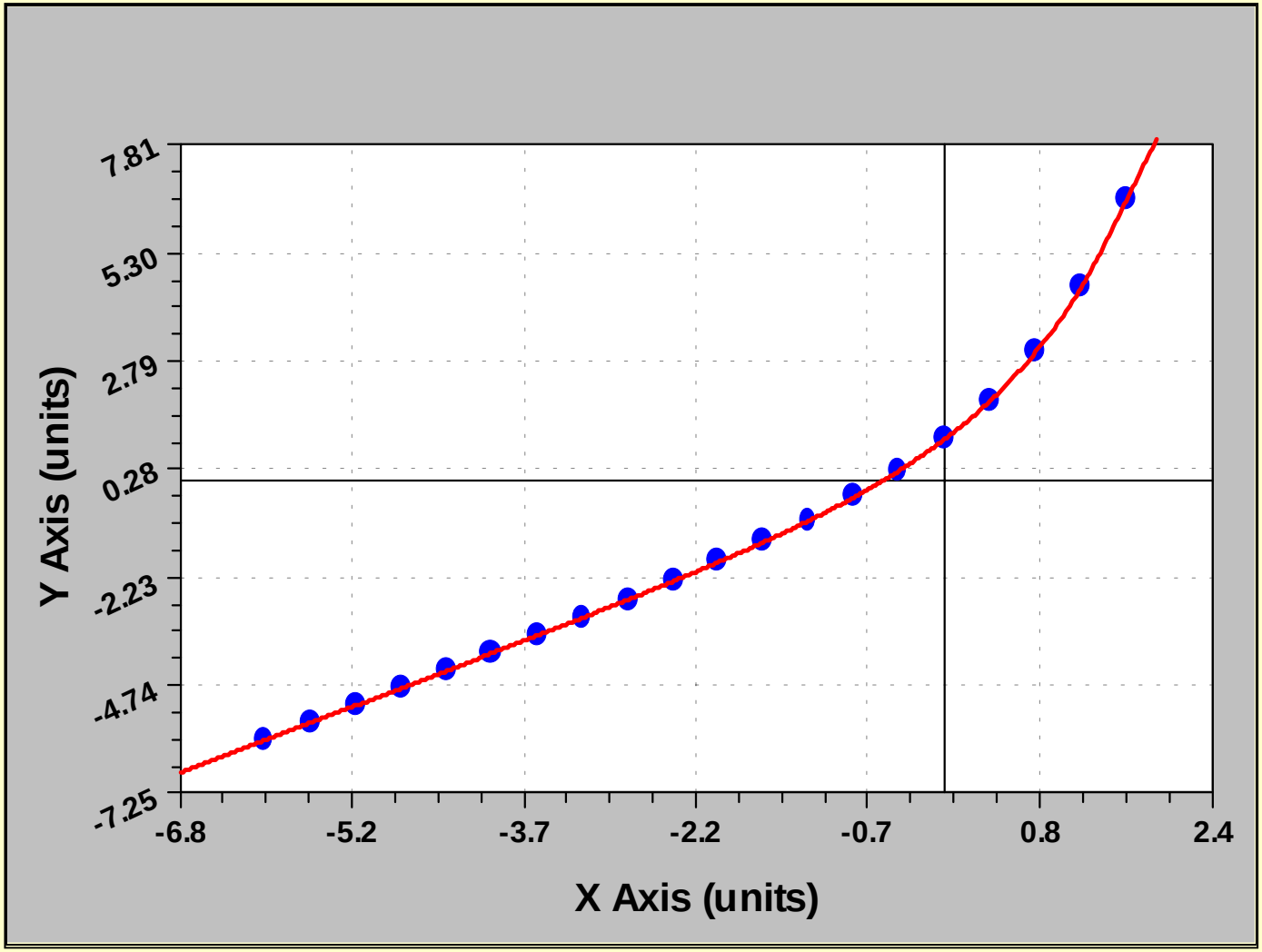
Analysis: A structure plan might be as follows:

```
1.  Read range (left (x0) and right (x1)) and tolerance
2. Call subroutine bisect to find a root in
   the interval (left, right)
3. If root found then
   3.1    Print root
          otherwise
   3.2  Print error message
```

```
Subroutine Bisect
   root: the root found
   delta: the uncertainty in the root (it will nor exceed
tolerance)
   error: status indicator

1.  If left < right then
2.  If x_left and x_right do not bracket a root then
    2.1    Set error = -1 and return
3.  Repeat indefinitely (until convergence criteria met)
    3.1    Calculate mid-point (x_mid) of interval
    3.2    If (x_mid-x_left) < tolerance then exit with
           root = x_mid
    3.3    Otherwise, determine which half interval the root
            lies in and set x_left and x_right appropriately
           3.3.1. If f(x_mid) = 0 then exit with root = x_mid,
                  delta = 0, and error = 0
Otherwise
           3.3.2. If f(x_left)*f(x_mid) is less than 0 then
                  3.3.2.1.  If f(x_left) and f(x_mid) have
                            opposite signs, so
                            set x_right to x_mid.
                        Otherwise
                  3.3.2.2.  f(x_left) and f(x_mid) have the
                            same sign, so
                            set x_left to x_mid
```

The variation of the function is:



The solution of x + ex = 0 using program bisect is:

```
-10.000000   10.000000
-10.000000     .000000
 -5.000000     .000000
 -2.500000     .000000
 -1.250000     .000000
  -.625000     .000000
  -.625000    -.312500
  -.625000    -.468750
  -.625000    -.546875
  -.585938    -.546875
  -.585938    -.566406
  -.576172    -.566406
  -.571289    -.566406
  -.568848    -.566406
  -.567627    -.566406
  -.567627    -.567017
  -.567322    -.567017
  -.567169    -.567017
  -.567169    -.567093
  -.567169    -.567131

 The root is    -.567141
```

Complex variables

In addition to the real and integer numeric data types, F contains a third intrinsic numeric data type, which called complex. It consists of two parts – a **real part** and an **imaginary part**. The intrinsic **`complex`** type stores a complex number in two consecutive numeric storage units as two separate real numbers – the first representing the real part and the second representing the imaginary part.

A complex variable is declared in a type specification statement of the form

complex :: name1, name2

while a complex constant is written as a pair of `real` literal constants, separated by a comma and enclosed in parentheses:

```
(1.5, 7.3)
(1.59e4, -12e-1)
(19.0, 2.5e-2)
```

The complex data type is, thus, a composite type and can be thought of as being an intrinsic version of a derived type. There are three intrinsic functions:

real(z)	where z is complex, delivers the real part of z
aimag(z)	where z is complex, delivers the imaginary part of z
cmplx(z)	if a is real, delivers the complex value (real(a), 0.0) if a is integer, delivers the complex value (real(a), 0.0) if a is complex, delivers the complex value a
cmplx(x,y)	where x and y must be integer or real, delivers the complex value (real(x), real(y))
conjg(z)	where z is complex, delivers the complex conjugate (x,-y), where x and y are the real and imaginary parts of z, respectively

All three functions are elemental, and their arguments can, therefore, be either scalar or array-valued.

The complex number (x, y) is written x+iy, where $i^2 = -1$.
Both real and integer numbers may be combined with complex numbers in a mixed-mode expression, and the evaluation of such a mixed-mode expression is achieved by first converting the real or integer number to a complex number with a zero imaginary part.
Thus, if `z1` is the complex number (`x1`, `y1`), and `r` is a real number then

```
r*z1
```

is converted to

```
(r,0.0)*(x1,y1)
```

which will be evaluated as

```
    (r*x1,r*y1)
```

Similarly, if `i` is an integer, then

```
i+z1
```

is evaluated as

```
(real(i)+x1,y1)
```

```
    program complex_arithmetic
    complex :: a,b,c
!   read two complex numbers
    read *, a,b
    c = a*b
!   print data
    print *,"a=",a
    print *,"b=",b
    print *,"c=",c
    end program complex_arithmetic
```

input:
(12.5,8.4),(6.5,9.6)

output:
a= (12.500000,8.400000)
b= (6.500000,9.600000)
c= (6.100004E-01,174,600000)