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)`	$\lvert x \rvert$
`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
```

```

   i1=8
   i2=1
   i3=3
   ! call subroutine
   call stats(n,i1,i2,i3,minimum,maximum,median,mean)
   ! after calling subroutine
   write(*,*) 'i1,i2,i3 = ',i1,i2,i3,'(after calling stats)'
   write(*,*) 'minimum = ',minimum
   write(*,*) 'maximum = ',maximum
   write(*,*) 'median = ',median
   write(*,*) 'mean = ',mean
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

```
module constants
implicit none
   integer, parameter :: rp = kind(0.d0)
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

```
program PROGRAM_NAME
use constants
implicit none
   real(rp) :: x
   ...
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

```
module constants
implicit none
   integer, parameter :: rp = kind(0.d0)
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

```
function f(x)
use constants
implicit none
   real(rp), intent(in) :: x
   real(rp) :: f
   f=log(x)-1._rp/exp(x)
end function f

function fp(x)
use constants
implicit none
   real(rp), intent(in) :: x
   real(rp) :: fp
   fp=1._rp/x+1._rp/exp(x)
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

```
subroutine secant(x0,x1,f,tol,maxstep)
use constants
implicit none
   ! subroutine arguments
   real(rp), intent(inout) :: x0,x1
   real(rp), external :: f
   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 = (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
```

```
! Left Endpoint Quadrature Rule
function left_endpoint(n,a,b,f)
use constants
implicit none
   ! number of grid points
   integer, intent(in) :: n
   ! left and right endpoints of interval
   real(rp), intent(in) :: a,b
   ! integrand
   real(rp), external :: f
   ! return value
   real(rp) :: left_endpoint
   ! local variables
   integer :: i
   real(rp) :: x, dx

   dx = (b-a)/n
   left_endpoint = 0._rp
   do i=0,n-1
      x=a+i*dx
      left_endpoint=left_endpoint+f(x)*dx
   end do
end function left_endpoint

! Midpoint Quadrature Rule
function midpoint(n,a,b,f)
use constants
implicit none
   ! number of grid points
   integer, intent(in) :: n
   ! left and right endpoints of interval
   real(rp), intent(in) :: a,b
   ! integrand
   real(rp), external :: f
   ! return value
   real(rp) :: midpoint
   ! local variables
   integer :: i
   real(rp) :: x, dx

   dx = (b-a)/n
   midpoint = 0._rp
   do i=0,n-1
      x=a+(2._rp*i+1._rp)*dx/2._rp
      midpoint=midpoint+f(x)*dx
   end do
end function midpoint
```

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

quadrature/quadrature.f95

```
program quadrature
use constants
implicit none
   ! number of grid points
   integer :: n
   ! left and right endpoints of interval
   real(rp) :: a,b
   ! integrand function and quadrature rule functions
   real(rp), external :: f, left_endpoint, midpoint
```

```
  a = 0._rp
  b = 1._rp
  n = 10

  write(*,*) 'By the Left Endpoint Rule:',left_endpoint(n,a,b,f)
  write(*,*) 'By the Midpoint Rule:',midpoint(n,a,b,f)
  write(*,*) "Analytic value in terms of 'error funciton':",.5*pi**.5*erf(1.)
end program quadrature
```

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

quadrature/quadrature.mak

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

$(BIN) : quadrature.f95
	$(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.