Subroutines

- Another example of a subprogram is a subroutine.
- Remember that the difference in a function and subroutine is that the subroutine allows multiple (or no) outputs whereas a function returns only one value to its calling program. Both allow multiple inputs.

```
subroutine name (  input and output variables  )
          declarations

          statements

end subroutine name
```

- Since a subroutine can have multiple outputs, we can not access it in the same way we did a function. Recall for a function we could invoke the subprogram by a statement like

```
centimeters = convert_to_centimeters( arguments)
```

and the single output of the function `convert_to_centimeters` is returned in the value `centimeters`.

- Subroutines must be invoked or called through the use of the `call` statement
- Recall that we used the `call` statement when we invoked fortran's random number generator
- The syntax for the `call` statement is simply

  `call` subroutine name (list of arguments)

- The arguments can be in any order, i.e., you can mix input and output variables.
- As in the function subprogram, the number of arguments and their declared type must be the same in the calling argument and the subprogram definition.

- Since we can have multiple inputs and outputs to a subroutine it is often useful to declare a variable as an input variable, output variable or both. The main reason for this is clarity, readability, and debugging.
- This is done through the use of the specifier `intent` in the declaration statement.
- Examples of declarations in a subroutine:

```
real, intent(in) ::  xk
integer, intent(out) ::  iteration_number
real, intent(inout) ::  x
```

- You will not get a compiler error if you fail to use the intent specifier.
- If you declare a variable as `intent(in)` and you attempt to modify it in the subroutine, then you will get an error.
- When you use `intent(inout) ::  x` this means that you are passing `x` into the subroutine and it is modified there and the new value returned.

For each of the examples below, determine if a we should write a function or a subroutine to perform the necessary calculations.

Remember that if you only have one output you can choose to use either a function or a subroutine. However, if you have multiple outputs (or no outputs) then you must use a subroutine

1. Write a subprogram to determine the area of a circle given its radius.
2. Write a subprogram to determine the area and perimeter of a circle given its radius.
3. Write a subprogram to determine the value of $f(x) = x^3 - 2x$ and its first derivative given the point x.
4. Write a subprogram to output (i.e., print) the values of three real numbers x, y, z

Example - A subprogram to return the value of $f = x^3 \sin(\pi x)$ and its first derivative

- In this example we want to input the point to evaluate the function and its derivative at.
- Since we have 2 outputs we need to use a subroutine instead of a function.

```
subroutine evaluate_f ( x, f_at_x, fp_at_x )

implicit none

real, intent(in) ::  x

real, intent(out) ::  f_at_x

real, intent(out) ::  fp_at_x

real, parameter ::  pi = 3.14159

f_at_x = ( x**3) * sin (pi * x)
```

```
fp_at_x = 3.0*x*x* sin (pi * x) + pi *( x**3) * cos (pi * x)

end subroutine evaluate_f
```

Calling statement:

```
call evaluate_x ( x, f, fp )
```

where `x` has been assigned a number.

The three variables `x, f, fp` must be declared as real in the calling program.

Note that the names don't have to match in the calling statement and the sub-program statement; just the number of arguments and their types much match.

Decide whether the calling statements and subprogram statements match

`call function g(x, y)` where x, y are declared real

`function g(x,y) result (value)` where g, x, y are declared real

`call subroutine convert_temperature(flag, temp_in, temp_out)` where `flag` is an integer and `temp_in, temp_out` are real

`subroutine convert_temperature(test, t1, t2)` where `test, t1, t2` are real

`z= g(x, y)` where x, y, z are declared real

`function g(x,y) result (value)` where x, y,value are declared real

Oftentimes our main or driver programs consists mainly of calls to invoke subprograms. This way our code is compartmentalized which allows us to debug each part and then never change that section of code. In addition, a routine may be invoked in several places in the code by simply calling it with the correct arguments. It also helps in making the code readable, i.e., determining what the code does. The following example illustrates this.

```
program euclidean_length

!  This program calls a subroutine to input a point in R^3
!  Then it calls a function to find the Euclidean length of
!  the vector from the origin to the given point
!  Lastly, the program calls a subroutine to output this info.
!
!**************************************************
implicit none
!
real :: x,y,z
real :: length
```

```
!
!*************************************************
! Input the coordinates of a point in 3-space
!
  call input_point (x, y, z )
!
! Find the Euclidean length of the vector (x,y,z) given by
!   sqrt ( x^2 + y^2 + z^2)

  length = find_length (x, y, z)
!
!  Output the vector and its Euclidean length
!
  call output (x,y,z, length )
!
!*************************************************
!*************************************************
    CONTAINS
!*************************************************
```

```
!*****************************************************

  subroutine input_point (u, v, w )
!
!  This subroutines asks the user to input the coordinates
!  of a point in R^3
!
!*****************************************************
  real, intent(out) :: u, v, w

  print *, "input a point in 3-space "
  read *, u, v, w
!
  end subroutine input_point
!
!*****************************************************

  function find_length (x,y,z ) result (value)
!
```

```
!  This function find the Euclidean length of a
!  vector in R^3

  real :: x,y,z
  real :: value

  value = x*x + y*y + z*z
  value = sqrt (value )

  end function find_length

!*************************************************
!
  subroutine output (x,y,z, length)
!
! This subroutine outputs a given point (x,y,z) and its
! Euclidean length
!
  real, intent(in) :: x,y,z
```

```
real, intent(out) :: length

print *, "The vector in R^3 with endpoint ", "(",x,y,z,")"
print *, "has Euclidean length", length

end subroutine output

end program euclidean_length
```

Here is an outline of a code `cubic.f90` which we are going to work on today.
It will NOT compile.

```
program cubic

!  This program
!    (1) calls subprogram to input coefficients of cubic
!          u(x) = ax^3 + bx^2 + cx + d
!    (2) calls subprogram to print out cubic
!    (3) for   points x in [0,1]  subprograms are called to
!        (i) evaluate u(x)
!        (ii) evaluate u'(x) and u''(x)
!        (iii) output x, u(x), u'(x) and u''(x)
!
implicit none
!
real :: a, b, c, d
real :: dx, x
real :: u, u_x, u_xx ! the value of cubic and its first 2 derivative
```

```
integer :: k

!  input the coefficients of the cubic polynomial
!    u(x) = ax^3 + bx^2 + cx + d
!
   call input_cubic ( )
!
!  call routine to output the polynomial being used
!

!  DO LOOP
!  For each point x,
!    (i) call subprogram to evaluate cubic at x
!    ii) call subprogram to calculate the first
!          and second derivatives of the cubic at x
!          and return values in cubic1 and cubic2
!  (iii) call subprogram to output the value of the cubic
!         and its first and second derivatives at each of the points
```

```
   dx = 0.25
   x = -dx

   do k = 1, 4

     x = x + dx

     u = eval_cubic (   )

     call eval_der ( )

     call output (x, u, u_x, u_xx    )

    end do

!******************************************************
!******************************************************
   CONTAINS
```

```
!*****************************************************
!*****************************************************

  subroutine input_cubic ( a, b, c, d )

  real :: a,b, c, d
!
! Reads in the coefficients of the cubic ax^3+bx^2+cx+d
!
  print *, "enter the four coefficients of the cubic separated by co
  print *, "enter the coefficient of x^3 first, then x^2, etc."



  end subroutine input_cubic

!*****************************************************
!
  subroutine output_cubic ( a, b, c, d )
  real :: a,b, c, d
!
! Outputs the cubic ax^3+bx^2+cx+d
!

  print *, "The cubic we are considering is"
  print *, a,"x^3 +", b,"x^2+", c, "x+", d
  print *,
  print *,

  end subroutine output_cubic
!
!******************************************************
!
  function eval_cubic

  real :: x
  real :: a,b, c, d
  real :: value
!
! Computes the value of the cubic ax^3+bx^2+cx+d at x
!
  value =

  end function eval_cubic

!******************************************************
!
  subroutine eval_der (x, a, b, c, d, f1, f2 )

  real :: x
  real :: a,b, c, d
  real :: f1, f2   ! first and second derivatives of cubic
!                                  ax^3+bx^2+cx+d at x
!
  f1 = a*(3.0 * x * x) + b*(2.0 *x) + c
  f2 =
  end subroutine eval_der
!******************************************************
!
  subroutine output  ( x, f, f1, f2 )

  real :: x
  real :: f, f1, f2
!
! Outputs the cubic ax^3+bx^2+cx+d and its first and second
!  derivatives at the point x
!
  print *, "The cubic evaluated at x = ", x, "is", f
  print *, "The first derivative of cubic evaluated at x = ", x, "is
  print *, "The second derivative of cubic evaluated at x = ", x, "i
  print *,

  end subroutine output
!
```

```
!*************************************************************
!*************************************************************
  end program cubic
```

Classwork

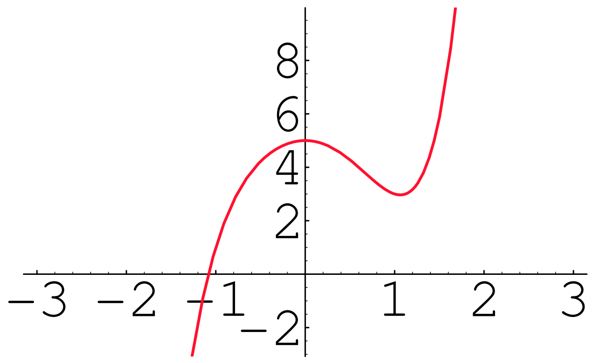
1. Download the code `cubic.f90` It will NOT compile.
2. Make the necessary modifications to the code to make it perform the tasks described in the comments.
3. After it is working add `intent(in)`, `intent(out)` statements in the declarations of the subroutines. Remember that we don't use this in functions because it only has one output, i.e., the variable in `result(value)`.

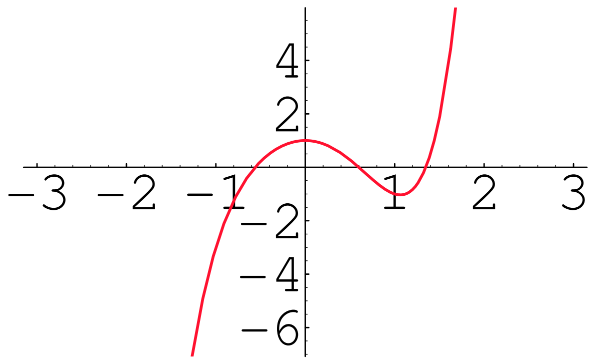
PART II - Solving a Nonlinear Equation

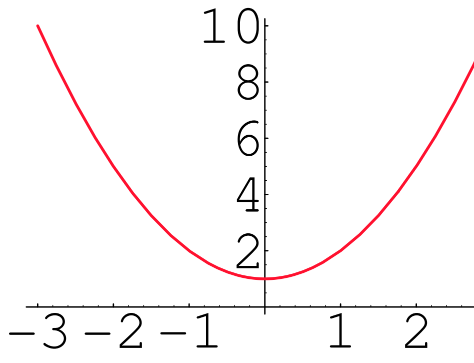
The next problem we want to investigate is finding the root of a single nonlinear equation $f(x) = 0$ (i.e., where the function $f(x)$ crosses the x-axis) such as

$$x^5 - 3x^2 - 3 = 0 \qquad \text{or} \qquad x - \sin x = 0 \qquad \text{or} \qquad e^{x^2} - 4 = 0$$

- A nonlinear equation $f(x) = 0$ may have no roots, only one root, or several roots.







- We know that there is a formula (the quadratic formula) for finding the roots of a quadratic polynomial (which is itself a nonlinear equation).

$$ax^2 + bx + c = 0 \qquad \Rightarrow \qquad x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

- Actually there are also formulas (not as well known) for finding the roots of third and fourth degree polynomials but not for higher degree polynomials.
- Nonlinear equations are much more difficult to solve than linear equations.
- We should not expect to be able to find a formula which gives us the exact answer, but rather we will look at methods which approximate the solution.
- The methods we will look at are called iterative methods.

Iterative Methods

- For an iterative method we start with an initial guess (which is a number in our case), say x^0, and then we will generate a sequence of approximations
$$x^0, x^1, x^2, x^3, \cdots$$
which we hope will tend to the exact root of our problem.

- For example, we might generate the sequence of iterates
$$2, \quad 1.5556, \quad 1.2586, \quad 1.0851, \quad 1.01324, \quad 1.0039, \quad 1.0023$$
In this case it appears that our approximations are approaching 1.0

- If x^n approaches our exact solution, say x^*, as $n \to \infty$ when we say the method converges.

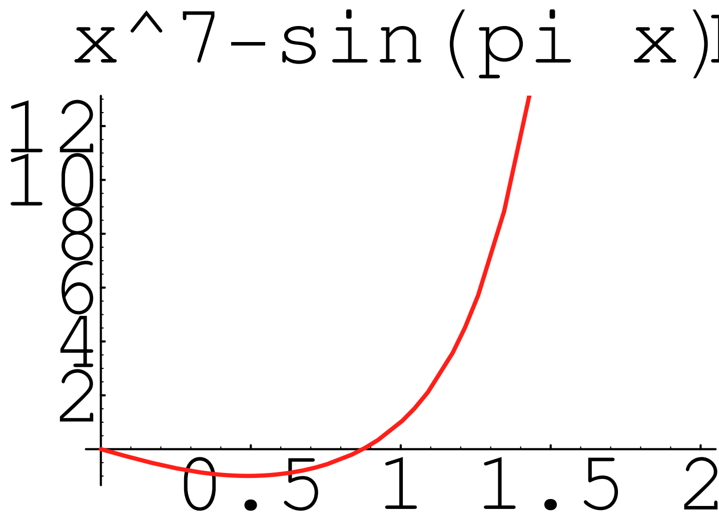
- As another example, we might generate the sequence
$$2, \quad 2.5556, \quad -5.3586, \quad -11.0851, \quad 41.01324, \quad 121.0039$$

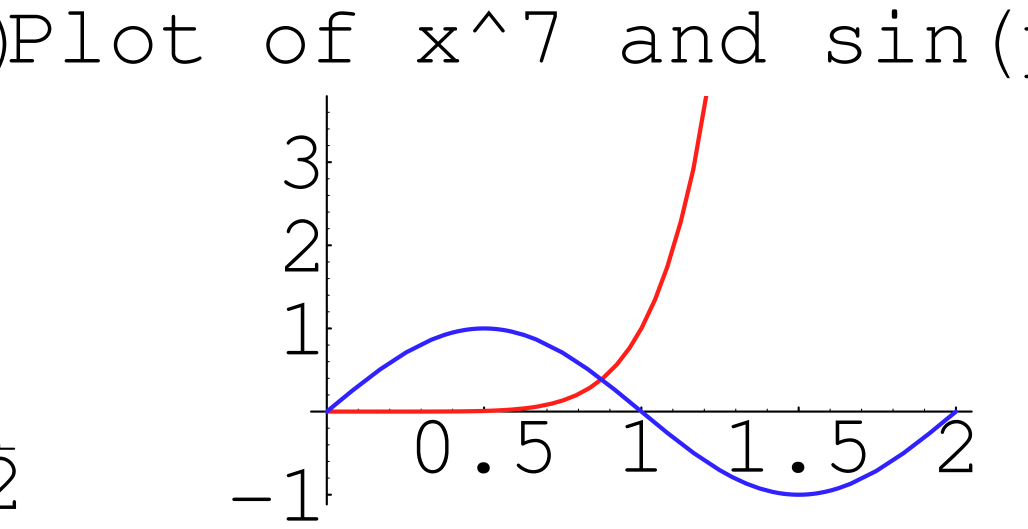
In this case our approximations are not tending to some value, so we say that the method does not converge or diverges.

- Several issues arise when we use iterative methods
  - How do we get our starting guess?
  - Is the method sensitive to the starting guess? That is, does it work for any starting guess?
  - What if there is more than one root?
  - When do we stop?
  - If the approximations are tending to some value, how fast are they tending to the value?

- How do we get our starting guess?

1. We could graph the function; e.g., for the root of $f(x) = x^7 - \sin \pi x$ in (0,2) we could graph this function and see approximately where it crosses the x-axis or alternately we could graph x^7 and $\sin \pi x$ and see where they intersect.
2. Physical intuition
3. Random





The speed at which an iteration converges will be important to us.

For example, if we have the following two sets of approximations generated by different methods both tending to 1.0, which one do we prefer?

$$2, \quad 1.5556, \quad 1.2586, \quad 1.0851, \quad 1.01324, \quad 1.0039$$

$$2, \quad 1.9576, \quad 1.8386, \quad 1.7651, \quad 1.51324, \quad 1.3033$$

We need a way to determine how fast (numerically) our approximations are converging.

How do we know when to stop an iterative method?

- When we program iterative methods we always want to have a maximum number of iterations.
- However, if our root is near say 4 and we have starting guesses of 2 and 100, both yielding convergent sequences, then we would guess that the one with a starting guess near the root (i.e., 2) would get to the root in fewer iterations.
- Consequently, we need some way to check to see if our answer is "good enough".
- Since $x^n \to x^*$ as $n \to \infty$ (where x^* is the root we are looking for) we expect the terms x^m, x^{m+1} (for sufficiently large m) to be close together. So we could check
$$|x^{m+1} - x^m| \leq \text{ prescribed tolerance}$$
- Although this is commonly used, it can sometimes lead to problems when the iterates themselves are small.

- It is actually better to look at a relative error such as

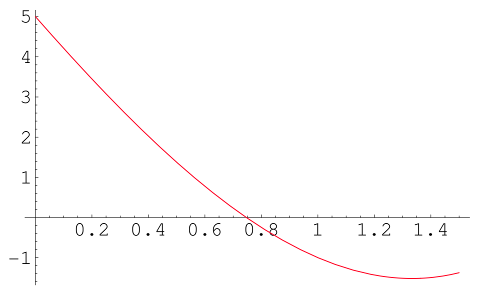
$$\frac{|x^{m+1} - x^m|}{|x^{m+1}|} \leq \text{ prescribed tolerance}$$

- Another approach to terminate an iterative process is when we know a theoretical bound on the error. In this case we can bound it by some tolerance. This error bound is typically not known in general but for the first algorithm we investigate it is known.

The Bisection Method

One of the simplest method for finding a root of a continuous function $f(x)$ is the bisection method.

- We first find an interval $[a, b]$ where the product $f(a)f(b) < 0$
  - This means that $f(a)$ and $f(b)$ are of opposite signs.
  - Recall from calculus that the Intermediate Value Theorem guarantees that a continuous function $f(x)$ must pass through zero somewhere in the interval (a, b) if $f(a)$ and $f(b)$ are of opposite signs. Of course it may have more than one root but it has at least one. In the figure below we are guaranteed that this function has a root in [0,1.4] since $f(0) > 0$ and $f(1.4) < 0$.



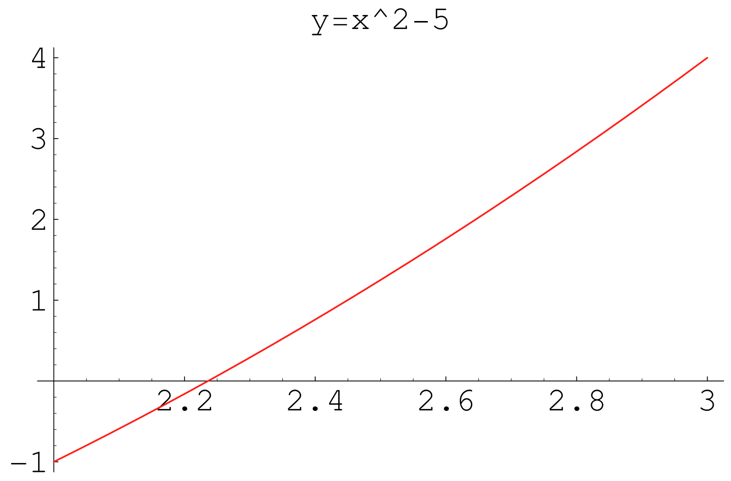
- So if we start with an interval $[a, b]$ where $f(a)f(b) < 0$ then we are guaranteed that a zero or root of $f(x)$ lies in (a, b).
- The basic idea behind the bisection method is that we take the midpoint of the interval $[a, b]$ and then check to see if the root is in $[a, \frac{a+b}{2}]$ or in $[\frac{a+b}{2}, b]$.
  - To do this we simply evaluate $f(\frac{a+b}{2})$ and compare the sign with that of $f(a)$ or $f(b)$.
  - We choose the interval which has function values at the endpoints of opposite signs.
- This process can be repeated and we are guaranteed that we can get as close to a root as we want by continually halving the interval.

Example

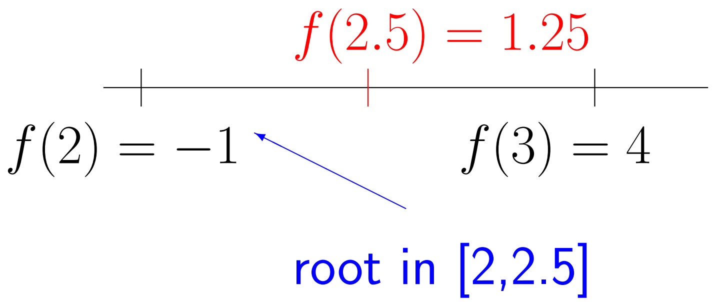
We can approximate the square root of 5 by finding a root of the equation

$$f(x) = x^2 - 5 = 0$$

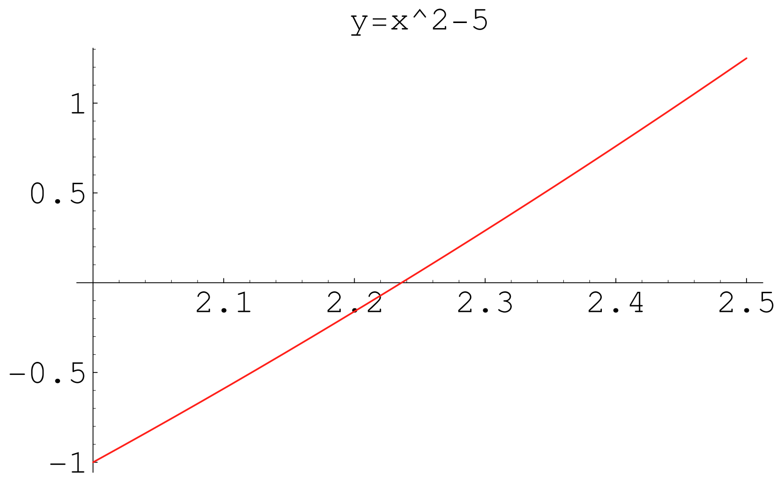
- We know that $2 < \sqrt{5} < 3$ and since $f(2) = -1 < 0$ and $f(3) = 4 > 0$, we can take the interval $[2, 3]$.

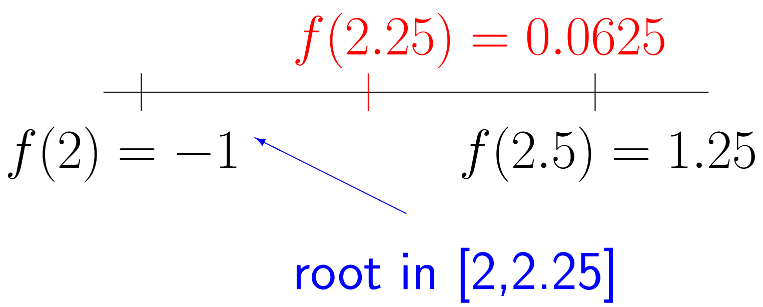


- Our first approximation would be the midpoint $x = 2.5$
- Since $f(2.5) = 1.25$ we see that $f(2) = -1$ and $f(2.5)$ are opposite signs (i.e., $f(2)f(2.5) < 0$) so we take [2,2.5] as our next interval.

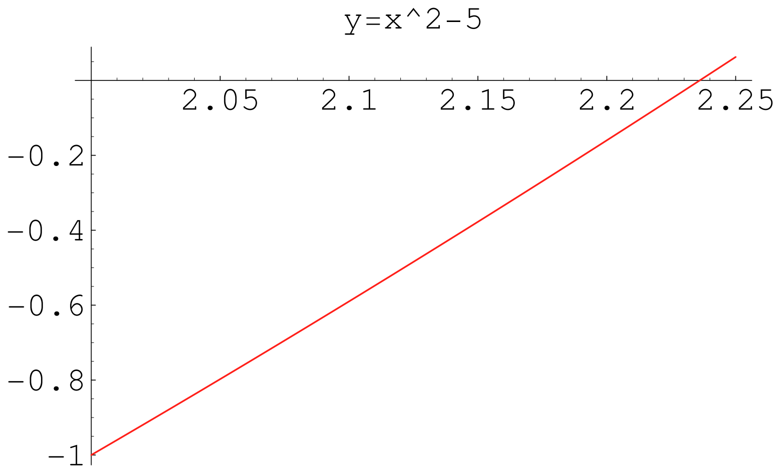


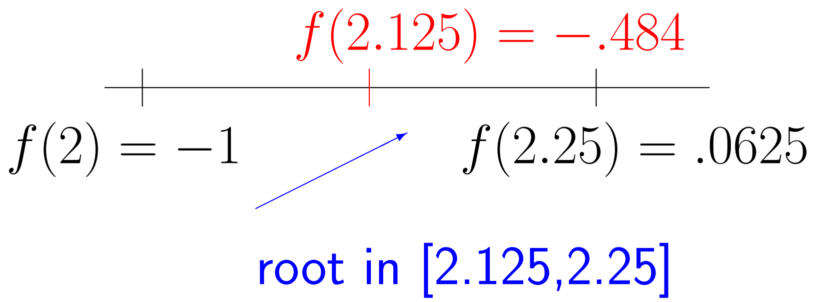
- Our next approximation is (2+2.5)/2= 2.25
- Since $f(2.25) > 0$ and $f(2) < 0$ we choose our next interval to be [2,2.25]





- Our next approximation is (2+2.25)/2= 2.125
- Since $f(2.125) < 0$, $f(2) < 0$ and $f(2.25) > 0$ we choose our next interval to be [2.125,2.25]





- Continuing in this manner, we form a sequence

$$2.5, 2.25, 2.125, 2.1875, 2.21875, 2.23438, 2.24219, 2.23828, 2.23633\ldots$$

which converges to $\sqrt{5} = 2.236067977$

How do we know when to stop the Bisection Method?

- When we program iterative methods we always want to have a maximum number of iterations.
- However, in our example, if we started with the initial interval of [2,3] we would expect it to take less steps than if we started with [0,100] although they both satisfy the condition $f(a)f(b) < 0$
- Consequently, we need some way to check to see if our answer is "good enough".
- We can do this if we know something about the error we are making. We don't have this information for every method, but we do for the Bisection Method. If we have an estimate for the error then we can check

  if error $<$ prescribed tolerance then stop

The Error in the Bisection Method

- As an example, let's say that we know the root is in the interval [1,2]; then our next guess is 1.5.
- How far away can 1.5 be from the answer; i.e., what error are we making ? Clearly it has to be < 0.5 from the answer since the root is in the interval $[1, 2]$.
- In general, the error at a particular step is going to be half the length of the current interval.
  - Initially, with an approximation of $\dfrac{a+b}{2}$ the error is less than $\dfrac{b-a}{2}$
  - At the end of the second step the interval is half the size of the previous step so taking its midpoint gives the error is less than $\dfrac{b-a}{4}$
  - At the end of the third step the interval is half the size of the previous step so the error is less than $\dfrac{b-a}{8}$

– In general, at the end of the nth step the error is less than $\dfrac{b-a}{2^n}$

– Note that the Bisection Method will always converge if you start with an interval $[a, b]$ containing the root.

– If there is more than one root in the interval $[a, b]$ to begin with, then you will simply converge to one of the roots.

– We will see that the convergence of the Bisection Method is very slow and that the error does not decrease at each step. The next methods we look at will try to address these problems. However, we may have to give up the guaranteed convergence of the method.

What do we need to write a program to implement the Bisection Method?

- We know that we will need a conditional (like `IF THEN`) to check the error and to see which interval our root is in.
  - Actually we will probably want to do one thing if the product of the two function values equals zero, another if it is less than zero and another if it is greater than zero
  - Consequently we may need a slightly more complicated `IF - ELSE IF - ELSE` construct
- We do not want to "hardwire" the function we are using so we need to write a separate `function` to do this. If we hardwired the function in the code in our example for finding the root of $x^2 - 5$, we would require statements like

```
f_at_a = a*a - 5.0
f_at_b = b*b - 5.0
mid = (a + b) /2.0
f_at_mid = mid*mid - 5.0
```

so if we want to find the root of a different function, then we have to recode all of these statements. Instead we just invoke our function `eval_f` with a statements like

```
f_at_a = eval_f ( a )
f_at_b= eval_f ( b )
f_at_mid = eval_f ( mid )
```

When we change the function we are using we simply change the function and don't have to modify the main program.

`IF-ELSE IF` constructs

We will need `IF-ELSE IF` constructs to write the code for the bisection method because when we compute $f\Big((a+b)/2\Big)$, i.e., at the midpoint of $[a,b]$ we do one thing if $f(a)\ f\Big((a+b)/2\Big) <= 0$ and another if it is > 0 and still another if it equals zero (i.e., we have found the root).

- The simplest type of conditional that we looked at was the situation where we need to test a condition and if it is true, then do something, that is, we have only one alternative. For example,

```
        if ( n > 10 ) stop
```

  – Here the condition we are testing is $n > 10$.

  – If the condition is true, then we terminate the program.

- In the case where we have two alternatives, that is, we need to test a condition and do one thing if it is true and another when it is false, we use the `if-else` construct. For example,

```
if (b*b-4.0*a*c >= 0 ) then
     term = sqrt ( b*b-4.0*a*c )
   else
     print *, " there are no real roots"
end if
```

- Here the condition we are testing is to see if $b^2 - 4ac \geq 0$ which could be part of a code to implement the quadratic formula.
- If it is true, then we can take the square root of this number.
- If it is false (and we are using the quadratic formula to find real roots only) we simply include a write statement to indicate the roots are complex.

- Clearly you can imagine a situation where we have more than two alternatives. In this case we use the `if-else if` construct.
- The syntax for the situation where we have 4 alternatives is described next.

```
if (  logical expression # 1  )    then

     statements

  else if  ( logical expression # 2  )    then

     statements

  else if  ( logical expression # 3  )    then

     statements

  else

     statements

end if
```

- Note that there is no `then` required after the final `else` since there is no `if`.

- Note that if the first expression is satisfied then the remaining statements are not reached; if the first statement is not true but the second is, then none of the remaining statements are checked, etc.

As an example, consider the following conditional written for the situation described below.

Suppose that the temperature, `temp, high_temp, low_temp`*, have been defined and we want to check to see if it is within a normal temperature range defined by the values* `low_temperature` *and* `high_temperature`*. If it is below the normal range, print this fact out and if it is above the normal range, print this fact out.*

```
if (temp > high_temp )   then

    print *, "temperature is above normal high"

  else if ( temp < low_temp )  then

    print *, "temperature is below normal low"

  else

    print *, "temperature is in normal range"

end if
```

As another example, consider a program which reads in the coefficients of two lines

$$ax + by = c$$
$$dx + ey = f$$

Assume that a function routine is available to compute the slope of a line $\alpha x + \beta y = \gamma$.

We want to calculate slope of each line, say m_1, m_2. Then

(i) If the lines are parallel (i.e., slopes are equal) print this out

(ii) If the lines are not parallel determine if they are perpendicular (i.e., the slopes satisfy $m_1 = -\frac{1}{m_2}$)

(iii) If neither is true, print this out.

The section of code to perform this follows, assuming coefficients have been entered.

```
m1 = eval_slope (a,b)

m2 = eval_slope (d,e)

if ( m1 == m2 ) then

    print *, "lines are parallel"

else if (m1 == -1.0 / m2) then

    print *, "lines are perpendicular"

else

    print*, "lines are neither parallel nor perpendicular"

end if
```

Classwork

1. Write a code to have the user input a score (between 0 and 100) which allows fractional scores like 75.5
2. The instructor has set the following scale for the exam.

A	$88 - 100$
B	$77 - 87$
C	$65 - 76$
D	$55 - 64$
F	< 55

   Write an appropriate `if-else if` construct to print out the student's numerical score and letter grade on the exam.

Implementing the Bisection Method

Strategy:

- We will have a function routine which as input has an x value and as output the function value.
- In our algorithm, we will start with an interval, say $[a, b]$ containing the root.
- At the end of each iteration we will have a new interval of half the length of the previous interval, either $[a, \frac{a+b}{2}]$ or $[\frac{a+b}{2}, b]$.
- We will still call this interval $[a, b]$; we will simply modify a or b to be the midpoint.
- Once we have a midpoint of a new interval we will check the error to see if it is less than the tolerance.

- To begin with we need an interval, say $[a, b]$, which brackets the root.
  - The values for a and b should be input by the user
  - The fact that $[a, b]$ brackets the root should be verified by the program; if it is not satisfied then an error message should be output and execution terminated. At this time we can also see if by change $f(a) = 0$ or $f(b) = 0$, i.e., we already found root.

- We will also need a maximum number of steps to do and a tolerance to check for convergence of the method; for now we can "hard wire" these.

- Evaluate function at initial a and b, call them `f_at_a, f_at_b`

- We will have an iteration loop which must accomplish the following
  - Evaluate midpoint, call it `midpoint`
  - Check for convergence; if $\frac{b-a}{2}$ is less than tolerance then stop.
  - Evaluate f at midpoint, call it `f_at_midpoint`

- Evaluate (for example) $f(a)f(\frac{a+b}{2})$ call this number `value`
- `if ( value ) == 0` terminate because root has been found
- `if ( value ) < 0` then we know that the root is in $\frac{b-a}{2}$ is less than tolerance then stop.

  $[a, \frac{a+b}{2}]$ so move the endpoint b to the midpoint. Also, since we have already evaluated f at the midpoint we might as well use it; i.e.,

  ```
  b= midpoint
  f_at_b = f_at_midpoint
  ```
- `else if ( value ) > 0` `then we know that the root is in` $[\frac{a+b}{2}, b]$ `so move the endpoint` a `to the midpoint; i.e.,`

  ```
  a= midpoint
  f_at_a = f_at_midpoint
  ```

Code for the Bisection Method

```
  program bisection_method
!
!  This program approximates the root of a nonlinear equation f(x) =
!  The Bisection Method needs the following information
!   1) an interval [a,b] which brackets the root (so that f(a)* f(b)
!   2)  a function routine to evaluate f(x) at any point x
!   3)  a maximum number of iterations to do
!   4)  a tolerance to determine if method has converged
!
!  The code first checks to make sure that we have an interval brack
!  The algorithm determines the midpoint of the interval [a,b] and t
!  the root is in [ a, (a+b)/2) ] or [ (a+b)/2, b]; then it moves th
!  the interval bracketing the root half the previous size
!
!  Convergence is achieved if the error given by (b-a)/2^k is less t
```

```
!***********************************************************************
  implicit none

  integer, parameter :: max_iterations = 30
  real, parameter :: tolerance = 1.0e-4
  real :: a, b   !  interval bracketing root
  real :: b_minus_a  ! length of initial interval
  real :: f_at_a, f_at_b, f_at_midpoint  ! stored function values at
  real :: f   !  the function for evaluation f(x)
  real ::  midpoint   !  (a+b)/2
  integer :: k  ! iteration counter

!***********************************************************************
!
!  set beginning interval [a,b]
!
   print *, "Enter the beginning interval [a,b] separated by comma"
   read *, a, b
   b_minus_a = b - a     ! save length of initial interval which is u
!
!  test to make sure that we have an interval bracketing the root
!   that is,  f(a) and f(b) have opposite signs which implies f(a) *
!   if not stop and print error message
!
   f_at_a = f(a)   ! save function evaluations
   f_at_b = f(b)

   if ( f_at_a * f_at_b > 0.0 ) then

        print *, "error - we do not have an interval that brackets t
        stop

     else if (f_at_a * f_at_b == 0.0 ) then   ! we have the root at

 print *, " root found at endpoint of beginning interval "
  if ( f_at_a   == 0.0 ) print *, "root is at x= ", a
  if ( f_at_b == 0.0 ) print *, "root is at x= ", b
  stop

  end if

! loop over number of iterations with a maximum number of iterations
!
   do k = 1, max_iterations

     print *, "iteration number ", k

     midpoint = ( a + b ) / 2.0    ! calculate midpoint
     f_at_midpoint = f(midpoint)   ! evaluate function at midpoint

     print *, " approximate root is ", midpoint
     print *, " function value (residual) at approximate root is ",

!  now test to see if root is in (a,midpoint) or (midpoint, b) or if
```

```
     if ( f_at_midpoint * f_at_a  < 0.0 ) then    !  root is between

   b = midpoint   ! move right end of interval bracketing root
   f_at_b = f_at_midpoint  ! save function value

else if ( f_at_midpoint * f_at_b  <  0.0 ) then  ! root is between m

   a = midpoint   ! move left end of interval bracketing root
   f_at_a = f_at_midpoint  ! save function value

else      !  f(midpoint) is zero

  print *, "root found exactly at ", midpoint
  stop

     end if

!  check for convergence;  we know that at the kth step error is no
!  is the original interval.  If this value is < tolerance, stop; ot
```

```
     if (  b_minus_a / 2.0** k  < tolerance ) then   ! convergence a

 print *, "root is found to desired tolerance in", k, "  iterations
 print *, "root is located at x = ", midpoint
 stop

    end if

  end do   !  end do over iteration loop
!
!  if the code reaches this point without finding the root,
!  then the maximum number of iterations has been reached
!
  print *, " Bisection Method failed to find root in ", max_iteratio

!*****************************************************
!*****************************************************
   CONTAINS
```

```
!*****************************************************
!*****************************************************

  function eval_f ( x ) result ( f)

  real :: x
  real :: f

  f = x**2 - 5.0

  end function eval_f

!*****************************************************
   end program bisection_method
!*****************************************************
```

Now let's look at the results for $f(x) = x^2 - 5$ on $[2, 3]$

approximation	relative error $= \left\| \dfrac{\sqrt{5} - \text{approximation}}{\sqrt{5}} \right\|$
2.5	0.118034
2.25	0.62306 $\times 10^{-2}$
2.125	0.496711 $\times 10^{-1}$
2.1875	0.217203 $\times 10^{-1}$
2.21875	0.774483 $\times 10^{-2}$
2.23438	0.757123 $\times 10^{-3}$
2.24219	0.0273673 $\times 10^{-2}$

We note that the error oscillates; it is not monotonically decreasing. Why is this?

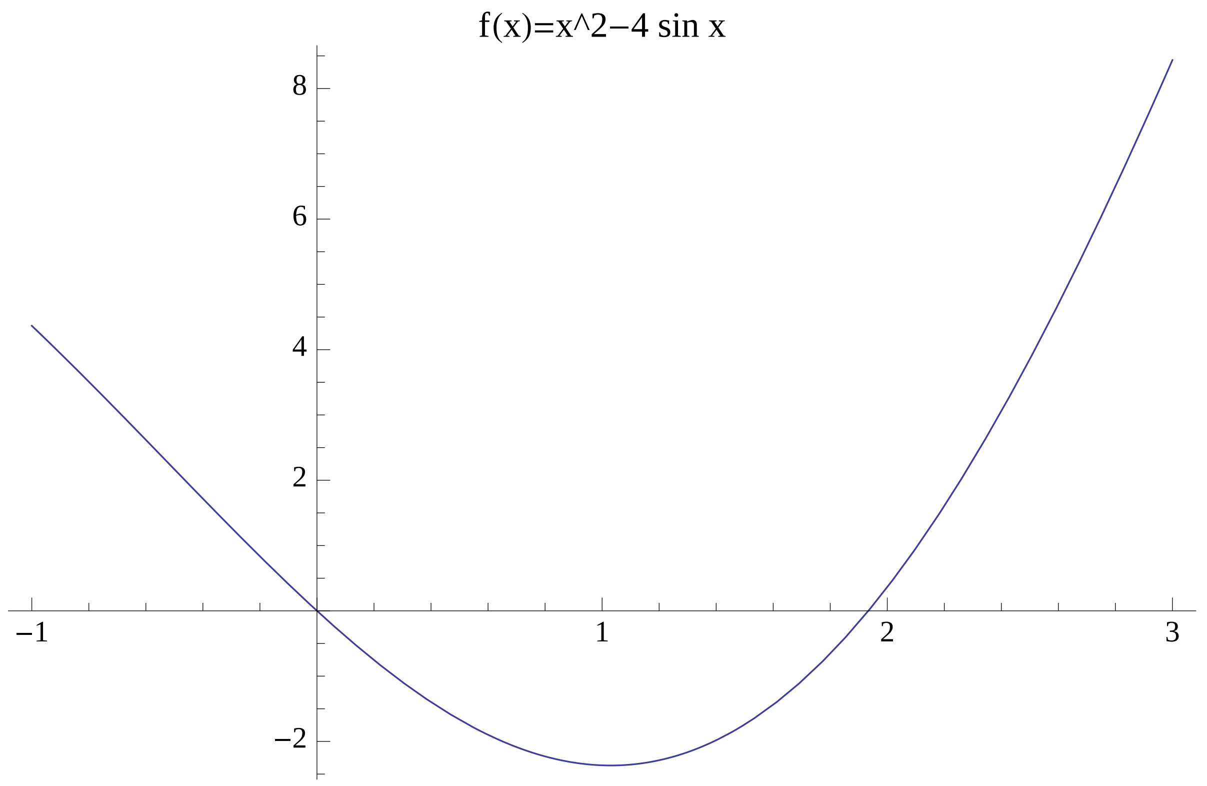
Classwork

1. Download the code `bisection.f90` and make the necessary modifications to run it for
$$f(x) = x^2 - 4\sin x$$
with a beginning interval of [1,3]. The function has one root in the interval at $x = 1.93375$. Set the tolerance to 10^{-3}. How many steps does it take to satisfy the tolerance?

2. The function $f(x) = f(x) = x^2 - 4\sin x$ has two roots in the interval $[-1, 3]$ at $x = 0$ and $x = 1.93375$. Using the graph below, which root do you think the bisection method will find? Try the code and see if you are right.

3. Instead of checking for convergence in the main program, write a subprogram to accomplish this. Do you need a function or a subroutine?

Out[448]=


f(x)=x^2−4 sin x