Chapter 6

Fortran Procedures: Subprograms and Functions

Numerical programming often requires repeating the same calculations with different data and variables. Examples include having to integrate different functions or the same function over different intervals repeatedly. Having to recode the same chunk of fortran statement is wasteful of time and error prone as typos and bugs can be introduced in the process, moreover, the time we have invested in validating a specific piece of code would have to be duplicated again and again. Most programming languages provide a mechanism to avoid these repetitions and pitfalls. In FORTRAN this is done simply and elegantly by turning algorithmically related code segments into subprograms or procedures. There are many kinds of subprograms in FORTRAN: subroutines, functions, and BLOCK DATA subprograms. Moreover, subroutines and functions can come in two flavors, internal and external. The func function already encountered in the trapezoidal rule integration is one example of an internal procedure. Subprograms are essential to build complex software as it allows programmers to think of their code in terms of algorithmic units and permits some degree of code abstraction. The benefits is that the development and validation of code can be done flexibly and in manageable units. Validated software can also be reused without having to worry repeatedly about its correctness.

6.1 Modular Design

The example of trapezoidal integration will be used again to illustrate the use of external procedures; the code for an adaptive trapezoidal integration is listed in 6.1. Clearly one can see a division of tasks among the different portions of the code. More specifically, one can distinguish the following phases in the program:

- An initialization phase that primarily reads in the input data; lines 8-9.
program trap
implicit none
integer :: N ! number of intervals
real :: a,b ! interval extrema
integer :: maxiteration ! maximum number of iterations allowed
integer :: i, iteration ! counters
real :: dx, Ans, xi, error, Ansold, tolerance

print *,'Enter a, b,tolerance, initial N, maxiteration'
read *, a,b, tolerance, N, maxiteration

do iteration = 1,maxiteration ! iteration count
dx = (b-a)/real(N) ! grid size
Ans = 0.5*( myfunc(a) + myfunc(b) ) ! Evaluate function at a and b
do i = 1,N-1
xi = i*dx + a
Ans = Ans + myfunc(xi)
enddo
Ans = Ans * dx ! Integral estimate with N points
if (iteration > 1) then ! one iteration before check
error = abs(Ans - Ansold)
if (error < tolerance) then ! absolute error
exit ! converged, exit loop
else ! if not converged do following
N = 2*N ! double number of intervals
Ansold = Ans ! store old answer
endif
endif
else
error = 0.0
endif
print *, N, Ans, error
enddo
print *, N, Ans, error ! print final answer
stop
contains
real function myfunc(x)
implicit none
real, intent(in) :: x
myfunc = x**5
return
end function myfunc
end program trap

Figure 6.1: Code for adaptive trapezoidal integration
6.2 Functions and subroutines

The logic of the program is pretty simple, the user input the parameters that include an initial estimate for the number of intervals, and the desired accuracy. The program then iterates \texttt{maxiteration} times calculating the integral with different values of \texttt{N}. If the error in two consecutive estimates is less than the required tolerance, the loop is exited (line 21); else the number of intervals is doubled (line 23), the previous result is stored (line 24) and the calculation is repeated starting from line 10. One drawback of the present design is that we have to check at line 18 if we have already an estimate of the answer before performing the convergence check.

6.2 Functions and subroutines

The code listed in figure 6.1 already uses a function call \texttt{myfunc} to calculate the function. It was used to avoid having to explicitly code the formula for the initialization of the sum, line 12 where it is used twice, and its repeated evaluation in line 15. Now that the code has been packaged in the function \texttt{myfunc}. Any changes to the integrand need to be performed once by modifying that function, the rest of the code would remain untouched. It is important to stress at this point that \texttt{myfunc} is a self contained procedure that receives an input, \texttt{x} and returns an output named \texttt{myfunc}. At run time, the function is called twice at line 12, once for \texttt{a} and \texttt{b}, and upon each iteration of the loop at line 15. Each time the \texttt{dummy argument} of the function, \texttt{x}, is associated in turn with the variables \texttt{a}, \texttt{b} and \texttt{xi}. One important feature of functions is that they can accept multiple arguments as input but can only return one argument as output.

Another functional unit amenable for “packaging” in a function is the code that actually evaluates the integral. It needs, three input variables and produces exactly one output. The code to achieve this is shown in figure 6.2. The procedure \texttt{Trapezoid} is a real function that receives the interval parameters and the number of cells, and that returns in its name the trapezoidal estimate to the integral. The list of variables in the function statement are called \texttt{dummy arguments} because they are only place holders for the actual arguments that will be passed to the procedure. These dummy arguments must have their data type declared; it is also good practice to declare their intent, whether they are input argument, \texttt{intent(in)}, output arguments, \texttt{intent(out)} or both, \texttt{intent(inout)}.

Finally it is possible to use, with some stretch, a subroutine call to package the input statements, and another one to package the iterative convergence check on the trapezoidal estimate. The extra code needed to achieve these results is shown in figure 6.3.
program trap
  implicit none
  integer :: N ! number of intervals
  real :: a,b ! interval extrema
  integer :: maxiteration ! maximum number of iterations allowed
  integer :: i, iteration ! counters
  real :: dx, Ans, xi, error, Ansold, tolerance

  print *,’Enter a, b,tolerance, initial N, maxiteration’
  read *, a,b, tolerance, N, maxiteration
  Ansold = Trapezoid(a,b,N) ! first estimate of integral
  do iteration = 2,maxiteration ! iteration count
    N = 2*N ! double number of intervals
    Ans = Trapezoid(a,b,N) ! new estimate of integral
    error = abs(Ans - Ansold) ! absolute error
    if (error < tolerance) then
      exit ! converged, exit loop
    else
      Ansold = Ans ! save old answer if not converged
    endif
    print *, N, Ans, error ! monitor adaptation progress
  enddo
  print *, N, Ans, error ! print final answer
  stop
contains
real function Trapezoid(a,b,N)
  implicit none
  integer, intent(in) :: N
  real, intent(in) :: a,b
  integer :: i
  real :: x,dx, res
  dx = (b-a)/real(N)
  res = 0.5 * (myfunc(a)+myfunc(b))
  do i = 1,N-1
    x = i*dx + a
    res = res + myfunc(x)
  enddo
  Trapezoid = dx * res
  return
end function Trapezoid

Figure 6.2: Modified portions of trapezoidal integration scheme
program numinteg
  implicit none
  integer :: N,maxiteration
  real :: a,b,aInteg, tolerance
  call Input(a,b,tolerance,N,maxiteration)
  call AdaptTrapezoid(aInteg,a,b,tolerance,N,maxiteration)
  print *,’Integral = ’, aInteg
  stop
contains ! the functions Trapezoid and myfunc should go below here

!***********************************************************************
! Adaptive integration
!***********************************************************************
subroutine AdaptTrapezoid(aInteg,a,b,tolerance,N,maxiteration)
  implicit none
  integer, intent(inout) :: N, maxiteration
  real, intent(in) :: a,b, tolerance
  real, intent(out) :: aInteg
  integer :: iteration
  real :: Ans, Ansold,err
  Ansold = Trapezoid(a,b,N) ! first estimate of integral
  do iteration = 2,maxiteration ! iteration count
    N = 2*N ! double number of intervals
    Ans = Trapezoid(a,b,N) ! new estimate of integral
    error = abs(Ans - Ansold) ! absolute error
    if (error < tolerance) then
      exit ! converged, exit loop
    else
      Ansold = Ans ! save old answer if not converged
    endif
    print *, N, Ans, error ! monitor adaptation progress
  enddo
  return
end subroutine AdaptTrapezoid

end program numinteg

!***********************************************************************
! Input parameters
!***********************************************************************
subroutine Input(a,b,tolerance,N,maxiteration)
  implicit none
  integer, intent(out) :: N, maxiteration ! number of subinterval
  real, intent(out) :: a,b, tolerance ! interval
  print *,’Enter a, b,tolerance, initial N, maxiteration’
  read *, a,b, tolerance, N, maxiteration
  return
end subroutine Input

Figure 6.3: Further modularization of the trapezoidal program
CHAPTER 6. FORTRAN PROCEDURES: SUBPROGRAMS AND FUNCTIONS

6.2.1 Remarks about the main program

- The main code has been shortened because the work has been relegated to subroutines.

- The logical sequence of execution is however neatly encapsulated by the sub-program units by giving subroutines informative names. Different subroutines in a single program must have different names.

- A subroutine is invoked by the call statement. A call is a kind of branch, transferring execution to the subroutine. When a subroutine is finished control is returned to the next statement after the invoking call.

- The variables that are used in each subroutine invocation are passed to the routine in a parenthesized list usually referred to as an argument list; more specifically they are called actual arguments in the call statement.

6.2.2 Remarks about the routines

- The beginning and end of a subroutine are defined by the pairs of subroutine, and end subroutine statements. The pair define the scope of a subroutine. The end statement must be present to tell the compiler there are no more source line in the routine. Classical fortran does not support the end subroutine statement, only end. The name given in the subroutine must match the one in the call statement for the subroutine to be invoked.

- When Input is invoked, control is transferred to the first executable statement inside the routine which is print *, 'Enter a,b,N'. After the return, control is transferred back to the first statement after call Input. Every subprogram must execute a return (or a STOP which returns control to the Operating System).

- The arguments listed in the subroutine statement must match in number, order and type, the argument appearing in the call statement that invoked the subroutine. However, the names of the variables need not be the same as the name in the calling routine. This is essential if the same routine need to be called repeatedly with different actual parameters. The variables listed in the subroutine statement are called dummy argument because their names are unrelated to those in other routines including the calling routine. For example the routine Trapezoid returns its result in variable res which is then stored in variable ainteg in subroutine AdaptTrapezoid.

- The argument list is declared in each subroutine. You can see that an attribute is attached to each argument that informs the compiler how we are going to use these arguments. Some of these are solely input arguments.
and will not be changed by the routine, these are given the \texttt{intent(in)} attribute. Others have no value upon entry to the subroutine, but their value is assigned upon exist; they are given the \texttt{intent(out)} attribute. Finally some arguments, have a value upon entry that is modified in the course of the subroutine execution, these are given the \texttt{intent(inout)} attribute. The \texttt{intent} attribute is not part of classical FORTRAN, but is extremely useful in documenting the code by immediately telling the programmer what variables are modified. Furthermore, declaring the attribute allows the compiler to first check and alert the programmer if a value is changed inadvertently, and second to perform some optimization.

- The variables $x$ and $i$ are local to subroutine \texttt{Trapezoid} and they have nothing to do with variables having the same name in other routines, including the calling routine. The \texttt{scope} of these variables is local to subroutine \texttt{Trapezoid}.

- Local variables are not necessarily saved across subroutine invocation even though some compiler do so by default. If we want to insure that local variables keep their values across, we would use the attribute \texttt{save} in declaring the variable.

- The main program and the subroutines it calls are each compiled separately, one at a time. This separate compilation is the reason that the names of variables in dummy argument lists and local variables, having nothing to do with the names of variables in other routines. The compiler in translating each routine into machine language has no way of knowing about any other routines to be used in the program. Only the loader knows about all the pieces that must be linked together to create the executable; by that time all variable names would have been translated into memory addresses. Separate compilation is one of the great features of FORTRAN as it allows the building and use of \textit{libraries}.

- Subprograms can invoke other subprograms to any depth. The only exception is that subprogram can invoke themselves in classical FORTRAN. \texttt{Recursion} is actually a new feature added to FORTRAN 90.

### 6.3 Call by reference

How are arguments transferred from the invoking subroutine to the called subroutine? One could imagine that a subroutine copies actual arguments into its dummy arguments at invocation, performs the calculations, and then the output arguments are copied back after the return. That would actually be a waste of memory and CPU, both precious commodities in the early days of computing.
FORTRAN uses instead what is known as call by reference or call by name because what is passed to the subroutine is not the value of the actual argument but its memory address. Subprograms manipulate the actual parameters directly by addressing the memory location where they are stored. Address calculation takes place entirely behind the scenes and memory locations are not directly available to the FORTRAN programmer. So if a subrprogram changes the value of a dummy argument, the value of the actual argument gets changed immediately. For this reason, passing arguments that have the same name can lead to surprising results if their values are changed and can be potentially dangerous. This is called aliasing of arguments.

6.4 Function Subprograms

Function subprograms are very similar to subprograms but there are some important differences.

- Functions return a single variable in the name of the function itself, no separate variable need be devoted to the return value.

- A subroutine does not have a type (and does not need to have one). But a function’s name returns a value and hence must have a type. The compiler knows about its built-in functions like sqrt but it has to be told the type of user defined functions.

- There are specific functions for every variable type, dsqrt for example returns a double precision variable. Fortran, however, allows some functions to have generic names, and the function invoked depends on the data type of the input argument. For example sqrt(4.0) returns a 32-bit real result whereas sqrt(4.d0) returns a 64-bit real since 4.d0 is a double precision constant.

- A subroutine is invoked with call whereas a function is invoked simply by mentioning its name in an expression.

- A subroutine is allowed to not have any parameters, but classical FORTRAN insists that a function must have at least one parameter.

- The scope of a function are delimited by the function and end function statements.

- The function’s name must be assigned a result and the compiler should complain if it is not.

- Functions and subroutines can invoke other functions. Special considerations should be taken to handle recursive functions.
6.5 Interface Blocks and Modules

The use of functions and subroutines hinges on effectively passing the data between the calling and called procedures correctly. There are many pitfalls that expose programmers to errors, such as missing arguments, wrong data types correspondence, ... The language introduces a number of constructs to minimize the chances of introducing bugs. The first one consists of using interface blocks that define the type of arguments passed, and the other relies on using modules. Interface checking is but one of the benefits of using modules and we will defer their presentation to the next section.

6.5.1 Interface block

An interface block is used to alert the calling program to the list and type of arguments a subroutine expects. It is basically, the subroutine or function but stripped of all its execution statements and declaration of local variables. An interface block would look like the following:

```fortran
interface
  function myfunc(x)
    real, intent(in) :: x
  end function myfunc
end interface
```

and would be inserted in the program unit that calls the function myfunc. Another example is

```fortran
interface
  subroutine Input(a,b,N)
    real, intent(out) :: a,b
    integer, intent(out) :: N
  end subroutine Input
end interface
```

6.5.2 Modules and procedures

The above is an example of a user-coded interface. When the program unit called is contained within another program unit or within a module, the interface block is built automatically by the compiler and the programmer need not bother with cutting and pasting these pieces of codes. Modules and procedures are powerful tools that allow program units to share data and procedures. For the moment we are going to limit ourselves to using these constructs solely for the purpose of building these interfaces automatically. An example module that gathers the functionality required in estimating the integral and verifying it is shown in figure
module trapez
contains
  !***********************************************************************
  ! Adaptive integration
  !***********************************************************************
subroutine AdaptTrapezoid(ainteg,a,b,tolerance,N,maxiteration)
  implicit none
  integer, intent(inout) :: N, maxiteration
  real, intent(in) :: a,b, tolerance
  real, intent(out) :: ainteg
  integer :: iteration
  real :: Ans, Ansold, err
  Ansold = Trapezoid(a,b,N) ! first estimate of integral
  do iteration = 2,maxiteration ! iteration count
    N = 2*N ! double number of intervals
    Ans = Trapezoid(a,b,N) ! new estimate of integral
    err = abs(Ans - Ansold) ! absolute error
    if (err < tolerance) then
      exit ! converged, exit loop
    else
      Ansold = Ans ! save old answer if not converged
    endif
    print *, N, Ans, err ! monitor adaptation progress
  enddo
end subroutine AdaptTrapezoid
  !***********************************************************************
  ! Numerical integration via trapezoidal rule
  !***********************************************************************
real function Trapezoid(a,b,N)
  implicit none
  integer, intent(in) :: N
  real, intent(in) :: a,b
  integer :: i
  real :: x,dx, res
  dx = (b-a)/real(N)
  res = 0.5 * (myfunc(a)+myfunc(b))
  do i = 1,N-1
    x = i*dx + a
    res = res + myfunc(x)
  enddo
  Trapezoid = dx * res
end function Trapezoid
end module trapez

Figure 6.4: Module example that encapsulates the functional units required in computing the integral of a function
6.6. **RECURSIVE PROCEDURES**

Fortran by default does not permit functions of subroutines to call themselves either directly or indirectly. However, complex algorithms can often be cast elegantly in terms of recursive calls, that is the algorithm divides the steps into subtasks that are identical by require different data. In that sense recursive procedures resemble do-loops by are much more powerful. They are also more expensive as a procedure call, and as local variable must be created for each instance the procedure is called.

Let us revisit the example of trapezoidal integration. Our adaptive procedure doubled the number of points uniformly, regardless of the shape of the function. This can be wasteful in places where the function behaves mostly like a straight line since the trapezoidal error is already small there. The error analysis reveals that the errors are largest in regions where the curvature of the function is large. A more astute adaptation strategy would resolve the high curvature regions, only sparse sampling of the function would be necessary in regions where it behaves
linearly. One strategy to achieve this objective is via the following pseudo-code.

1. 1-trapezoid estimate of the integral is

   \[ S_1 = (b - a) \frac{f_a + f_b}{2} \]

2. 2-trapezoid estimate of the integral is

   \[ S_2 = \frac{b - a}{2} \left( \frac{f_a}{2} + f \left( \frac{a + b}{2} \right) + \frac{f_b}{2} \right) = \frac{S_1}{2} + \frac{b - a}{2} f(x_m) \]

   where \( x_m = \frac{(a + b)}{2} \) is the mid-point of the interval.

3. Clearly if \( S_2 = S_1 \) then the curve is a straight line in the interval \( a \leq x \leq b \), and we are done. The answer \( S_2 \) might still be acceptable if \( |S_2 - S_1| < \epsilon \), a specified tolerance.

4. If the error is unacceptable, then I would need to divide the interval \([a \ b]\) into two subintervals, \([a \ x_m]\) and \([x_m \ b]\) and calculate the trapezoidal estimate separately in each one. But these last two steps are exactly identical to the step started at 1. We merely need to repeat steps 1-3, with the tolerance specified at \( \epsilon/2 \) (because I divided the interval) to calculate these integrals.

The code listed in 6.6 implements the algorithm described above. Notice that the statement `recursive` is required ahead of the function declaration to make the function recursive. A critical issue in any recursive procedure is to make sure that it terminates at some points. In the present instance the stopping criterion is embedded within the error bounds which does not call the function if it drops below the tolerance.
6.6. RECURSIVE PROCEDURES

!***********************************************************************
! Numerical integration via trapezoidal rule
!***********************************************************************
recursive real function TrapezoidRec(a,b,tolerance)
  implicit none
  real, intent(in) :: a,b,tolerance
  real :: xm,dx, est1, est2, error, myfunc
  dx = (b-a) ! interval width
  xm = 0.5*(a+b) ! interval midpoint
  est1 = 0.5 * (myfunc(a)+myfunc(b)) * dx ! one-trapeze estimate
  est2 = 0.5 * (dx * myfunc(xm) + est1) ! two-trapeze estimate
  error = abs(est2 - est1)
  if (error > 3.0*tolerance) then
    est1 = TrapezoidRec(a,xm,0.5*tolerance)
    print *, xm
    est2 = TrapezoidRec(xm,b,0.5*tolerance)
    TrapezoidRec = est1 + est2
  else
    TrapezoidRec = est2
  endif
  return
end function TrapezoidRec

Figure 6.6: Recursive trapezoidal integration

Figure 6.7: Quadrature points for the recursive trapezoidal rule, the grid spacing for these boxes is shown in the right hand figure.