Appendix C

A Fortran Primer: (and cheat sheet)

This section will provide a basic intro to most of the commonly occuring features of Fortran that you will need for the course. This list is by no means exhaustive, but it should be enough to get you where you need to go. For more information, We have extensive fortran manuals scattered about the observatory. By the end of this section you should understand the basic data types, the structure of arrays, standard arithmetic operations, loops and conditional statements, subroutines and functions, and basic IO. This section will also briefly discuss how to build programs using `make` and how to debug them using `dbxtool`/`debugger`.

C.1 Basic Fortran Concepts and Commands

Data types for the most part there will be only three basic types of data you will have to deal with, integers, floating point numbers and characters. In fortran these data types are declared as

`integer` exact whole numbers (-3, 0, 5 234), usually stored in 4 bytes

`real` inexact representation of decimal numbers (3.1415, 27.2, 1.e23). Usually stored as 4 bytes, good for about 6-9 significant digits, ranges from about 10^{-38}–10^{38}

`double precision` same as real but much larger range and precision. Usually stored as 8 bytes, good for about 15-17 significant digits, ranges from about 10^{-308}–10^{308}. You can get more precision than this but you should have a good reason.

`character, character*n` either a single character or a string of characters of length `n`.

Constant and Variable names A constant or variable can have any name with up to 32 characters (To play it safe though, Standard Fortran allows only 6 characters). The name must start with a letter, but can have most of the

alphanumeric characters in it. Fortran also has the annoying feature of **implicit typing** so that if undeclared, any variable name that starts with the letters `i` through `n` are assumed to be integers, everything else is assumed real. Rigorous programming practice suggests that you start every program with `implicit none` and declare every variable. It's annoying but it keeps you honest. I will try to do this in my programs. A sample program declaration might look like

```
implicit none
integer npnts, n
real  c, dcdt
real  car(1000),tar(1000)
character*40 fileout
```

arrays Perhaps the most important feature for modeling is arrays. Arrays are simply ordered sets of numbers that can be addressed by index. Every array has a name (e.g. `car` or `tar` above) and a length (here both `car` and `tar` are arrays of real numbers of length 1000). Arrays can be of integers, reals, double precision or even characters (`fileout` is actually an array of 40 characters). Each member of an array can be addressed as by specifying its index in the array (`car(10)`, `tar(n)` etc.). Arrays can also have up to 7 dimensions. A two dimensional array `a(10,10)` can be thought of as 10 1-dimensional arrays of 10 numbers each (a total of 100 elements). Most importantly in fortran, the leading index increases fastest i.e. for `a(i,j)`, the `i=1` and `i=2` are next to each other in memory.

Simple operations integer and floating point numbers and arrays of them can all be operated on by standard mathematical options. The most commonly used arithmetic are

`=` assignment `x=y`

`**` exponential `a=x**y`

`/, *` divide, multiply `a=x/y, or a=x*y`

`+, -` add subtract `a=x+y, or a=x-y`

The order of evaluation is standard algebraic and parentheses can be used to group operands. In addition to the simple operations, Fortran also has some built in intrinsic functions. The most commonly occuring are

trigonometric functions `sin(x), cos(x), tan(x)`, `asin(x)`, `acos(x)`, `atan(x)`, `atan2(x)` (inverse trig functions) `sinh(x)`, `cosh(x)`, `tanh(x)` etc.

exponential functions `exp(x)`, `log(x)` (natural log), `log10(x)` (log base ten), `sqrt(x)` square-root.

conversion functions these functions will convert one data type to another, e.g. `int(x)` returns the integer value of `x`, `real(?)` converts anything

to a real, `dble(?)` converts anything to a double (also operations for complex numbers)

misc. functions see table 6.1

Program flow control Any program will just execute sequentially one line at a time unless you tell it to do something else. Usually there are only one of two things you want it to do, loop and branch. The control commands for these are

`do` **loops** Do loops simply loop over a piece of internal code for a fixed number of loops and increment a loop counter as they go. do loops come in the standard line number flavour

```
      do 10 i=1,n,2
        j=i+1
        other stuff to do
 10   continue
```

(note the spacing of the line number and starting commands at least 6 characters over is important (stupid holdover from punch cards). Or in the non-standard but more pleasant looking form

```
      do i=1,n,2
        j=i+1
        other stuff to do
      enddo
```

Do loops can be nested a fair number of times (but don't overdo it)

conditional statements Being able to make simple decisions is what separates the computers from the calculators. In fortran separate parts of the code can be executed depending on some condition being met. The statements that control this conditional execution are

`if` a single line of the form

```
      if (iam.eq.crazy) x=5.
```

will assign a 5. to the variable `x` if the expression in parentheses is true, otherwise it will simply skip the statement.

block if statements More useful blocks of execution can be delimited by block ifs of the form

```
      if ( moon.eq.full ) then
        call howl(ginsberg)
        x=exp(ginsberg)
      endif
```

the block if statements can also be extended to have a number of condition statements. If there are only two, it looks like

```
      if ( moon.eq.full ) then
        call howl(ginsberg)
        x=exp(ginsberg)
```

Table 6-1 Arithmetic Functions (continued)

Intrinsic Function	*Definition*	*No. of Args*	*Generic Name*	*Specific Name*	*Type of*	
					Argument	*Function*
Absolute Value	|a| Read Note 6 $(ar^2 + ai^2)^{**}(1/2)$	1	ABS	IABS ABS DABS CABS CQABS♦ QABS ♦ ZABS ♦ CDABS♦	Integer Real Double Complex Complex*32 Real*16 Complex*16 Complex*16	Integer Real Double Real Real*16 Real*16 Double Double
Remainder	a1-int(a1/a2)*a2 Read Note 1	2	MOD	MOD AMOD DMOD QMOD ♦	Integer Real Double Real*16	Integer Real Double Real*16
Transfer of Sign	|a1| if a2 ≥ 0 -|a1| if a2 < 0	2	SIGN	ISIGN SIGN DSIGN QSIGN ♦	Integer Real Double Real*16	Integer Real Double Real*16
Positive Difference	a1-a2 if a1 > a2 0 if a1 ≤ a2	2	DIM	IDIM DIM DDIM QDIM ♦	Integer Real Double Real*16	Integer Real Double Real*16
Double and Quad Products	a1 * a2	2		DPROD QPROD ♦	Real Double	Double Real*16
Choosing Largest Value	max(a1, a2, …)	≥ 2	MAX	MAX0 AMAX1 DMAX1 QMAX1 ♦	Integer Real Double Real*16	Integer Real Double Real*16
				AMAX0 MAX1	Integer Real	Real Integer
Choosing Smallest Value	min(a1, a2, …)	≥ 2	MIN	MIN0 AMIN1 DMIN1 QMIN1 ♦	Integer Real Double Real*16	Integer Real Double Real*16
				AMIN0 MIN1	Integer Real	Real Integer

```
      else
         call wait(1,month)
      endif
```

or if there are several conditions

```
      if ( expression ) then
         statements galore....
      elseif (expression2)
         more statements
      elseif (expression3)
         are you tired of this yet
      else
        default statements
      endif
```

Note: a block if of this type will execute the first true expression and the jump to the end if (even if several conditions are true).

relational operators The statement `moon.eq.full` is a conditional statement that evaluates to `true` if the variable `moon` is logically equivalent to the variable `full` (don't use `=` for `.eq.` very different animals). The operator `.eq.` is a relational operator that tests for equivalence. Other relational operators are `.lt.` (less-than),`.le.` (less-than-or-equal-to) ,`.ge.` (greater-than-or-equal-to),`.gt`] (greater-than). Individual conditional statements can also be linked together using the operators `.and.`, `.or.`, `.not.` (and a few less useful things like exclusive or `.xor.`. Examples include

```
      if ((moon.eq.full).and.(i.eq.werewolf))
     &     call runlikehell()
        if ( (x.eq.0.5).or.(i.le.2) ) then
          x=x*i
        endif
```

Subroutines and Functions Subroutines and functions are separate pieces of code that are passed arguments, do something and return. Subroutines and functions are similar except that functions return a value and are used like the intrinsic functions i.e.

```
integer myfunction, k,j
external myfunction
real x
...
k=myfunction(x,j)
```

and are declared like

```
integer function myfunction(r,i)
real r
integer i
  myfunction=mod(int(r),i))
```

```
  return
  end
```

Note the variables `r` and `i` are "dummy variables" that hold the place of `x` and `j` in the main routine. Subroutines also have dummy variables but are "called" and don't return a value (although it will often change the values of what it is passed). Example subroutine call would be

```
integer  k,j
real x(100),a
....
j=10
a=2.5
call arrmult(x,j,a)
```

and the subroutine itself would look like

```
subroutine arrmult(ar,n,con)
integer n
real ar(n),con
integer i
do i=1,n
  ar(i)=ar(i)*con
enddo
return
end
```

Note that arrays are passed to subroutines and functions by name. Most importantly, the dimension of the array within the subroutine can be passed. In the above routine, only the first 10 elements of `x` are multiplied by `2.5`. If we wanted to multiply the elements from 6 to 16 we could also do

```
call arrmult(x(6),11,a)
```

Note that there are actually 11 elements between indexes 6 and 16. In addition, arrays that are declared as 1-D arrays in the main program can be operated on as variable length n-d arrays in a subroutine (and vice-versa). E.g.

```
integer  i,j
real x(1000),a
...
i=10
j=10
a=2.5
call arrmultij(x,i,j,a)
```

```
...
subroutine arrmultij(ar,m,n,con)
integer m,n
real ar(m,n),con
integer i,j
do j=1,n
  do i=1,m
    ar(i,j)=ar(i,j)*j*con
  enddo
enddo
return
end
```

We will make extensive use of this feature for efficient programming in n-dimensions. This is the one thing you cannot do well in C which is a real shame.

Input/Output The last important thing you might want to do is actually read information into a program and spit it out. This is perhaps the worst part of fortran, particularly when dealing with character strings. Nevertheless, with a few simple commands and unix, you can do most anything. A few important io concepts

logical units and open uggh, in fortran, files are referred to by "logical units" which are simply numbers. To open a file called `junk.txt` with logical unit 9 you would do something like

```
lu=9
open(lu,file='junk.txt')
```

The two most important files `stdin` and `stdout` already have logical units associated with them. `stdin` is 5 and `stdout` is 6.

reading a file to read data from a file you use the `read` statement. The principal version of this you will need to read things from the keyboard or from stdin looks like

```
read(5,*) tmax,npnts,(ar(i),i=1,3)
```

the `5` is the logical unit and the `*` says to just read in each datatype as whatever it was declared as so if `tmax` and `ar` are real and `npnts` is an integer it will assume that the numbers in stdout will be in those formats. Note the funny way of reading 3 items of array `ar` is known as an `implicit do loop` useful but clunky.

writing to a file to write data from a file you use the `write` statement which works just like read i.e.

```
write(6,*) 'here are some numbers ',tmax,npnts,(ar(i),i=1,3)
write(9,*) 'Howdy '
```

a synonym for `write(6,*)` is the `print` statement i.e.

```
print *, 'here are some numbers ',tmax,npnts,(ar(i),i=1,3)
```

is equivalent to the first of the above two lines.

C.2 A Few pointers to good coding

Comment liberally Always write your code so that 6 months (or two weeks) from now you know what it does. comment lines start with a `c` in the first column or after a `!` in any column (the `!` comment is non-standard so be careful).

make 1 code to do 1 thing , Super-duper multi-purpose codes with hundreds of options become a nightmare to maintain and debug. Since you're only writing private code for yourself, I find it is most sensible to create separate programs for each difference in boundary or initial conditions etc. Using `make` and `makefiles` can simplify this process immensely. This way if you try something and it doesn't work, you can just go back to a previous version.

Write your loops right Always write your loops with the computer in mind. i.e. your inside loop should always be over the fastest increasing index. This will give the biggest increase in performance for the least amount of work.

NO GOTO's except in dire need avoid uncontrolled use of the `goto` statement as it will lead to immense confusion. See the first chapter of Numerical Recipes for the few necessary controlled uses of `goto`.

limit `ifs` and functional calls within array loops If a loop is designed to quickly handle an array, an embedded `if` statement or heavy function calls can destroy performance (although many optimizing compilers will do some strange things to try and prevent this).

keep it simple and conservative There are loads of fancy extensions in Sun fortran that might not work on other machines. The less fancy gee-gaws you use the less you have to replace when you change platforms

use `dbxtool`/`debugger` and `make` see below and man pages, these tools will make your life much easier for organizing and debugging code.

C.3 A sample program

```
      program euler1
c*****************************************************************
c euler1:  program to calculate the concentration of a
c   single radioactive element with time using an
c   euler method
c*****************************************************************

      implicit none
      integer nmax              !maximum array size
      parameter (NMAX=500)      ! set nmax to 500
      integer npnts, n          ! number of steps, counter
      real  c, dcdt             ! concentration, decay rate,
```

```
      real tmax, t,dt           ! max time,time, timestep
      real  car(NMAX),tar(NMAX) ! arrays for concentration and time
      character*40 fileout ! character array for output filename
      integer kf, lu  ! integer for character index, output file
      integer iprinttrue, iprinterr ! flag for calculating true solution,
                                ! or error (1 yes, 0 no)
      integer mylnblnk          ! even functions need to be typed
      external mylnblnk
c
c     -------read input
c
      read(5,*) fileout
      read(5,*) tmax, npnts, iprinterr, iprinttrue
c
c     -------set up initial parameters
c

      dt = tmax/ (npnts - 1)    ! set the time step (why is it n-1?)
      c = 1.                    ! initial concentration
      t = 0.                    ! initial time
      car(1)=c                  ! store in arrays car and tar
      tar(1)=t
c
c-------loop over time steps with an euler method
c
      do n=1,npnts-1            !start the loop
         call decay1(t,c,dcdt)  ! get the decay rate at the present step
         c=c + dcdt*dt          ! take an euler step
         t = dt*n               ! calculate the time for the next step
         car(n+1)=c             ! store in array car
         tar(n+1)=t             ! store the time in array tar
      enddo                     ! end the loop

c
c-------write the solution to fileout_c.xy
c
      lu=9                      ! set the fortran logical unit (ugh!) to any number
      kf=mylnblnk(fileout,len(fileout)) ! find the last blank space in string
      print *, 'Writing file ',fileout(1:kf)//'_c.xy'
      call writexy(lu,fileout(1:kf)//'_c.xy',tar,car,npnts) !writem-out
c
c-------if iprinterr=1, then calculate and write out fractional error between
c-------solution and true solution to fileout_cerr.xy
c
      if (iprinterr.eq.1) then
         do n=1,npnts
            car(n)=car(n)/exp( -1.*tar(n)) - 1.  ! calculate error
         enddo
         print *, 'Writing file ',fileout(1:kf)//'_cerr.xy'
         call writexy(lu,fileout(1:kf)//'_cerr.xy',tar,car,npnts)
      endif
c
c-------if iprinttrue=1, then calculate and write out true solution
c------- to fileout_ctrue.xy
c
      if (iprinttrue.eq.1) then
         do n=1,npnts
            car(n)=exp( -1.*tar(n)) ! calculate true concentration
         enddo
         print *, 'Writing file ',fileout(1:kf)//'_true.xy'
         call writexy(lu,fileout(1:kf)//'_ctrue.xy',tar,car,npnts)
      endif

c
c-------exit the program
c
      end


c**********************************************************************
c decay1:  subroutine that returns the rate of decay of a radioactive
c   substance as a function of  concentration (and time)
```

```
c   c is concentration
c   dcdt is the change in c with time
c   t is time
c   here, dcdt= -c
c*************************************************************************

      subroutine decay1(t,c,dcdt)
      implicit none
      real t, c, dcdt

      dcdt= -c

      return
      end

c*****************************************************************
c      mylnblnk integer function to return position of last non-character
c      in a string
c*****************************************************************

      integer function mylnblnk(str,lstr)
      character str(lstr),space
      integer l
      space=' '
      l=lstr
 10   if (str(l).eq.space) then
         l=l-1
         goto 10
      endif
      mylnblnk=l
      return
      end
C*************************************************************************
C     writexy:  writes 2 1-d arrays (x,y) to output file luout of name
C               fname.xy
C*************************************************************************

      subroutine writexy(luout,fname,xarr,yarr,npnts)
      integer luout
      character *(*) fname
      real xarr(npnts), yarr(npnts)

      open(luout,file=fname)
      do 10 j=1,npnts
         write(luout, *) xarr(j), yarr(j)
 10   continue
      close(luout)
      return
      end
```

C.4 A sample makefile

```
#########################################################################
#     makefile for the program euler1 which calculates
#    radioactive decay for a single element using an euler method
#               uses suns xtypemap to make double precision
#########################################################################

PROGRAM=euler1
OBJECTS=$(PROGRAM).o  decay1.o   writexy.o lnblnk.o
OPTLEVEL= -g
FFLAGS= $(OPTLEVEL) -xtypemap=real:64,double:64,integer:64
DESTDIR=$(HOME)/$(ARCH)


$(PROGRAM): $(OBJECTS)
$(FC)  $(FFLAGS) $(OBJECTS)  -o $(PROGRAM)

$(OBJECTS):
```

```
install: $(PROGRAM)
mv $(PROGRAM) $(DESTDIR)

clean:
rm -f *.o *.a core

cleanall:
rm -f *.o *.a core $(PROGRAM)
```