Using Monte Carlo to Estimate π using Buffon's Needle Problem

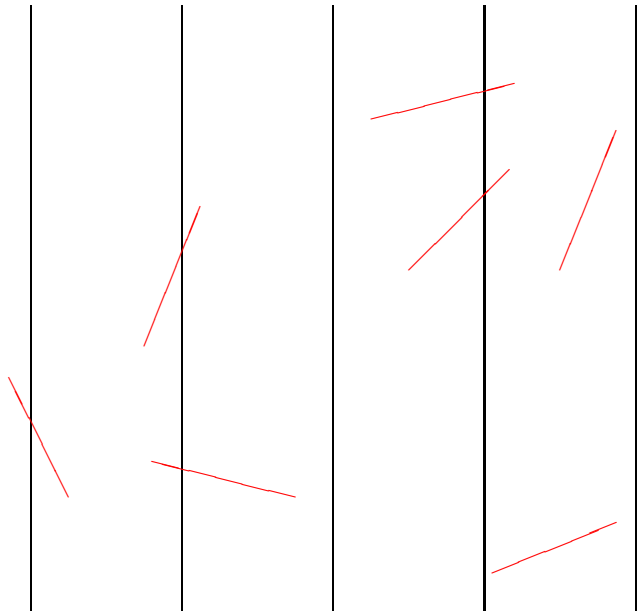
An interesting related problem is Buffon's Needle which was first proposed in the mid-1700's.

Here's the problem (in a simplified form).

- Suppose you have a table top which you have drawn lines every 1 inch.
- You now drop a needle of length 1 inch onto the table.
- What is the probability that the needle will be lying across (or touching) one of the lines?
- Actually, one can show that the probability is just $\frac{2}{\pi}$

So if we could simulate this on a computer, then we could have another method for approximating π. To see this we simply realize that if we drop n needles and n_{hits} of the dropped needles cross a line, then the ratio $\frac{n_{\text{hits}}}{n}$ is an approximation

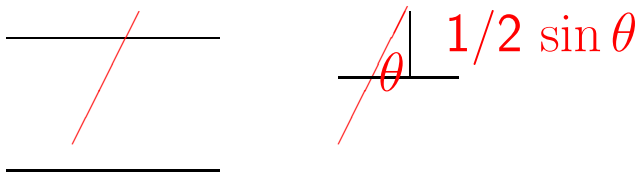
to the probability given by $\frac{2}{\pi}$.



With 7 needles, the approximation to $\frac{2}{\pi}$ is $\frac{5}{7}$ or $\pi = \frac{14}{5}$

Let's analyze the problem to see how we can implement it.

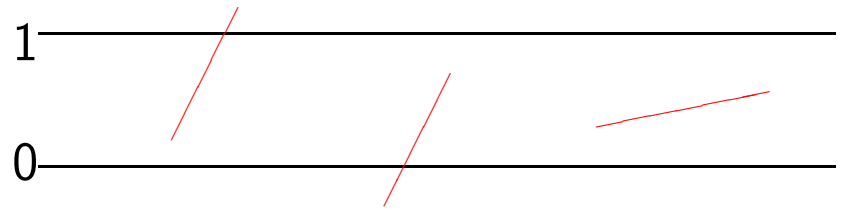
- Let's assume without loss of generality that the lines are horizontal, they are spaced 1 unit apart and the length of the needle is 1 unit.
- Moreover, it's enough to consider two lines, one at y=0 and the other at y=1
- Assume, as in the figure, that we have dropped a needle and that we know the location of the middle of the needle (actually we just need the y-coordinate) and the angle θ the needle makes with the horizon.
- So in the figure we see that the vertical side of the triangle has length $\frac{1}{2}\sin\theta$



- Since we know the y-coordinate of the middle of the needle we know the y coordinates of the end of the needle

$$y \pm \frac{1}{2}\sin\theta$$

- Here are the 3 situations we can have



- If the top of the needle has a y-coordinate greater than one, then the needle touches the line, i.e., we have a "hit".
- If the bottom of the needle has a y-coordinate less than zero, then it touches the other line and we have a "hit".
- If neither of these conditions are satisfied, then we don't have a hit.
- Since it is known that the probability that the needle will touch the line is $2/\pi$ then

$$\frac{\texttt{number of hits}}{\texttt{total number of needles}} \approx \frac{2}{\pi}$$

and thus

$$\pi \approx 2 \times \frac{\texttt{total number of needles}}{\texttt{number of hits}}$$

Let's look at the similarities of this problem and the other way to approximate π

- In the previous problem we generated two random numbers (x, y) between 0 and 1; here we generate two random numbers (y, θ) where $0 \leq y \leq 1$ and $0 \leq \theta \leq 180°$. Remember though that Fortran requires angles to be in radians so $0 \leq \theta \leq \pi$ radians.
- Before we checked to see if $x^2 + y^2 \leq 1$ to get a "hit'; here we have to see if the top of the needle hits the line $y = 1$ or the bottom of the needle hits the line $y = 0$. The y-coordinates of the ends of the needle are just the y-coordinate of the center of the needle plus or minus $.5 \sin\theta$.
- Here π is approximated by the formula

$$\pi \approx 2 \times \frac{\texttt{total number of needles}}{\texttt{number of hits}}$$

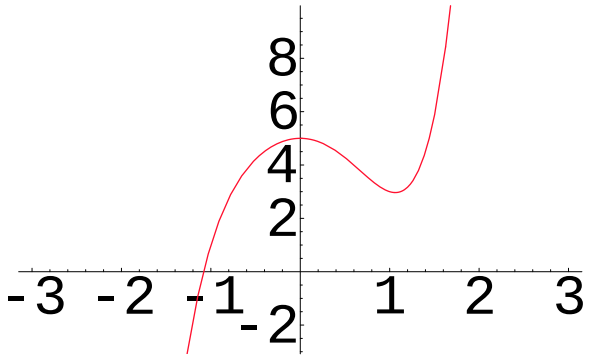
- This is your project for Part I (undergraduates).

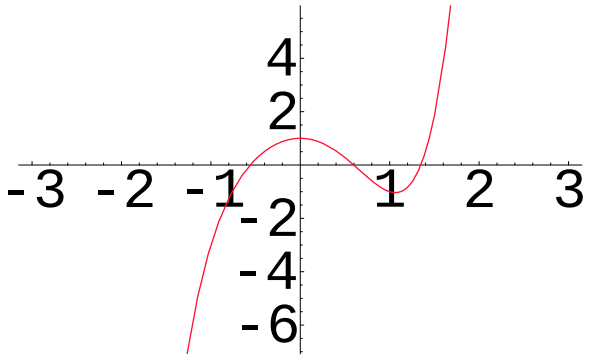
PART II - Solving a Nonlinear Equation

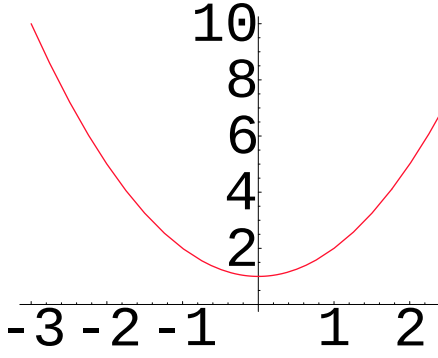
The next problem we want to investigate is finding the root of a single nonlinear equation $f(x) = 0$ (i.e., where the function crosses the y-axis) such as

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

- 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 could be a single value as in our case or a vector in other cases), say x^0, and then we will generate a sequence of approximations

$$x^0, x^1, x^2, x^3, \ldots$$

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

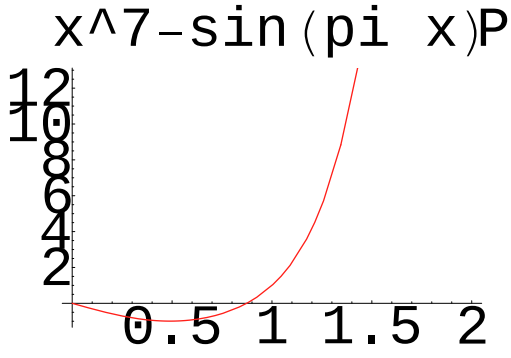
$$2, \quad 2.5556, \quad -5.3586, \quad -11.0851, \quad 41.01324, \quad 121.0039$$

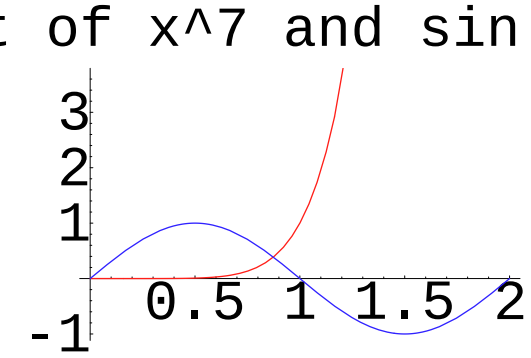
In this case our approximations are not tending to some value, so we say

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?
  - 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.





  - Physical intuition
  - 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?

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

$$2, \quad 1.9576, \quad 1.8386, \quad 1.7651, \quad 1.51324, \quad 1.3033 \quad 1.1911 \quad 1.1788 \quad 1.1172 \quad 1.0977$$

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".
- We can do this if we know something about the error we are making. We don't have this information for every method, but for the first method we will consider, we do. If we have an estimate for the error then we can check

$$\text{if error} < \text{prescribed tolerance} \qquad \text{then stop}$$

- If we don't have an estimate about the error, then what do we do?
  - Since $x^n \longrightarrow x^*$ as $n \longrightarrow \infty$ (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

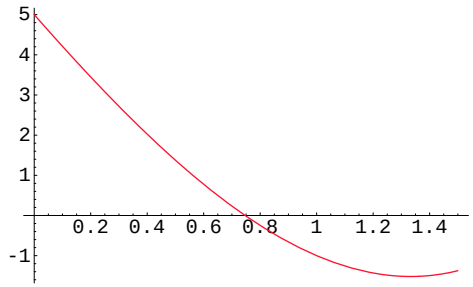
$$\frac{|x^{m+1} - x^m|}{|x^{m+1}|} \leq \text{ prescribed tolerance}$$

  - We will talk more about this later.

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 $f(1.4) < 0$.



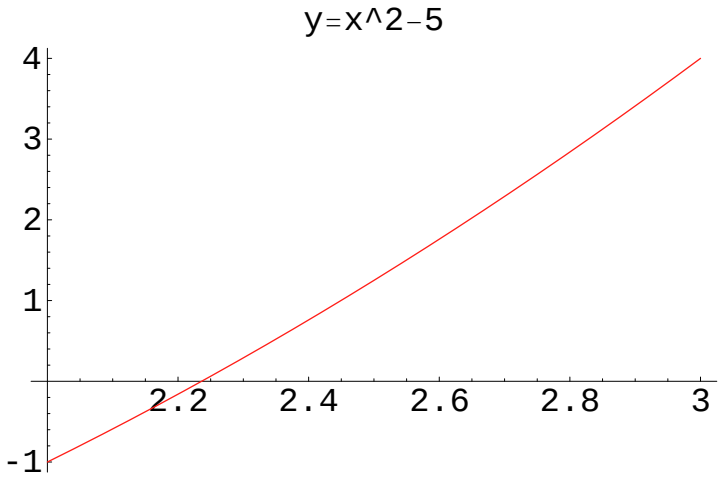
- So if we start with an interval $[a, b]$ where $f(a)f(b) < 0$ then we are guaranteed that a zero or root of $f(x)$ lies in (a, b).
- The basic idea behind the bisection method is that we take the midpoint of the interval $[a, b]$ and then check to see if the root is in $[a, \frac{a+b}{2}]$ or in $[\frac{a+b}{2}, b]$.
  - To do this we simply evaluate $f(\frac{a+b}{2})$ and compare the sign with that of $f(a)$ and $f(b)$.
  - 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

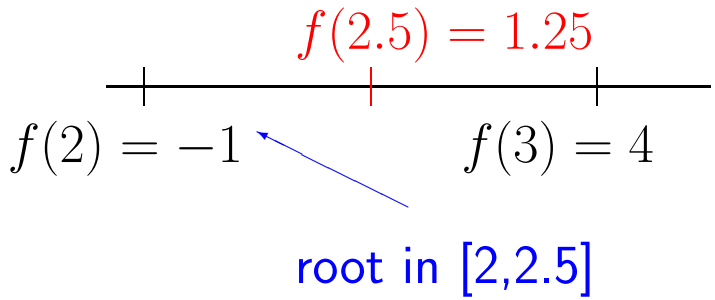
We can approximate the square root of 5 by finding a root of the equation

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

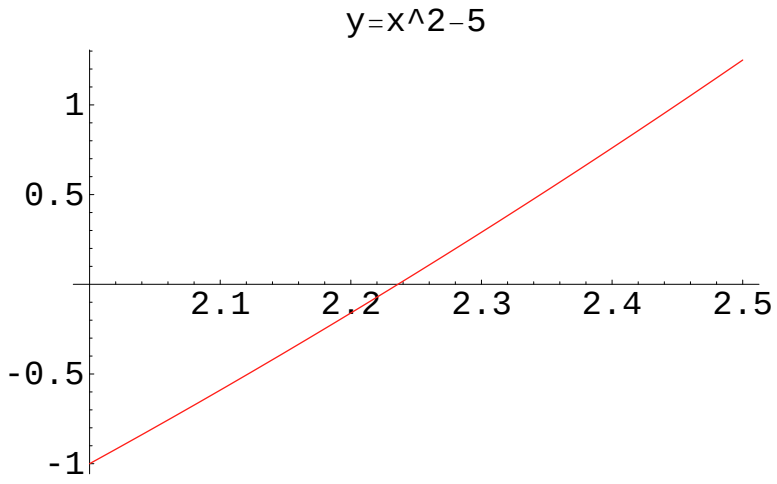
- We know that $2 < \sqrt{5} < 3$ and since $f(2) = -1 < 0$ and $f(3) = 4 > 0$, we can take the interval $[2, 3]$.

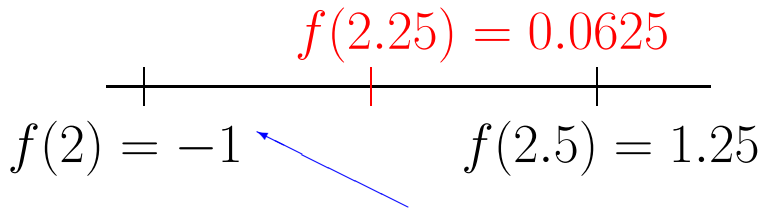


- Our first approximation would be the midpoint $x = 2.5$
- Since $f(2.5) = 1.25$ we see that $f(2) = -1$ and $f(2.5)$ are opposite signs (i.e., $f(2)f(2.5) < 0$) so we take [2,2.5] as our next interval.



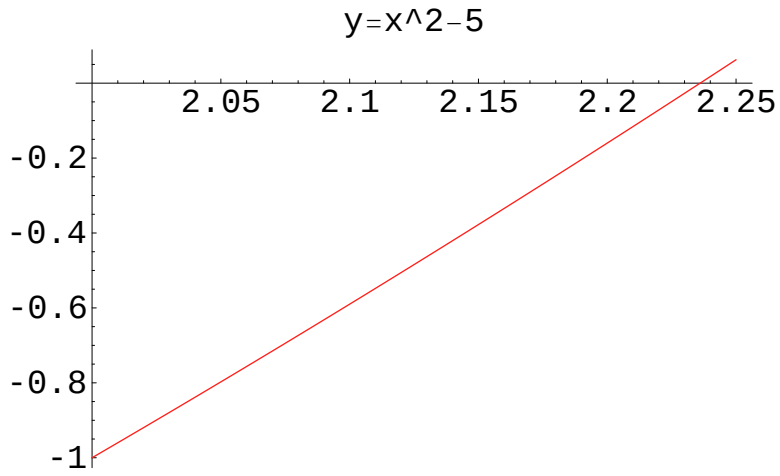
- Our next approximation is (2+2.5)/2= 2.25
- Since $f(2.25) > 0$ and $f(2) < 0$ we choose our next interval to be [2,2.25]

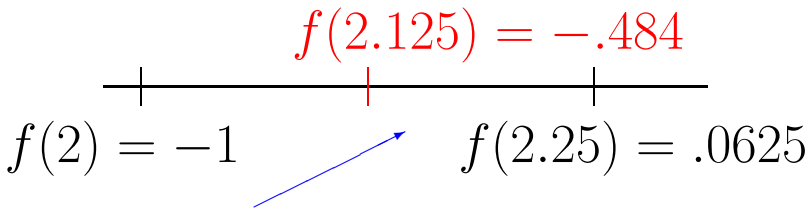




root in [2,2.25]

- 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

- So we will have a tolerance specified in our program and at each step of the bisection method we will check to see if our error is less than the tolerance.

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 $\frac{a+b}{2}$ the error is less than $\frac{b-a}{2}$
  - 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 $\frac{b-a}{4}$
  - At the end of the third step the interval is half the size of the previous step so the error is less than $\frac{b-a}{8}$

– 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
- However the main problem we face is that we have to evaluate a given function at specific points.
- We do not want to "hard wire" the function into the code. This means that in the body of the code you have the explicit function coded. This is what you did when you estimated the definite integral for a homework exercise.

In our example for finding the root of $x^2 - 5$, this 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
```

- If we want to find the root of a different function, then we have to recode all of these statements.
- Note that there is a lot of redundancy in these statements. In all three cases we are simply evaluating $x^2 - 5$ at different points.
- It would be convenient if we could program this like we would write it mathematically

```
f_at_a = f(a)

f_at_b = f(b)
```

where somehow we can tell the compiler what f is. This way we could simply define f one time and if we change f, we only have to change the code in

one place – the definition of f.

- This is achieved through the use of a function routine which is just a sub-program.
- We first look at the `IF-ELSE IF` constructs and then at functions in fortran.

IF-ELSE IF CONSTRUCTS

- 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 are terminating 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.

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

Suppose that the temperature, `temp`*, has 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 (temp > high_temperature)   then
    print *, "temperature is above normal high"
  else if ( temp < low_temperature )  then
    print *, "temperature is below normal low"
  else
    print *, "temperature is in normal range"
end if
```

Functions in Fortran

- Fortran allows the use of subprograms in the form of functions or subroutines.
- Subprograms are designed to perform particular tasks under the control of some other program unit.
- Using subprograms is effective programming because
  - it compartmentalizes our code
  - it avoids having the same section of code in multiple places
- Execution of these subprograms is controlled by some other program unit which can be the main (or driver) program or another subprogram.
- We have already encountered functions in the form of intrinsic functions such as the trig functions or the random number generator. These are routines that are written by someone else. We are interested in writing our own.
- Sometimes we can use either a function or a subroutine to perform some task

- Functions are designed to return a single value to the calling program.
- Subroutines are designed to return multiple outputs but they can only return a single output.
- Both functions and subroutines can have multiple inputs.

- A simple example of the use of a function is a mathematical function such as $f(x) = x^2 - 5$ where we have the value of x as the input and the value of f at x as the output.
- For $f(x, y, z) = x^2 - yz^4$ we would have x, y, z as input and a single value of f as the output.

Syntax for Functions

```
function name (  input variables  )
    declarations

    statements

end function name
```

- The input variables are given in a list separated by a comma and enclosed in parentheses; e.g., `(x, y, z)`
- The output variable is the name of the function which is returned to the calling program. It can be real, integer, etc.
- The function subprogram is an independent unit which can be compiled but not executed.

- The main program that calls the function does not have access to any of the local variables used in the function. It only gets the value that is returned once the function is called.
- The function routine only gets the input variables passed through the argument statement. It knows nothing else about the calling program.
- The number of arguments in the calling statement must equal the number of arguments in the function statement. In addition, they must be the same type; that is, you shouldn't pass a real variable into an variable declared as an integer in the function routine (integers and reals take different amounts of storage).
- The declarations are the same as in a main program. We add the `implicit none` statement and we declare each variable used as real, integer, etc.
- Since the output variable is returned in the name of the function routine, we must also declare this.
- Another format for the function statement exists where you are allowed to give the output variable a name other than the name of the function. This requires the use of `result` option in the function statement; we will address

this later.

- As an example, consider a function routine for $f(x) = x^2 - 5$.

```
function fx ( x )
implicit none
real ::  fx
real ::  x

    fx = x*x - 5.0

end function fx
```

Calling statements in your main program might look like

```
f_at_a = fx ( a )
```

```
f_at_b = fx ( b )
```

- If you have a statement like one of these in your program and you fail to have a function named `fx` then you will get an error message like undefined symbol: fx
- When you have this statement in your main program you are invoking the function `fx` and returning a value. Consequently, in the main calling routine you must also declare `fx`.

```
real ::  fx
```

- If you fail to do this, then you will get an error message like fx - this name does not have a type and must have an explicit type.

Alternate Syntax for Functions

```
function name ( input variables ) result ( output variable )
    declarations

    statements

end function name
```

- In this case you have to declare the output variable, not the function name.

An example of a function for converting inches to centimeters

A function routine is the natural choice for this since you have one output - the converted value in centimeters

```
function convert_to_centimeters (inches)

implicit none

real ::  inches

real ::  convert_to_centimeters

    convert_to_centimeters = 2.54 * inches


end function convert_to_centimeters
```

- The result is returned in the `convert_to_centimeters`.

- In the calling program we must declare `convert_to_centimeters`

```
real ::  convert_to_centimeters
```

- Examples of calling statements in the main program are

```
value = convert_to_centimeters(2.0) ! converts 2 inches to cms
a = 4.5
value = convert_to_centimeters(a) ! converts a inches to cms
```

Alternate syntax for function

```
function convert_to_centimeters (inches) result (answer)
implicit none
real ::  inches
real ::  answer
    answer = 2.54 * inches
end function convert_to_centimeters
```

The calling statements are the same:

```
  value = convert_to_centimeters(2.0)
```

You do not have to declare the name of the function.

Subroutines

- Another example of a subprogram is a subroutine.
- Remember that the difference in a function and subroutine is that the subroutine allows multiple outputs whereas a function returns only one value to its calling program.

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

    statements

end subroutine name
```

- Since a subroutine can have multiple outputs, we can not access in the same way we did a function (recall for a function we could invoke the subprogram by a statement like `area = compute_area_circle(2.)`)

- Subroutines must be invoked or called through the use of the `call` statement
- 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 must be the same in the calling argument and the subprogram definition (there is the capability to use an `optional` declaration for an argument but we won't consider this now).
- Again the types (integer, real, character, etc.) must match in the calling program and the subprogram.

- 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, readibility, 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.

Example - A Subroutine to Convert Inches to Centimeters & Meters

- In this example we want to input inches to the subroutine and output the value in centimeters and meters.
- Since we have 2 outputs (cm and m) we need to use a subroutine instead of a function.

```
subroutine convert_inches ( inches, cm, meters )

implicit none

real, intent(in) ::  inches

real, intent(out) ::  cm

real, intent(out) ::  meters

cm = 2.54 * inches
```

```
meters = 0.001 * cm

end subroutine convert_inches
```

Calling statement:

```
call convert_inches ( a, cm, meters )
```

where `a` has been assigned a number.

The three variables `a, cm, meters` must be declared as real in the calling program.

Where do you put subprograms?

- You can have subprograms in a separate file which can be compiled but not executed.
- Alternately you can add the subprograms to the end of the main program.
- For now, we will simply add them to the end of our main program but ultimately we will have many of our subprograms in separate files which we will call modules.
- You can add them AFTER the `end program` statement. If you do this then you have to have `implicit none` in each subprogram.
- You can add them just BEFORE the `end program` statement if you add the statement `CONTAINS`. In this case you do NOT have to have `implicit none` in each subprogram.