Chapter 3: Elements of the Program

3.1 Data Types and Operators

Variables are declared in Fortran by assigning them a particular *data type*. The most common data types and their interpretations are

`integer`	integer
`real`	real number
`logical`	boolean (has value `.true.` or `.false.`)
`character`	string
`complex`	complex number

In Fortran, you can do arithmetic with all the operators you expect from a basic calculator (+,−,∗,/, etc.), but there are several others intrinsic to the language. Some of the less obvious ones and their interpretations are

`m**n`	m^n
`mod(m,n)`	$m \mod n$
`sign(m,n)`	$m \times \frac{n}{\lvert n \rvert}$ for $n \neq 0$

The following example demonstrates how some operations featuring `integer` and `real` data types behave.

arithmetic.f95

```fortran
program arithmetic
implicit none
   integer :: m = 3, n ! declare two integers, assign value of 3 to m
   real :: x, y ! declare two real numbers

   n = 5.9 ! rounded down to 5
   x = 3 ! converted integer to real number
   y = n ! converted integer to real number

   ! no decimal in output
   write(*,*) 'int(3) = ', m
   ! rounded down to 5
   write(*,*) 'int(5) = ', n
   ! decimal in output
   write(*,*) 'real(3) = ', x
   ! converted integer to real
   write(*,*) 'real(5) = ', y
   ! integer division is rounded down
   write(*,*) 'int(5)/int(3) = ', n/m
   ! real division
   write(*,*) 'real(5)/real(3) = ', y/x
   ! converted to real
   write(*,*) 'int(5)/real(3) = ', n/x
   ! real(1.)*int(n) is computed first
   write(*,*) 'real(1)*int(5)/int(3) = ', 1.*n/m
   ! integer(n)/int(m) is computed first
   write(*,*) 'int(5)/int(3)*real(1) = ', n/m*1.
   ! the compiler treats n and m as reals
   write(*,*) 'real(5)/real(3) = ', real(n)/real(m)
```

```
    ! the compiler treats x and y as integers
    write(*,*) 'int(5.)/int(3.) = ', int(y)/int(x)
end program arithmetic
```

- All variables must be declared at the top of the source code before other procedures.
- A variable can be assigned a constant value during or after it is declared.

arithmetic - commands and output

```
gfortran arithmetic.f95 -o arithmetic
./arithmetic
int(3) =            3
int(5) =            5
real(3) =    3.00000000
real(5) =    5.00000000
int(5)/int(3) =            1
real(5)/real(3) =    1.66666663
int(5)/real(3) =    1.66666663
real(1)*int(5)/int(3) =    1.66666663
int(5)/int(3)*real(1) =    1.00000000
real(5)/real(3) =    1.66666663
int(5.)/int(3.) =            1
```

- The compiler treats numbers without a decimal point, such as `3`, as integers and treats numbers with a decimal point, such as `5.9`, as real numbers.
- When an integer is assigned a real value, the decimal is rounded down; for example the assignment `n=5.9` stores the value 5 in `n`.
- When a binary operation is performed between two variables of the same type, the result is assumed to be of that type. For example, the integer division `n/m` where values of 5 and 3 are stored in `n` and `m`, resp., results in an integer with value 1 since 5/3 rounds down to 1. Also, the real division `y/x` where values of 5. and 3. are stored in `y` and `x`, resp., results in a real with value 1.66666663 approximately equal to 5/3.
- When a binary operation is performed between an integer and a real, the result is assumed to be real. For example, the division `n/x` where `n` is an integer with value 5 and `x` is a real with value 3. results in a real with value 1.66666663 approximately equal to 5/3.
- You can instruct the compiler to treat integer `n` as a real with `real(n)` or you could intstuct it to treat real `x` as an integer with `int(x)`.

3.1.1 Single and Double Precision

Data types can optionally be declared with specifiers and attributes. An application of this is creating a flag that can be used to designate `real` as *double precision*. In short, the IEEE standard specifies two ways of representing real numbers, *single precision* and *double precision*. Each single precision real occupies 32 bits of memory and each double precision real occupies 64 bits of memory. A larger set of numbers are representable in double precision and they are used for higher accuracy.

In the following program, we demonstrate how to designate reals as double precision using data type specifications and attributes.

precision.f95

```fortran
program precision
implicit none
   ! store the default kind of a double precision real
   integer, parameter :: rp = kind(0.d0)
   ! declare single precision, parameter pi1
   real, parameter :: pi1 = 2.*acos(0.)
   ! declare double precision, parameter pi2
   real(rp), parameter :: pi2  = 2._rp*acos(0._rp)
   character(2) :: s

   s = "pi"

   print*, rp
   write(*,*) 'single precision zero = ',0.
   write(*,*) 'double precision zero = ',0.d0
   write(*,*) 'double precision zero = ',0._rp
   write(*,*) 'double precision zero = ',real(0.,rp)
   write(*,*) 'single precision ',s,'1 = ',pi1
   write(*,*) 'double precision ',s,'2 = ',pi2
   write(*,*) 's.p. acc., d.p. rep. ',s,' = ',2._rp*acos(0.)
end program precision
```

- A variable declared with the `parameter` attribute is a constant that must be immediately assigned a value and may not be reassigned another value after declaration.
- By default, a `real` is single precision, but appending the suffix `d0` to an unnamed real number, such as `0.d0`, designates it as double precision.
- Each data type has a corresponding integer that the `kind` function returns. By storing the `kind` of a double precision real in the parameter `rp`, we can later designate a `real` as double precision with the specifier `dp`, such as `real(rp) :: x` for declaring a variable, and `real(x,rp)` or `2._rp` for casting a `real`.
- The length of a character data type can be declared with `character(len=LENGTH)`.

precision - commands and output

```
gfortran precision.f95 -o precision
./precision
           8
single precision zero =    0.00000000
double precision zero =    0.0000000000000000
double precision zero =    0.0000000000000000
double precision zero =    0.0000000000000000
single precision pi1 =    3.14159274
double precision pi2 =    3.1415926535897931
s.p. acc., d.p. rep. pi =    3.1415927410125732
```

- The real `0.` is stored as zero with 8 digits in the decimal, whereas `0.d0` is stored as zero with 16 digits in the decimal.
- The double precision `pi2` was assigned a value based on the compuatation $\pi = 2 \arccos 0$ where both 2 and 0 were represented in double precision. It is more accurate than the single precision `pi1`.
- Computing $\pi = 2 \arccos 0$ where 2 is represented in double precision but 0 is represented in single precision results in a real with a double precision representation, but only single precision accuracy. Avoid mixing single precision operations with double precision variables as it may result in inaccuracies.

3.2 Control Sequences

Fortran has keywords that allow you to specify the procedural flow of the program. In this section, we outline these common control sequences.

- `if/then/else` - execute certain pieces of code based on logical conditions. The main logical operators and their interpretations are

<	or	`.lt.`	less than
<=	or	`.le.`	less than or equal to
==	or	`.eq.`	equal to
/=	or	`.ne.`	not equal to
>=	or	`.ge.`	greater than or equal to
>	or	`.gt.`	greater than

`.and.`	logical and
`.or.`	logical or
`.not.`	logical not

The following program discovers whether an integer `n` is positive, negative, or zero.

ifelse.f95

```fortran
program ifelse
implicit none
   integer :: n = 0
   if (n>0) then
      write(*,*) 'n is positive'
   else if (n<0) then
      write(*,*) 'n is negative'
   else
      write(*,*) 'n is zero'
   end if
end program ifelse
```

- `else` and `else if` statements are not strictly required. You could have a control sequence of the form `if (LOGICAL_CONDITION) ... end if`.

ifelse - commands and output

```
gfortran ifelse.f95 -o ifelse
./ifelse
n is zero
```

- `do` loops - execute a block of code repeatedly for a range of values of a variable. At least the upper and lower bounds but also the increment size can be specified in a `do` loop with `do i=LOWER_BOUND,UPPER_BOUND` or `do i=LOWER_BOUND,UPPER_BOUND,INCREMENT`, resp.
- `cycle` - increments to the next iteration in a `do` loop.

The following program demonstrates `do` loops with some recursive arithmetic.

do.f95

```fortran
program doex
implicit none
   integer :: i, n, factorial = 1
   real :: j, x

   ! add 1+2+3+4+5+6+7+8+9+10
   n = 0
   do i=1,10 ! from 1 to 10 increment by 1
```

```
      n=n+i
   end do
   write(*,*) '1+2+3+4+5+6+7+8+9+10 = 10*11/2 ? ',n==10*11/2

   ! compute 10 factorial
   do i=10,1,-1 ! from 10 to 1 increment by -1
      factorial=i*factorial
   end do
   write(*,*) '10 factorial = ',factorial

   ! add 1 through 10, excluding multiples of 3
   n = 0
   do i=1,10 ! from 1 to 10 increment by 1
      if (mod(i,3)==0) then
         cycle
      end if
      n=n+i
   end do
   write(*,*) '1+2+4+5+7+8+10 = 10*11/2-(3+6+9) ? ',n==10*11/2-(3+6+9)
end program doex
```

do - commands and output

```
gfortran do.f95 -o do
./do
1+2+3+4+5+6+7+8+9+10 = 10*11/2 ?  T
10 factorial =      3628800
1+2+4+5+7+8+10 = 10*11/2-(3+6+9) ?  T
```

- `do while` loops - execute a block of code while a logical condition is true.
- `exit` - exits a `do` or `do while` loop.

The following program discovers the nearest floating point number greater than 1 on your computer, and demonstrates how to exit from an infinite loop.

dowhile.f95

```
program dowhile
implicit none
   integer :: n
   real :: x, r = .5

   x=r ! initialize x = .1 (binary)
   n=0
   do while(1.+x>1.) ! while 1.000...0001 is greater than 1.
      x=x*r ! shift decimal bit rightward
      n=n+1
   end do
   print*, 'Nearest floating point number greater than 1: '
   print*, 1.+r**n, nearest(1.,1.)

   n=0
   do while(.true.) ! infinite loop
      n=n+1
      if (n>10) then
         exit ! exit from while loop
      end if
   end do
   write(*,*) 'n = ',n
end program dowhile
```

- The intrinsic function `nearest` returns the nearest floating point number to a given number in a given direction, for example `nearest(1.,1.)` returns the nearest floating point number greater than 1. The first `1.` indicates to look for the floating point number closest to 1. and the second `1.` because it is positive indicates to look in the positive (right) direction.

dowhile - commands and output

```
gfortran dowhile.f95 -o dowhile
./dowhile
Nearest floating point number greater than 1:
  1.00000012       1.00000012
n =           11
```

3.3 Input/Output

In this section, we introduce several methods Fortran offers for inputting and outputting data.

3.3.1 Input/Ouput to the Screen

- Input - You can ask the user to provide data from the terminal command line with the `read(*,*)` statement.
- Output - You can output data to the terminal screen with the `write(*,*)` or `print*,` statements.

In the `read` statement above, the first asterisk tells the compiler to read from the default source, the terminal command line, and the first asterisk in the `write` statement above tells the compiler to write to the default destination, the terminal screen. In both statements the second asterisk tells the compiler to use the default “list-directed” or free formatting.

The following program outputs whether or not a positive integer entered by the user is prime.

readwritescreen.f95

```
program readwritescreen
implicit none
   integer :: i = 2, n
   logical :: n_is_prime = .true.

   write(*,*) 'Enter a positive integer'
   read(*,*) n ! read integer, throws error if not integer
   if (n>0) then
      ! determine whether n is prime
      if (n==1) then
         n_is_prime = .false.
      else if (n==2) then
         n_is_prime = .true.
      else
         do while (i<=n/2)
            if (mod(n,i)==0) then
               n_is_prime = .false.
               exit
            end if
            i=i+1
         end do
      end if
      ! write the result
      print*, n,' is prime ? ',n_is_prime
   else
      ! write if input is not positive
```

```
      print*, n,' is not positive.'
   end if
end program readwritescreen
```

readwritescreen - commands and output

```
gfortran readwritescreen.f95 -o readwritescreen
./readwritescreen
Enter a positive integer
    1300021  is prime ?  T
```

3.3.2 Input/Ouput to a File

To input or output data from a file, you first must open the file with the `open` command. This command is passed an integer that corresponds to the file called a unit number. Some file unit numbers are reserved for the system and you should avoid passing them to `open`. With the `gfortran` compiler,

- **standard error (stderr)** is 0 - used to output error messages to the screen.
- **standard in (stdin)** is 5 - used to input data from the terminal command line, as with `read(*,*)`.
- **standard out (stdout)** is 6 - used to output data to the screen, as with `write(*,*)`.

You can pass optional specifier arguments to `open` such as `file='FILENAME'`, `action='read'`, or `action='write'`.

The following program opens two files, reads from one of them and writes to the other.

readwritefile.f95

```
program readwritefile
implicit none
   integer :: i
   integer, parameter :: rp = kind(0.d0)
   real(rp) :: x

   open(10,file='readfile.dat',action='read')
   open(11,file='writefile.dat',action='write')
   do i=1,5 ! i know that 'readfile.dat' has 5 lines
      ! read from file
      read(10,*) x
      ! write to file
      write(11,*) gamma(x) ! the gamma function
   end do
   close(11) ! remember to close each opened file
   close(10)
end program readwritefile
```

readwritefile - commands and output

```
gfortran readwritefile.f95 -o readwritefile
./readwritefile
```

readfile.dat

```
2.
2.5
3.
3.5
4.
```

writefile.dat

```
    1.0000000000000000
    1.3293403881791370
    2.0000000000000000
    3.3233509704478426
    6.0000000000000000
```

3.3.3 Formatted Input/Output

Sometimes the default format is not sufficient for your task, and a specific format has to be chosen. In this section, we introduce how to specify formatting. All of the examples we consider are for outputting data, but the similar rules apply for inputting data.

The second argument in the `write(*,*)` command is for specifying output format. Format is specified by a list (of type `character`) of descriptors for what each field of the output should look like. The common descriptors and their interpretations are listed below. Each of `W`, `D`, and `E` should be thought of as placeholders that should be replaced by positive integers that specify the width, number of decimal digits, and number of exponent digits, respectively.

`aW`	character
`iW`	integer
`fW.D`	floating point (decimal)
`esW.DeE`	scientific notation
`Wx`	space

For example, the format `(a5,i10,f15.5,1x,es30.15e3)` specifies the format to output a character with width 5, an integer with width 10, a decimal number with width 15 and 5 decimal digits, one space, and a decimal number in scientific form with width 30, 15 decimal digits, and 3 exponent digits. A repetitive portion of a format can be multiplied to shorten the list of format descriptors. For example, `(3(i5,f15.5))` is equivalent to `(i5,f15.5,i5,f15.5,i5,f15.5)`. You can also have the width default to trucate leading and trailing zeros for numbers or leading a trailing spaces for strings.

The following example demonstrates some basic examples of formatted output.

formatio.f95

```fortran
program formatio
implicit none
   integer, parameter :: rp = kind(0.d0)
   real(rp), parameter :: pi = 2._rp*acos(0._rp)
   character(len=100) :: frmt
   integer :: n = 1
   real(rp) :: x, y, z

   frmt = '(a,i5,i5,i5)'
   write(*,frmt) 'Integer: width 5 : ',n,n+4,n+9
   frmt = '("Same as above: ",3i5)'
   write(*,frmt) n,n+4,n+9

   x = 111.111_rp
   y = 222.222_rp
   z = 333.333_rp
   write(*,'(a)') '1 space, Floating point: width 7, dec. 3 : '
   frmt = '(3(1x,f7.3))'
   write(*,frmt) x,y,z

   x = x*pi
   y = y*pi
   z = z*pi
```

```
   write(*,*)'Scientific: width 30, dec. 15, exp. 3, 2 per line : '
   frmt = '(2es30.15e3)'
   write(*,frmt) pi,x,y,z

   frmt = '(a12, 1x, es20.15)'
   write(*,frmt) 'width too small',pi

   frmt = '(a, 1x, i0, 1x, f0.16)'
   write(*,frmt) 'default width',n+100,pi
end program formatio
```

formatio - commands and output

```
 gfortran formatio.f95 -o formatio
 ./formatio
Integer: width 5 :     1    5   10
Same as above:     1    5   10
1 space, Floating point: width 7, dec. 3 :
 111.111 222.222 333.333
 Scientific: width 30, dec. 15, exp. 3, 2 per line :
          3.141592653589793E+000         3.490655013330155E+002
          6.981310026660310E+002         1.047196503999047E+003
width too sm ********************
default width 101 3.1415926535897931
```

- If the specified output format width of a string is too small the string is truncated, and if the specified output format width of a number is too small the number is replaced by asterisks.

3.4 Example: Monte Carlo Experiment

We apply what we've learned so far to approximate π through a Monte Carlo experiment. This example also introduces how to choose a pseudorandom number from a uniform distribution on 0 to 1. The experiment is based on the fact that the unit circle is contained in the square with coordinates $(\pm 1, \pm 1)$ and the ratio of their areas (circle to square) is $\frac{\pi}{4}$. Therefore, the probability that m out of n points randomly chosen from the square $(\pm 1, \pm 1)$ lie in the unit cirlcle $x^2 + y^2 = 1$ is $\frac{m}{n} \approx \frac{\pi}{4}$. And by symmetry, we could consider only the portion of the picture in, say, the first quadrant.

The following program chooses random points from the square with vertices $\{(0, 0), (0, 1), (1, 0), (1, 1)\}$ and approximates π until the magnitude of the error in the approximation is small enough.

montepi.f95

```
program montepi
implicit none
   integer, parameter :: rp = kind(0.d0)
   real(rp), parameter :: pi = 2._rp*asin(1._rp)
   real(rp) :: x, y, s = 0., tol = 1.e-5
   integer :: m = 0, n = 0

   ! initialize pseudorandom number generator
   call srand(0)

   do while(abs(pi-s)>tol) ! while error > tol
      x = rand() ! random number between 0 and 1
      y = rand()
      if (x**2._rp+y**2._rp<1._rp) then ! if in unit circle
         m=m+1 ! increase count of points in unit circle
      end if
```

```
      n=n+1 ! increase count of points in square (+-1,+-1)
      s=4._rp*real(m)/real(n) ! approximate pi
   end do
   write(*,*) m, n, s
end program montepi
```

- The pseudorandom number generator is “seeded” with `srand(0)`. This initializes the generator so that every time you call `rand()` a new number is generated. As long as the generator is seeded with the same integer, every time you run the program, calling `rand()` repeatedly will generate the same sequence of numbers. If you need `rand()` to generate a different sequence of numbers each time you run the program, you could seed the generator with the current time.
- By default `rand()` chooses a random number from a uniform distribution on 0 to 1.

montepi - commands and output

```
gfortran montepi.f95 -o montepi
./montepi
         355          452   3.1415929203539825
```

- The program halts after choosing 452 points when $|\pi - s| < 10^{-5}$ where s is the approximation. The chosen points and a plot of $|\pi - s|$ vs. n where n is the number of chosen points are depicted in Figure 1.

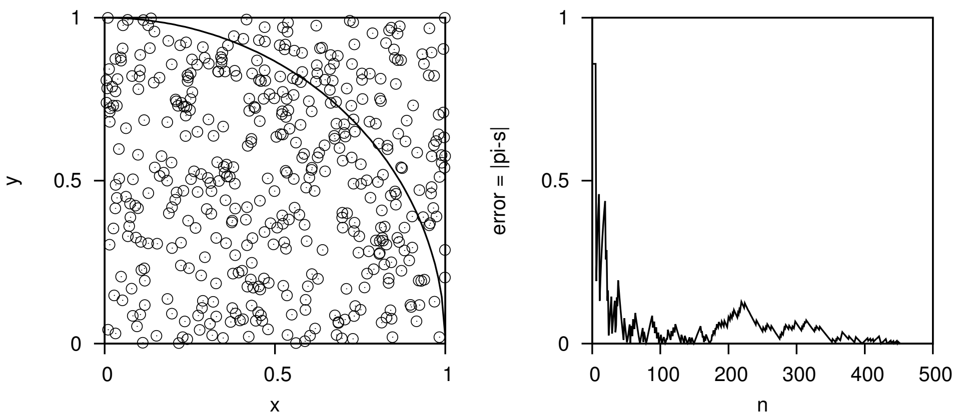

Figure 1: Random samples from the square with vertices $\{(0,0),(0,1),(1,0),(01,1)\}$ (left) and the π approximation error as a function of number of points chosen n (right).

3.5 Example: Rootfinding

Now, we write a program that solves the equation

$$f(x) = 0.$$

This example also introduces the Fortran `function` construct that can be used to code mathematical functions. There are several iterative methods that approximate the solution to this equation useful when an analytic approach is intractable, such as the bisection method, the secant method, and Newton’s method (see Quateroni, *Numerical Mathematics*, Ch. 6). In each method, an initial guess(es) x_0 $(, x_1, x_2, \ldots)$ at a root of f is iteratively refined according to some rule

$$x_n = \phi(x_0, \ldots, x_{n-1}) \qquad n \geq 1$$

until $|x_n - x_{n-1}|$ is less than a chosen tolerance or n exceeds a chosen maximum number of steps.

In the following example, we implement the secant method to solve the equation $\ln x = e^{-x}$ or, equivalently,

$$f(x) := \ln x - e^{-x} = 0.$$

In the secant method, x_n is chosen to be the root of the secant line of f through points at x_{n-1} and x_{n-2}, hence ϕ depends only on x_{n-2} and x_{n-1} for $n \geq 2$ and two initial guesses x_0 and x_1 are required. We summarize the method in the following algorithm.

Algorithm: The secant method

Data: $x_0, x_1, f, tol, maxstep$

Result: A root of f or a non-convergence message

$n \leftarrow 1$

while $|x_n - x_{n-1}| > tol$ *and* $n < maxstep$ **do**

  $m \leftarrow [f(x_n) - f(x_{n-1})]/(x_n - x_{n-1})$

  $x_{n-1} \leftarrow x_n$

  $x_n \leftarrow x_n - f(x_n)/m$

  $n \leftarrow n + 1$

end

if $|x_n - x_{n-1}| \leq tol$ **then**

  return x_n

else

  return non-convergence message

end

secant.f95

```fortran
program secant
implicit none
   integer, parameter :: rp = kind(0.d0)
   real(rp) :: x0, x1, y0, y1, m, tol
   integer :: step, maxstep

   tol = 1.e-5_rp ! set tolerance
   maxstep = 1e5 ! set max # steps

   x0 = 1._rp ! initialize x
   x1 = 2._rp
   y0 = f(x0) ! initialize f(x)
   y1 = f(x1)
   step = 0 ! initialize step counter
   do while (abs(x1-x0)>tol.and.step<maxstep)
      m = (y1-y0)/(x1-x0) ! compute slope
      x0 = x1 ! store x
      x1 = x1 - y1/m ! iterate x
      y0 = y1 ! store f(x)
      y1 = f(x1) ! update f(x)
      step = step + 1 ! increment step counter
      write(*,*) step, x1, y1
   end do
   if (abs(x1-x0)<=tol) then
      write(*,*) 'f(x) = 0 for x = ',x1
   else
      write(*,*) 'Method did not converge before ',maxstep,' steps.'
   end if
contains
   function f(x)
      integer, parameter :: rp = kind(0.d0)
      real(rp), intent(in) :: x
```

```
      real(rp) :: f
      f = log(x)-1._rp/exp(x)
   end function
end program secant
```

secant - commands and output

```
gfortran secant.f95 -o secant
./secant
           1   1.3974104821696125       8.7384509621480227E-002
           2   1.2854761201506528      -2.5389724827401428E-002
           3   1.3106767580825409       9.0609778401362639E-004
           4   1.3098083980193003       9.1060669357712065E-006
           5   1.3097995826147546      -3.2957613305129030E-009
f(x) = 0 for x =     1.3097995826147546
```

Exercises

1. Write a program that computes the sum of the series

$$\sum_{n=0}^{\infty} \frac{1}{n!}.$$

(Hint: For large enough n, $\frac{1}{n!}$ is stored as zero in your computer.) Recall that the sum of this series is e. Report the error $e-s$ where s is your approximation of the sum.

2. Write a program that solves $f(x) = x - \sin x - 1 = 0$ using Newton's method in double precision with initial guess $x_0 = 3$ and tolerance 10^{-5}. Newton's method chooses x_n to be the root of the line tangent to f at x_{n-1} for $n \geq 1$. We summarize the method in the following algorithm.

Algorithm: Newton's method

Data: $x_0, f, tol, maxstep$
Result: A root of f or a non-convergence message
$x_1 \leftarrow x_0 - f(x_0)/f'(x_0)$
$n \leftarrow 1$
while $|x_n - x_{n-1}| > tol$ *and* $n < maxstep$ **do**
  $m \leftarrow f'(x_n)$
  $x_{n-1} \leftarrow x_n$
  $x_n \leftarrow x_n - f(x_n)/m$
  $n \leftarrow n+1$
end
if $|x_n - x_{n-1}| \leq tol$ **then**
  return x_n
else
  return non-convergence message
end

Report a table of the form

n	x_n	$\|x_n - x_{n-1}\|$	$f(x_n)$
0	3	–	1.8588799919401329
⋮	⋮	⋮	⋮