Fortran: FORmula TRANslation

Lecture Notes

Joe McKenna
Department of Mathematics - The Florida State University
www.math.fsu.edu/~jmckenna

Contents

Chapter 1: Introduction

Welcome to "Everything I wanted to know about Fortran, but was afraid to ask." This seminar operates as an introductory crash course in Fortran for applied mathematicians. Although Fortran dates back to the 1950s and many newer langugaes have proved more useful for a broader range of projects, Fortran is still widely used by the scientific computing community. The advantages of Fortran are especially prevalent in vector operations.

The name Fortran comes from FORmula TRANslation. That is, the language was originally developed for easy implementation of mathematical formulas, vector and matrix operations in particular. The flexibility of Fortran to natively handle arrays makes your life much easier when coding basic routines, like matrix vector products, to more advanced routines like linear solvers or conjugate gradient.

This seminar will not include all aspects of Fortran, but will be enough to get you up and running, solving problems, and coding with organization. The seminar is structured to learn Fortran through example, with commentary along the way. All materials for the course are available at `http://www.math.fsu.edu/~jmckenna/fortran`.

Software

For Linux users, all the software that you need likely comes prepackaged for your distribution, and you can install as you would normally with a package manager. For Windows users, "MinGW" and "Cygwin" are systems that provide open source software and functionality similar to Linux distributions on Windows machines. For Mac users, "XCode" in addition to the open source software package manager "Homebrew" provide functionality similar to Linux. Alternatively, on either Windows or Mac, you can emulate a Linux operating system with a virtual machine (`https://www.virtualbox.org/`).

Required Software

In order to create programs, you will need a Fortran *compiler*. The job of the compiler is to translate Fortran *source code*, i.e. the part that you write, to machine code that can be executed on your computer. There are a few choices for Fortran compilers, but for the purposes of this course we will want to use the one freely available from the GNU software movement, `gfortran`. This compiler comes prepackaged for Linux systems and is easily installed. The only other software that you will need to start coding is a text editor. There are many text editors. The one you choose to use is up to you. Several programmers prefer "Emacs" or "Vim" since they are free, extensible and have a lot of features geared toward program development.

Optional Software

Before coding scientific computing programs in low-level procedural languages such as Fortran or C/C++, it is useful to test your proposed algorithms in a higer-level interactive environment. These allow you to define variables and perform computations with them without having to recompile everything from source any time you want to alter your code. There are a few choices for these.

- MATLAB - A proprietary high-level language interactive environment available from MathWorks. The license is expensive, but the program is available in the math grad basement.
- Octave - A free alternative to MATLAB provided by the GNU software movement.

After generating data from your code, you need a convenient way to present it. This will include making plots and tables and collecting these into a typed report. For this, I recommend

- `gnuplot` - A free plotting program from the GNU free software movement. With this program, you can write scripts that generate plots. It is well documented on the web. It comes prepackaged for a lot of Linux distributions and is available for Windows and Mac.
- LaTeX - The *de facto* standard for typesetting scientific documents. It comes prepackaged for a lot of Linux distributions (as the package `texlive`). It is also availble for Windows and Mac. For Windows or Mac users, I suggest installing "MikTeX" or "MacTeX", respectively, to get started with LaTeX.

Fortran Standards

Since Fortran was invented in the 1950s, it has gone through a number of revisions (i.e. FORTRAN 66, 77 and Fortran 90, 95, 2003, 2008) that modified the syntax and changed features of the language. With each revision, official standards that precisely specify the syntax and behavior are released by the Fortran standards committee (`http://www.j3-fortran.org/`). Unfortunately, these are not freely available. If you desire, you can purchase the official Fortran standard through the International Organization of Standardization: ISO/IEC 1539-1:2010.

There are a number of other resources, although less official, that you can use for free to find just about everything you'll need to know about Fortran. These are

- The `gfortran` compiler documentation - The `gfortran` compiler fully supports Fortran 95 and partially supports Fortran 2003 and 2008. The documentation specifies exactly which functions and routines intrinsic to Fortran are supported. This can be found online.
- `fortran90.org` - An unofficial collection of Fortran tips and information.

Chapter 2: Program Structure

2.1 Hello World

Let's look at the basic structure of a Fortran program by writing a customary "Hello World" program.

hello.f95

```
program hello
implicit none
   write(*,*) 'Hello world'
   ! Equivalently,
   ! print*, 'Hello world'
   ! write(6,*) 'Hello world'
end program hello
```

- The *source code* for the program is delimited by the `program PROGRAM_NAME ... end program PROGRAM_NAME` tags.
- Fortran is a compiled language in which variables are *explicitly* declared. For example, an integer `i` is declared by including `integer :: i` at the top of the source code. If after declaring `i` as an integer, you assign it a value of 1 with `i=1`, your computer knows not to waste disk space on storing the decimal component of `i` since it is zero. Your computer also has certain rules for dealing with undeclared, that is *implicitly* declared, variables. Using implicitly declared variables makes it more difficult to read or debug code so we avoid using them by including the statement `implicit none` immediately after the program declaration. This statement instructs your computer to throw an error when an undeclared variable is encountered.
- The `write` command is used to output data to a particular destination and in a particular format. The asterisks ("*" is an asterisk) in `write(*,*)` tell your computer to output to the default destination (the terminal screen) with the default formatting ("list-directed" or free format). The first asterisk is for destination and the second is for formatting. The default destination is assigned the file unit "6" so `write(6,*)` has the same effect as `write(*,*)`. Additionally, `print*,` has the same effect as `write(*,*)`.
- An exclamation point ("!" is an exclamation point) is used to comment. The compiler will ignore anything to the right on the same line as an exclamation point.
- The file extension `.f95` indicates that the source code is written in the 1995 version of Fortran. The language has gone through a number of revisions since it first appeared, but the most recent version that is fully supported by `gfortran` is Fortran 95.
- Fortran 95 is **not case-sensitive**. For example, the keywords `program`, `PROGRAM`, and `PrOgRaM` all have exactly the same effect. Furthermore, if you try to declare two variables with names `i` and `I`, an error will be thrown indicating that a duplicate variable was declared.

Invoking the `gfortran` *compiler* translates Fortran source code into executable *machine code*, a *binary*, that can be called to run.

hello - commands and output

```
gfortran hello.f95 -o hello
./hello
Hello world
```

- Calling `gfortran` with the option `-o hello` instructs the compiler to ouput to the file `hello`. If this option is excluded, the compiler by default outputs to `a.out`.
- While in the same directory, we can execute the binary with `./hello`, which prints "Hello World" to the terminal screen.

2.2 Automating Your Report

In numerical mathematics courses, you will be expected to write programs and submit reports that explain how your program works and the results of any tests you ran with it. You will often need to create tables of numerical data, graphical plots, and code listings. It is useful to automate as much of this process as possible so that incremental updates can easily be incorporated.

In this section, we present a technique for creating a automated report. In particular, we create a single program from Fortran that does some computations, creates tables, creates figures, and collects everything into a document. Most of the Fortran syntax may be new to you now but we will look at it more closely in subsequent chapters. Furthermore, the program will call a `gnuplot` script to plot data and call LaTeX to compile a report, which requires installations of `gnuplot` and LaTeX callable from the command line and knowledge of `gnuplot` and LaTeX syntax.

Let's examine the following source code for a program that creates an automated report.

automate/automate.f95

```
1  program automate
2  implicit none
3     integer :: i
4     real :: x(0:10) ! an array indexed from 0 to 10
5     ! compute pi and store as a constant
6     real, parameter :: pi = 2.*acos(0.)
7
8     ! populate array of x-values between 0 and 2 pi
9     x=(/(i/5.*pi,i=0,10)/)
10
11    ! write sine and cosine data to file 'figure.dat'
12    open(10,file='figure.dat',action='write',status='replace')
13    do i=0,10
14       write(10,*) x(i), sin(x(i)), cos(x(i))
15    enddo
16    close(10)
17    ! call gnuplot script 'automate.plt' that plots data
18    call execute_command_line('gnuplot automate.plt', wait=.true.)
19
20    ! write LaTeX table to file 'table.tex'
21    open(10,file='table.tex',action='write',status='replace')
22    write(10,*) '\begin{tabular}{|c|c|c|} \hline'
23    write(10,*) '$x$ & $\sin x$ & $\cos x$ \\ \hline'
24    do i=0,10
25       write(10,*) x(i), '&', sin(x(i)), '&', cos(x(i)), '\\'
26    enddo
27    write(10,*)'\hline \end{tabular}'
28    close(10)
29    ! call pdflatex on 'automate.tex' to compile report to pdf
30    call execute_command_line('pdflatex automate.tex', wait=.true.)
31 end program automate
```

- Lines 3-9 declare variables (an integer `i`, an array of real numbers `x`, and a real parameter `pi`) and populate `x` with values $x = i\frac{\pi}{5}$ for $i = 0, 1, 2, \ldots, 10$.

- Lines 11-16 open a file with unit 10 to be overwritten called `figure.dat` and output data in three columns: x, $\sin x$, and $\cos x$, for $x = i\frac{\pi}{5}$ for $i = 0, 1, 2, \ldots, 10$, then close the file. Similarly, lines 21-28 open a file with unit 10 to be overwritten called `table.tex` and output the same data but in LaTeX table syntax.
- Line 18 calls `gnuplot` to execute the script `automate.plt`. The script was written separately and requires knowledge of the `gnuplot` syntax. The script produces the plot `figure.eps`.
- Line 30 calls `pdflatex` to compile the report source file `automate.tex`. It was written separately and requires knowledge of the LaTeX syntax. The table is included in `automate.tex` with the line `\input{table.tex}`. The plot is included in `automate.tex` using the LaTeX `graphicx` package.

The source files `automate.f95`, `automate.plt`, and `automate.tex` located in `f95/automate` can be used as a starting point for creating your own automated report.

Exercise

1. Install `gfortran`. Write and execute a "Hello World" program in Fortran.

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

```
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)
```

```
30    ! the compiler treats x and y as integers
31    write(*,*) 'int(5.)/int(3.) = ', int(y)/int(x)
32 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

```
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

```
1  program ifelse
2  implicit none
3     integer :: n = 0
4     if (n>0) then
5        write(*,*) 'n is positive'
6     else if (n<0) then
7        write(*,*) 'n is negative'
8     else
9        write(*,*) 'n is zero'
10    end if
11 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

```
1  program doex
2  implicit none
3     integer :: i, n, factorial = 1
4     real :: j, x
5
6     ! add 1+2+3+4+5+6+7+8+9+10
7     n = 0
8     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

```
1  program readwritescreen
2  implicit none
3     integer :: i = 2, n
4     logical :: n_is_prime = .true.
5
6     write(*,*) 'Enter a positive integer'
7     read(*,*) n ! read integer, throws error if not integer
8     if (n>0) then
9        ! determine whether n is prime
10       if (n==1) then
11          n_is_prime = .false.
12       else if (n==2) then
13          n_is_prime = .true.
14       else
15          do while (i<=n/2)
16             if (mod(n,i)==0) then
17                n_is_prime = .false.
18                exit
19             end if
20             i=i+1
21          end do
22       end if
23       ! write the result
24       print*, n,' is prime ? ',n_is_prime
25    else
26       ! 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

```
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.

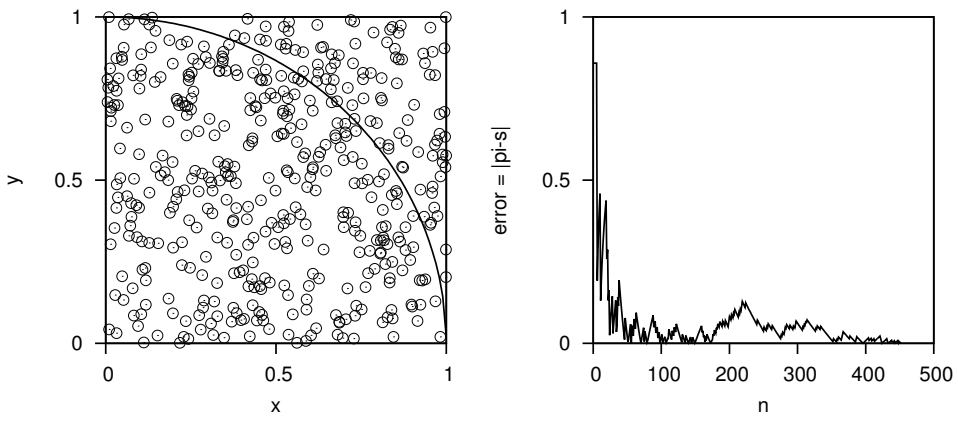

Figure 1: Random samples from the square with vertices $\{(0,0),(0,1),(1,0),(01,1)\}$ (left) and the π approximation error as a function of number of points chosen n (right).

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

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

Chapter 4: Program Organization

Now that we know some basics of coding in Fortran, we learn how to make code easier to read, test, and reuse by organizing programs into manageable parts. Some basic constructs useful for breaking up the workflow of a program are

- `function` - takes in multiple arguments and returns a single argument.
- `subroutine` - takes in and returns multiple arguments.
- `module` - contains variable declarations, functions, and subroutines that can be used by a program.

In the `function` and `subroutine` constructs, the argument variables may declared with special attributes that tell the compiler what values they should have at the beginning and end of a call. There are three options:

- `intent(in)` - used with functions and subroutines; the value of the argument may not be changed by the function/subroutine
- `intent(out)` - used with subroutines; the value of the argument is undefined on entry to the procedure and must be assigned a value before exit
- `intent(inout)` - used with subroutines; the value of the argument is defined on entry to the procedure and can be assigned a new value before exit

4.1 Functions

A function in Fortran is a procedure that accepts multiple arguments and returns a single result. In addition to allowing users to declare their own functions, called *external* functions, the language already includes some *intrinsic* functions.

4.1.1 Intrinsic Functions

A list of intrinsic procedures supported by the `gfortran` compiler can be found in the compiler documentation. Some common intrinsic functions and their interpretations are

`abs(x)`	$\|x\|$
`exp(x)`	e^x
`log(x)`	$\ln x$
`log10(x)`	$\log_{10} x$
`sin(x)`	$\sin x$ where x is in radians
`asin(x)`	$\arcsin x$
`floor(x)`	greatest integer less than or equal to x
`ceiling(x)`	least integer greater than or equal to x

4.1.2 External Functions

These are procedures written by the user that can be called by a program. They can be written in the same file as the program source code outside the `program ... end program` tags or in a separate

file. In the following example we write a program that calls an external function in a separate file that evaluates

$$f(x) = x^2 - x - 1.$$

The file with the function is as follows.

function/function.f95

```
function f(x)
implicit none
   integer, parameter :: rp = kind(0.d0)
   real(rp), intent(in) :: x
   real(rp) :: f

   f=x**2._rp-x-1._rp
end function f
```

- The function code is delimited by the `function FUNCTION_NAME(ARGUMENTS) ...` `end function FUNCTION_NAME` tags. Inside these tags, a variable with the same name of the function is declared. The value of this variable is what is returned by the function and it **must** be assigned a value.
- The function argument `x` is declared with the `intent(in)` attribute. If you try to assign a value to `x` inside the function, an error will be thrown. You do not have to declare function variables with `intent(in)`, but doing so ensures that any changes made to them do not affect their value in the main program.

The file with the program is as follows.

function/functionex.f95

```
program functionex
implicit none
   integer, parameter :: rp = kind(0.d0)
   real(rp):: x, y
   real(rp), external :: f

   x = (1._rp+sqrt(5._rp))/2._rp ! the golden ratio is a root of f
   write(*,*) 'x = ',x,'(before calling f)'
   y = f(x)
   write(*,*) 'x = ',x,'(after calling f)'
   write(*,*) 'f(x) = ',y
end program functionex
```

- The function `f` is defined with the `external` attribute to instruct the compiler that the function is declared outside the `program ... end program` tags.

Since the code is in separate files, we call `gfortran` to compile both.

functionex - commands and output

```
gfortran function.f95 functionex.f95 -o functionex
./functionex
x =    1.6180339887498949      (before calling f)
x =    1.6180339887498949      (after calling f)
f(x) =    0.0000000000000000
```

4.2 Subroutines

Subroutines are more general than functions as they allow multiple input arguments and output results. However, you do have to be diligent when you assign the `intent` of each argument. Just like functions, subroutines can be written in the same file as the program source code outside the `program ... end program` tags or in a separate file. In the following example, we write a program that calls a subroutine in a separate file. The subroutine calculates some statistics with three integers that it is passed and orders the integers.

The file with the subroutine is as follows.

subroutine/subroutine.f95

```
subroutine stats(n,i1,i2,i3,minimum,maximum,median,mean)
implicit none
   integer, intent(in) :: n
   integer, intent(inout) :: i1,i2,i3
   integer, intent(out) :: minimum,maximum,median
   real, intent(out) :: mean

   ! compute min and max
   minimum = min(i1,i2,i3) ! min: intrinsic function
   maximum = max(i1,i2,i3) ! max: intrinsic function
   ! compute median
   if (i1==minimum.or.i1==maximum) then
      if (i2==minimum.or.i2==maximum) then
         median = i3
      else
         median = i2
      end if
   else
      median = i1
   end if
   ! compute mean
   mean = (real(i1)+real(i2)+real(i3))/real(n)
   ! order i1, i2, i3
   i1=minimum
   i2=median
   i3=maximum
end subroutine stats
```

- The subroutine code is delimited by the `subroutine SUBROUTINE_NAME ...` `end subroutine SUBROUTINE_NAME(ARGUMENTS)` tags.
- Of the arguments, `n` is declared `intent(in)`, `i1`,`i2`, and `i3` are declared `intent(inout)`, and `minimum`, `maximum`, `median`, and `mean` are declared with `intent(out)`. The value of `n` cannot be reassigned in the subroutine. The values of `i1`,`i2`, and `i3` are assigned before being passed to the subroutine and they can be reassigned in the subroutine. `minimum`, `maximum`, `median`, and `mean` should not be assigned values before being passed to the subroutine, and they should be assigned values by the subroutine.

The file with the program is as follows.

subroutine/subroutineex.f95

```
program subroutineex
implicit none
   integer :: n = 3
   integer :: i1, i2, i3
   integer :: minimum, maximum, median
   real :: mean
```

```
7
8     i1=8
9     i2=1
10    i3=3
11    ! call subroutine
12    call stats(n,i1,i2,i3,minimum,maximum,median,mean)
13    ! after calling subroutine
14    write(*,*) 'i1,i2,i3 = ',i1,i2,i3,'(after calling stats)'
15    write(*,*) 'minimum = ',minimum
16    write(*,*) 'maximum = ',maximum
17    write(*,*) 'median = ',median
18    write(*,*) 'mean = ',mean
19 end program subroutineex
```

- A subroutine is called with `call SUBROUTINE_NAME(ARGUMENTS)`.

Since the code is in separate files, we call `gfortran` to compile both.

subroutineex.f95 - commands and output

```
gfortran subroutine.f95 subroutineex.f95 -o subroutineex
./subroutineex
i1,i2,i3 =            8           1           3 (before calling stats)
i1,i2,i3 =            1           3           8 (after calling stats)
minimum =           1
maximum =           8
median =           3
mean =    4.00000000
```

4.3 Modules

Modules are constructs where variables (or functions or subroutines) can be defined once but used by multiple programs, functions, or subroutines. That is, they can be thought of as a construct used to "factor out" common code. For example, in order to work in double precision, we have repeatedly added the declaration

```
integer, parameter :: dp = kind(0.d0)
```

to the beginning of several programs, functions, and subroutines. Rather than rewriting this declaration in multiple locations we could write it once in a module like

```
1 module constants
2 implicit none
3    integer, parameter :: rp = kind(0.d0)
4 end module constants
```

then use the module named `constants` wherever we need `rp`. To import the module to a program use the `use` command like

```
1 program PROGRAM_NAME
2 use constants
3 implicit none
4    real(rp) :: x
5    ...
6 end program PROGRAM_NAME
```

- The `use constants` command is included before `implicit none` in programs, functions, or subroutines.

4.4 Makefiles

When organizing code into multiple files, it can be cumbersome to call `gfortran` on all of them. The UNIX utilitiy `make` can be used to script complicated compilation jobs. A script that instructs the compiler what to do is a *Makefile*. Rather than describing generally how to write a Makefile, we provide an example in the next section.

4.5 Example: Rootfinding Revisited

We apply what we've learned about program organization to reorganize the program in Chapter 3 that approximated the root of the equation $f(x) = \ln x - e^{-x}$ using the secant method. In addition, we add code for computing the root with Newton's method. We divide the source code among four different files `module.f95`, `function.f95`, `subroutine.f95`, and `rootfind.f95`.

In `module.f95`, we write a module that provides the parameter for flagging reals as double precision.

rootfind/module.f95

```
1 module constants
2 implicit none
3    integer, parameter :: rp = kind(0.d0)
4 end module constants
```

In `function.f95`, we code $f(x)$ in double precision. Since Newton's method requires f', we also code it. This file depends on `module.f95`.

rootfind/function.f95

```
1 function f(x)
2 use constants
3 implicit none
4    real(rp), intent(in) :: x
5    real(rp) :: f
6    f=log(x)-1._rp/exp(x)
7 end function f
8
9 function fp(x)
10 use constants
11 implicit none
12    real(rp), intent(in) :: x
13    real(rp) :: fp
14    fp=1._rp/x+1._rp/exp(x)
15 end function fp
```

In `subroutine.f95`, we implement the secant and Newton's method iterations in double precision. This file depends on `module.f95` and `function.f95`.

rootfind/subroutine.f95

```
1 subroutine secant(x0,x1,f,tol,maxstep)
2 use constants
3 implicit none
4    ! subroutine arguments
5    real(rp), intent(inout) :: x0,x1
6    real(rp), external :: f
7    real(rp), intent(in) :: tol
8    integer, intent(in) :: maxstep
9    ! local variables, no intent
10   integer :: step
11   real(rp) :: m
12
13   step = 0
```

```
   do while (abs(x1-x0)>tol.and.step<maxstep)
      m = (f(x1)-f(x0))/(x1-x0)
      x0 = x1
      x1 = x1 - f(x1)/m
      step=step+1
   end do
   if (step>=maxstep) then
      print*, 'Max steps taken by secant method!'
   end if
end subroutine secant

subroutine newton(x0,x1,f,fp,tol,maxstep)
use constants
implicit none
   ! subroutine arguments
   real(rp), intent(inout) :: x0, x1
   real(rp), external :: f, fp
   real(rp), intent(in) :: tol
   integer, intent(in) :: maxstep
   ! local variables, no intent
   integer :: step
   real(rp) :: m

   step = 0
   do while (abs(x1-x0)>tol.and.step<maxstep)
      m = fp(x1)
      x0 = x1
      x1 = x1 - f(x1)/m
      step=step+1
   end do
   if (step>=maxstep) then
      print*, "Max steps taken by Newton's method!"
   end if
end subroutine newton
```

In `rootind.f95`, we write the main program that calls the secant and Newton's method iteration subroutines. This file depends on `module.f95`, `function.f95`, and `subroutine.f95`.

rootfind/rootfind.f95

```
program rootfind
use constants
implicit none
   real(rp) :: x0, x1, tol=1.e-5_rp
   integer :: step, maxstep = 1e5
   real(rp), external :: f, fp

   ! The secant method
   x0 = 1._rp
   x1 = 2._rp
   call secant(x0,x1,f,tol,maxstep)
   write(*,*) 'By the secant method, x = ',x1

   ! Newton's method
   x0 = 1._rp
   x1 = 2._rp
   call newton(x0,x1,f,fp,tol,maxstep)
   write(*,*) "By Newton's method, x = ",x1
end program rootfind
```

Finally, we write a makefile `rootfind.mak` that is used by the `make` utility to compile the program. First, we list the name of the compiler and options to pass the compiler, then the name of the binary to be

created, then the filenames that have source code for any dependencies and the main program.

rootfind/rootfind.mak

```
COMPILER = gfortran
FLAGS = -O3 # this is an optimizer that makes loops more efficient
DEP = module.f95 function.f95 subroutine.f95
PROG = rootfind.f95
BIN = rootfind

$(BIN) : rootfind.f95
   $(COMPILER) $(FLAGS) $(DEP) $(PROG) -o $(BIN)
```

- The list of source code files in `DEP` must be such that no file depends on a file that is listed after it.
- A named makefile is called with the `make` utility by

```
make -f MAKEFILE_NAME
```

rootfind - commands and output

```
make -f rootfind.mak
./rootfind
By the secant method, x =    1.3097995826147546
By Newton's method, x =    1.3097995858041505
```

4.6 Example: Quadrature

In the next example, we write a program that approximates a definite integral $\int_a^b f(x)\,dx$. Approximating definite integrals using numerical techniques is referred to as (quadrature). We organize our code similarly to the example in the last chapter. Specifically, we divide the source code among three different files `module.f95`, `function.f95`, and `quadrature.f95`. In `function`, we will include both the mathematical function f that we want to integrate as well as a few approximation rules, the Left Endpoint Rule and the Midpoint Rule. These quadrature rules just calculate Riemann sums to approximate $\int_a^b f(x)\,dx$ on the type of grid indicated in the names of the rules. In `quadrature.f95`, we will call the functions and print their results.

In `module.f95`, we write a module that provides the parameter for flagging reals as double precision.

quadrature/module.f95

```
module constants
implicit none
   integer, parameter :: rp = kind(0.d0)
   real(rp), parameter :: pi = 2._rp*asin(1._rp)
end module constants
```

The file with the functions is as follows. This file depends on `module.f95`.

quadrature/function.f95

```
! f(x)=e^(-x^2)
function f(x)
use constants
implicit none
   real(rp), intent(in) :: x
   real(rp) :: f
   f = exp(-x**2._rp)
end function f
```

```
9
10 ! Left Endpoint Quadrature Rule
11 function left_endpoint(n,a,b,f)
12 use constants
13 implicit none
14    ! number of grid points
15    integer, intent(in) :: n
16    ! left and right endpoints of interval
17    real(rp), intent(in) :: a,b
18    ! integrand
19    real(rp), external :: f
20    ! return value
21    real(rp) :: left_endpoint
22    ! local variables
23    integer :: i
24    real(rp) :: x, dx
25
26    dx = (b-a)/n
27    left_endpoint = 0._rp
28    do i=0,n-1
29       x=a+i*dx
30       left_endpoint=left_endpoint+f(x)*dx
31    end do
32 end function left_endpoint
33
34 ! Midpoint Quadrature Rule
35 function midpoint(n,a,b,f)
36 use constants
37 implicit none
38    ! number of grid points
39    integer, intent(in) :: n
40    ! left and right endpoints of interval
41    real(rp), intent(in) :: a,b
42    ! integrand
43    real(rp), external :: f
44    ! return value
45    real(rp) :: midpoint
46    ! local variables
47    integer :: i
48    real(rp) :: x, dx
49
50    dx = (b-a)/n
51    midpoint = 0._rp
52    do i=0,n-1
53       x=a+(2._rp*i+1._rp)*dx/2._rp
54       midpoint=midpoint+f(x)*dx
55    end do
56 end function midpoint
```

The main program is as follows. This file depends on `module.f95` and `function.f95`.

quadrature/quadrature.f95

```
1 program quadrature
2 use constants
3 implicit none
4    ! number of grid points
5    integer :: n
6    ! left and right endpoints of interval
7    real(rp) :: a,b
8    ! integrand function and quadrature rule functions
9    real(rp), external :: f, left_endpoint, midpoint
```

```
10
11    a = 0._rp
12    b = 1._rp
13    n = 10
14
15    write(*,*) 'By the Left Endpoint Rule:',left_endpoint(n,a,b,f)
16    write(*,*) 'By the Midpoint Rule:',midpoint(n,a,b,f)
17    write(*,*) "Analytic value in terms of 'error funciton':",.5*pi**.5*erf
      (1.)
18 end program quadrature
```

Finally, we write a makefile `quadrature.mak` that compiles the program.

quadrature/quadrature.mak

```
1 COMPILER = gfortran
2 FLAGS = -O3 # this is an optimizer that makes loops more efficient
3 DEP = module.f95 function.f95
4 PROG = quadrature.f95
5 BIN = quadrature
6
7 $(BIN) : quadrature.f95
8    $(COMPILER) $(FLAGS) $(DEP) $(PROG) -o $(BIN)
```

quadrature - commands and output

```
make -f quadrature.mak
./quadrature
By the Left Endpoint Rule:  0.77781682407317720
By the Midpoint Rule:  0.74713087774799747
Analytic value in terms of 'error funciton':  0.74682412083799810
```

Exercise

1. Write a program that approximates the integral $\int_0^1 e^{-x^2}\,dx$ using Simpson's Rule. Simpson's Rule is given by

$$\int_a^b f(x)\,dx \approx \frac{\Delta x}{3}[f(x_0)+4f(x_1)+2f(x_2)+4f(x_3)+\cdots+2f(x_{n-2})+4f(x_{n-1})+f(x_n)]$$

where n is even, $\Delta x = (b-a)/n$, and $x_i = a+i\Delta x$, $i = 0,\ldots,n$. It is derived by using quadratic functions that interpolate f at x_{i-1}, x_i, and x_{i+1} for $i = 1, 3, 5, \ldots, n-1$ to approximate the area under f (see Quateroni, "Numerical Mathematics", 9.2).

Report a table of the form

n	I_n	$\int_0^1 e^{-x^2}\,dx - I_n$
2	?	?
4	?	?
6	?	?
⋮	⋮	⋮
20	?	?

for $n = 2, 4, 6, \ldots, 20$ where I_n is the approximation and $\int_0^1 e^{-x^2}\,dx - I_n$ is the error.

Chapter 5: Arrays

Fortran handles arrays easily compared to other low-level languages. For example it is very flexible when it comes to indexing or accessing elements of arrays. Therefore, it is an ideal choice for coding vector and matrix operations. By default, the initial index of an array is 1, but this can be easily changed.

It is important to note that ***Fortran stores array data by column***, often referred to as ***column-major.*** By default, when writing an array without formatting the leftmost column from top to bottom is written, then the next leftmost from top to bottom, and so on.

5.1 Basics

Some common functions that operate on arrays and their interpretations are

`size(A)`	number of elements in A
`transpose(A)`	the transpose of A
`maxval(A)`	maximum value in A
`minval(A)`	minimum value in A
`matmul(A,B)`	matrix product $A \times B$
`dot_product(a,b)`	the dot product $a \cdot b$
`sum(A)`	sum on elements in A
`product(A)`	product of elements in A

The following program introduces some basics of arrays.

array.f95

```
1  program array
2  implicit none
3     integer :: i, j, n
4     integer, dimension(5) :: A
5     integer :: B(5)
6     real, dimension(-3:1) :: C
7     real :: D(-2:2)
8     integer :: E(2, 2), F(-1:1,2), eye(3,3)
9
10    character(1024) :: frmt
11
12    A = (/ 1, 2, 3, 4, 5 /) ! explicit assignment
13    write(*,'(a,5(i0,1x))') 'A = ',A ! write as space-delimited row
14
15    B=(/(2*i,i=1,size(B))/) ! implicit do loop in explicit constructor
16    write(*,'(a)') '(as a column) B ='
17    write(*,'(i0)') B ! write as column
18
19    do i=1,size(B)
20       B(i)=2*i
21    end do
22    write(*,'(a)') '(as a row) B ='
23    write(*,'(5(i0,x))') B ! write as row
24
25    write(*,'(a,i0)'), 'A dot B = ',dot_product(A,B) ! dot product
26
27    C = (/ 1., 3., 5., 4., 2. /)
```

```
28    write(*,'(a,3(f0.0,1x))') 'C(-2:0) = ',C(-2:0) ! middle 3 elements as
      space-delimited row
29
30    forall(i=-2:2) D(i)=real(i)**2. ! forall declaration, more concise than do
       loop
31    write(*,'(a,5(f0.0,1x))') 'D(:) = ', D(:)
32    write(*,'(a,f0.0)'),'maxval(D) = ',maxval(D)
33    write(*,'(a,f0.0)'),'minval(D) = ',minval(D)
34    write(*,'(a,i0)'),'lbound(D) = ',lbound(D)
35    write(*,'(a,i0)'),'ubound(D) = ',ubound(D)
36
37    E = reshape((/1,2,3,4/),(/2,2/))
38    write(*,*) '(unformatted) E = ',E
39    write(*,*) '(formatted) E = '
40    do i=1,2
41       write(*,'(2(i0,1x),a,i0,a)') E(i,:),' (row ',i,')'
42    end do
43    write(*,*) '(formatted) E = '
44    do i=1,2
45       write(*,'(2(i0,1x),a,i0,a)') E(:,i),' (col ',i,')'
46    end do
47
48    F = reshape((/1,2,3,4,5,6/),(/3,2/))
49    write(*,'(a)') 'F = '
50    do i=-1,1
51       write(*,'(2(i0,1x))') F(i,:)
52    end do
53    write(*,'(a)') 'F = '
54    write(*,'(2(i0,1x))') transpose(F)
55
56    n=3
57    forall(i=1:n,j=1:n) eye(i,j)=(i/j)*(j/i) ! trick for creating identity
      matrix
58    write(*,'(a)') 'eye = '
59    write(frmt,'(a,i0,a)') '(',n,'(i0,1x))' ! write to frmt string
60    write(*,frmt) eye ! write eye with frmt string
61 end program array
```

- An array may be declared either with the `dimension` attribute following the data type declaration or by appending the array index range(s) to the variable name. For example, `integer, dimension(5) :: A` or `integer :: A(5)` declares an an array of 5 integers; the first integer is in `A(1)` down to the last in `A(5)`. You can specify an index range other than the default. For example, either `integer, dimemsion(-2:2) :: A` or `integer :: A(-2:2)` declare arrays of integers with the first element in `A(-2)` and the last element in `A(2)`. In general, you can declare an array with arbitrary data type, dimension and indexing with `DATATYPE :: ARRAYNAME(MIN1:MAX1,MIN2:MAX2,...,MINN:MAXN)`.
- There are a number of ways of assigning values to an array. To assign values explicitly, use the array constructor `(/ ... /)`; for example, if `A` is an array with size 5, use `A = (/ 1, 2, 3, 4, 5 /)`. The `reshape` command is useful for explicitly assigning values to a multi-dimensional array. Array assignments can also be made one element at a time; for example `B(i)=2*i` assigns a value of `2*i` to the i^{th} element in `B`. This should be used in conjunction with `do` loops. As a concise alternative, `forall` statements can be used to assign values one element at a time; for example, if `eye` is a 3×3 matrix `eye=forall(i=1:3,j=1:3) eye(i,j)=(i/j)*(j/i)` creates the identity matrix. This last example is a bit tricky since it relies on the fact that integer division is rounded down, i.e. the only time that `(i/j)*(j/i)` computes to `1` is if `i=j`, otherwise it computes to `0`.
- By default, `write(*,*)A` will write the elements of `A` in column-major order; that is, if `A` is an $n \times n$ matrix indexed from 1 to n in both dimensions `write(*,*)A` prints the list `A(1,1), A(2,1),...,`

`A(n,1)`, `A(1,2)`, `A(2,2)`,..., `A(n,2)`,..., `A(1,n)`, `A(2,n)`,..., `A(n,n)`. For better output, `do` loops or formatting should be used.

- Blocks of arrays can be accessed directly by specifying the desired indices. For example, If `A` is a 3 × 3 array indexed from 1 to *n* in both dimensions, the 2 × 2 minor matrix in the upper left of `A` is `A(1:2,1:2)` or the last column of `A` is `A(:,3)`.

array - commands and output

```
 gfortran array.f95 -o array
 ./array
A = 1 2 3 4 5
(as a column) B =
2
4
6
8
10
(as a row) B =
2 4 6 8 10
A dot B = 110
C(-2:0) = 3. 5. 4.
D(:) = 4. 1. 0. 1. 4.
maxval(D) = 4.
minval(D) = 0.
lbound(D) = -2
ubound(D) = 2
 (unformatted) E =           1           2           3           4
 (formatted) E =
1 3  (row 1)
2 4  (row 2)
 (formatted) E =
1 2  (col 1)
3 4  (col 2)
F =
1 4
2 5
3 6
F =
1 4
2 5
3 6
eye =
1 0 0
0 1 0
0 0 1
```

Sometimes you will not know the dimensions of an array at declaration. For this, you can declare the array with the attribute `allocatble` and a deferred shape and later `allocate` memory for the array. After you no longer need an allocated arrray, you can use `deallocate` to free the memory that it is using.

The following program demonstrates how to allocate arrays.

allocate.f95

```
1 program allocation
```

```
2  implicit none
3     real, allocatable :: A(:), B(:,:)
4     integer :: i, j
5
6     allocate(A(1:5),B(3,3))
7     A = (/ 1., 2., 3., 4., 5. /)
8     write(*,*) 'A=',A
9
10    forall (i=1:3,j=1:3) B(i,j)=i+j
11    write(*,*) 'B='
12    do i=1,3
13       write(*,*) B(i,:)
14    end do
15
16    deallocate(A,B)
17
18 end program allocation
```

- To declare an array with a deferred shape, use only commas and semi-colons to assign indices. For example, to declare a 1-dimensional allocatable array `A` of integers, use `integer, allocatable :: A(:)` or for a 2-dimensional array `B` of integers, use `integer, allocatable :: B(:,:)`.
- With the allocate function, array indices can be specified as usual.

allocate - commands and output

```
gfortran allocate.f95 -o allocate
./allocate
A=   1.00000000       2.00000000       3.00000000       4.00000000       5.00000000
B=
  2.00000000       3.00000000       4.00000000
  3.00000000       4.00000000       5.00000000
  4.00000000       5.00000000       6.00000000
```