Programming Style Guide & Tips– Phys 2274

Style Guide

You will spend a lot of time coding in this course. It is therefore important that you follow good programming style conventions. Some of these are listed here.

use descriptive variable names

Calling an index `i` is traditional, but it is much better to give it a descriptive name such as `particleNumber`.

comment your code

Everything will be in your head while you are coding, but you will not have a clue what is going on the next time you have to dust your code off. Put the file name, your name, the date, and what the code does at the top of your file. You should also place revision history and compiler directives here. Use comments throughout your code to label sections, flag complicated structures, etc.

typing

Fortran has 'implicit typing', meaning that, unless you specify otherwise, variables starting with the letters `i-n` will be treated as integers, with the rest treated as reals. Do not use implicit typing. Ie, always use the statement `implicit none` in your main code and subroutines. Other languages enforce explicit typing so this is not an issue.

use structured code

Try to avoid 'spaghetti code' with complicated sequences of statements. In practice, this means avoiding `goto` statements as much as possible. Here is an example of old-school programming.

```
      DO 1002 I=1,100,4
      F=F1(FLOAT(I))
      GOTO 1003,1004,1005 INT(F)
 1003 STOP
 1004 CONTINUE
      F=F/2.0
```

```
      GOTO 1006
 1005 F=F*F
 1006 CONTINUE
 1002 CONTINUE
```

Often library code will look like this. It is ok for code written by old time masters, but is not for us.

A more modern form will look like this.

```
implicit none
integer hospitalCode,numberOfPatients,patient
real patientFee,EvaluatePatientFee
character*9 hospitals(3)
data hospitals /'UPMC','Mercy','Childrens'/

numberOfPatients = 100
print*, ' enter the hospital code you wish to use '
print*, '[1=UPMC, 2=Mercy, 3=Childrens]'
read*,hospitalCode
write(11,*) ' patientFeeCalculator '
write(11,*) '                      '
write(11,*) ' hospital: ',hospitals(hospitalCode)
write(11,*) ' patient number        fee '
write(11,*) '          '
patient = 1
while (patient .le. numberOfPatients) do
 patientFee = EvaluatePatientFee(hospitalCode,float(patient))
 ...
 write(11,*) patient,patientFee
 patient = patient + 1
end while  ! end of the patient loop
print*,' patient number and fee are in fort.11 '
stop
end
```

I use the java convention for variable names: lower case to start with, no underscores, upper case for subsequent words. Start classes with upper case letters. In fortran you can start subroutines and functions with uppercase letters.

error trapping

A part of structured programming is trapping and dealing with errors when they are generated. This is implemented, for example, in java with the `try` and `catch` statements. Unfortunately, no such functionality exists in C or fortran.

indentation and spacing

Indent and space your code appropriately. I use the 'tight C' convention, eg

```
for (i=1; i<100; i++) {
 f = f(i)
 ...
}
```

Some may use the 'open C' convention but I find too much white space makes code hard to read:

```
for (i=1; i<100; i++)
          {
          f = f(i)
          ...
          }
```

objects

An 'object' is a programming data structure that combines data and methods in one unit. Object oriented coding is a computational paradigm that permits the application of powerful techniques, including data abstraction, encapsulation, modularity, polymorphism, and inheritance. C++, java, and python are object oriented languages. Unfortunately, C and fortran are not (although fortran 2003 has some OO features). It is therefore fortunate that most scientific programming is 'procedural' and OO capabilities are not needed.

fortran and C

It is worth noting that C has no simple way to evaluate powers and has no complex variable type (for another obscure issue with C, see **precision** below). Both languages support dynamic memory allocation (although you need newer versions of fortran for this). In general C is used for systems programming and fortran is used for scientific programming, although C is becoming more popular for scientific programming as library routines are being converted.

Programming Tips

call by reference and call by value

Call by reference and *call by value* refer to different methods by which information is passed to and received from subroutines. You must be very careful to keep in mind which method your language uses!

fortran is call by reference: a parameter that is passed to a subroutine is passed by its address. Thus changes to the parameter are propagated back to the main program.

C is call by value: the parameter to be passed is copied into a new area of memory and this copy is manipulated in the subroutine. The new value of the parameter is not passed back to the main program.

matrices

Arrays are treated slightly differently in C and fortran. In C arrays default to zero-offset: namely indexing starts at 0 (see, however, the discussion in Ch. 1 of *Numerical Recipes in C*). In fortran, indexing starts at 1.

Matrix indexing is also different in the two languages.

```
\\ C:
   float a[5][9];
\\   a[i][j] is then computed as (address of a) + 9*i + j.
```

Thus looping over matrix indices should be done from right to left (j, then i).

Note that the story is completely different for `float **a;`!

```
c     fortran:

      real a(5,9)
c     a(i,j) is computed as  (address of a) + 5*(j-1) + (i-1)
```

In this case looping over matrix indices should be done from left to right (i, then j).

precision

Computers necessarily represent real numbers as approximations (exactly how, and the implications of this will be discussed in class). It is up to the programmer to decide is single precision numbers (represented by 32 bits) or double precision (64 bits) are necessary for the task at hand. In C these options are denoted `float` and `double` while in fortran they are

`real` and either `real*8` or `double precision`. In the old days chips used 32 bit (or less) registers and double precision arithmetic required more memory fetches and more multiply and adds, thus slowing code down (sometimes by as much as 50%). Recent cpus are 64 bit, so (in principle) there is no overhead to pay for using double precision. Of course your compiler should take advantage of the word size of your cpu. Finding out whether it does is often a difficult task! Unbelievably, C will convert all float operations to double and then dump the extra precision if a float receives the results.

Most library routines are in double precision. And since round off error can creep in surprisingly quickly in single precision, I recommend using double precision in all numerical work.

evolving code

If you are working on a project for a while, you will find that your code grows with ever more options. Often this brings with it kludgey `if` statements or other ways to handle multiple cases. In this case, you will be better off building separate versions of your code. This is also much safer: many times a minor change can kill a program! The only problem with this approach is a proliferation of programs that do the same thing with minor variations. If you see this happening I recommend maintaining a small text file that lists the codes with brief descriptions of what they do and how they differ (this information should also be in the header comments).

small things

Always echo all of your input data into you output data files.

For complicated code I label `end do` statements with a comment specifying what loop has ended.

Print a description of the output files at the end of your code. I have code that generates 40 or 50 output files and can not remember what they all are. You certainly won't remember when you run the code again next year!

Give sample default values for your input variables. You will not remember what you ended up using when you run this code again in a year.

stability

People often confuse physical parameters and algorithmic parameters. The former define the physics problem that you want to solve. The latter specify the method in which you solve the problem – in principle your answer should be independent of algorithmic parameters! It is up to you to confirm this! In practice this means extrapolating to infinity or zero. You can

always double or halve an algorithm parameter and confirm stability of your answer. Even better is to plot how your answer changes with algorithmic parameters, and even better is to have a theoretical understanding of the functional form of that dependence so that reliable extrapolations can be made.

code testing

If you are writing scientific code, you are doing science, and it must be accurate! This means that your code must be tested extremely carefully.

Do not code too much at one time. Develop, debug, repeat. Sketch your algorithm out before coding. Proof read your code before compiling. Check the end points of your loops.

Once the code is running check it against known limits, special cases that you have computed analytically, and other programs. If you must strike out on your own, try to make the change from known territory as minimal as possible (for example, substituting a function with a complicated potential for the test square well function).

Debugging Monte Carlo code is extremely difficult! You need analytic limits and simple cases worked out for comparison.