**Abstract**

A course on scientific computation with fortran.

# An Elementary Course on Scientific Computation with Fortran

Dhrubaditya Mitra

April 26, 2012

A course of 5 weeks. Each week consists of three 1.5 hr lecture. Each lecture will be roughly 30 minutes of lecture and 1 hour of hands on session. The main topic is scientific computation. The method of teaching will use fortran heavily. You can use any other language (e.g., C, C++) you wish (but not packages software e.g, mathematica or matlab).

A pdf file of this notes is also available at `http://www.nordita.org/~dhruba/teach/scientific_compu/article.pdf`

## 1 Syllabus

1. Errors, Floats, precision arithmatic
2. Matrices, diagonalisations, inversions.
3. Integration
4. Interpolation and extrapolation
5. Extermisation.
6. Solution of (linear and non-linear) ODEs with different schemes, conservation laws, stiff equations.
7. Random numbers
8. FFT

9. basic of PDEs

10. Computation and hardware.

# 2 Schedule

Monday (10-11:30), Wednesday (9-10:30) and Friday (12:30-14) in room 122:028, the NORDITA astrophysic seminar room in house 12. The first meeting of the course will be on 5 th of November, Friday.

# 3 Course material

Primarily three sources, Ref. [8] and Ref. [1]. Individual lectures may use other sources. Although I shall mostly not use Ref. [6] in these elementary lecture (except maybe the last one where we deal with random numbers), if you are serious about scientific computation it is absolutely necessary that you are aware of these classics. Also have a look at Don Knuth's webpage `http://www-cs-faculty.stanford.edu/~uno/`.

The course has a webpage `www.nordita.org/~dhruba/` look for "scientific computation" link in the left (blue) column. I shall try to put a sketch of the forthcoming lectures. A very similar course from which I may borrow quite a number of ideas can be found here in the website of Ake Nordlund `http://www.astro.ku.dk/~aake/`.

# 4 Lecture 0

This is a short and intense course, so you have to come prepared and also to keep pace you have to work hard. To attend the first lecture you have to go through some prerequisites. These lectures are not for absolute beginners, e.g., those who have not seen a computer before. If you are not equipped with any of those listed below it will be difficult for you to follow what is going on.

1. A laptop with some form of Linux/Unix running in it. If you have a windows laptop you can install virtual box `www.virtualbox.org` in it and run Linux. Among different flavours of Linux I use ubuntu. You are welcome to use any as long as you know how to use it.

2. Your computer must have the following software, a fortran compiler (e.g., gfortran), the utility called "make", some software to plot, e.g., matlab/octave/gnuplot/matplotlib/IDL and you must know how to plot simple plots using that software.

3. Some knowledge of Unix/Linux, e.g., you must know how to open a file, edit it, what a "shell" is and elementary uses of it. How to compile a fortran code and how to run the executable.

4. Also elementary knowledge of computer programming, e.g., how to do simple arithmatic in fortran and simple input/output. To be concrete see that you understand and can run the fortran programs given below.

```
!*********************************!
program hello

write(*,*)'hello world'

end program
!*********************************!
!*********************************!
program square
integer, parameter :: N=20
integer :: i,j
do i=1,N
  j=i*i
  write(*,*) i,j
enddo
end program
!*********************************!
```

Compile and run each of these two program in your laptop. In the first case you should see the text "hello world". In the second case two columns of integers. You can redirect the output from the second program to a file and the plot the two columns of the file in the software of you choice to see a parabola. Below is how I do it.

```
> gfortran square.f90
> ./a.out
```

```
 1           1
 2           4
 3           9
 4          16
 5          25
 6          36
 7          49
 8          64
 9          81
10         100
11         121
12         144
13         169
14         196
15         225
16         256
17         289
18         324
19         361
20         400
```

Then I want to write the output to a file (redirect), and plot it. I am using a combination of numpy,scipy and matplotlib to do this. You can install them in your computer and use them. In ubuntu this can be done through synaptic. Otherwise you are welcome to use any other software you know of.

```
>./a.out >sq
> ipython --pylab
Python 2.6.5 (r265:79063, Apr 16 2010, 13:09:56)
Type "copyright", "credits" or "license" for more information.

IPython 0.10 -- An enhanced Interactive Python.
?         -> Introduction and overview of IPython's features.
%quickref -> Quick reference.
help      -> Python's own help system.
object?   -> Details about 'object'. ?object also works, ?? prints more.

  Welcome to pylab, a matplotlib-based Python environment.
  For more information, type 'help(pylab)'.
```

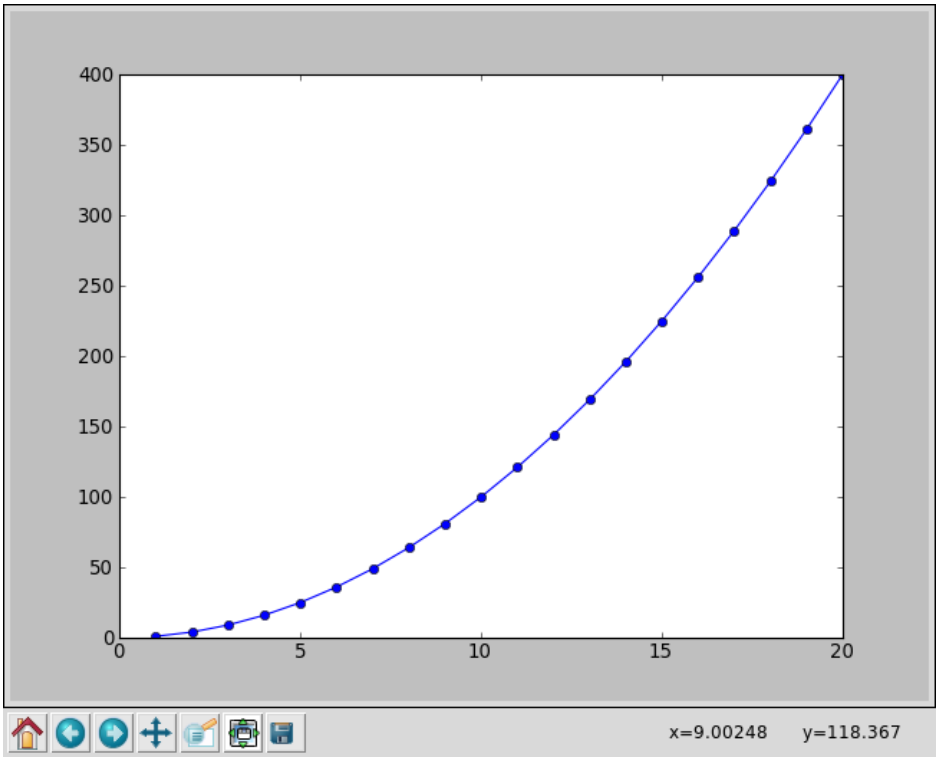


Figure 1: Plot of a parabola

```
In [2]: pp=numpy.loadtxt('sq')
In [4]: plot(pp[:,0],pp[:,1],'o-')
Out[4]: [<matplotlib.lines.Line2D object at 0xa897c4c>]
```

The output of the plot is in a separate window which is shown in Fig. 1

# 5 Lecture 1

Finding the root of a quadratic equation – How one can loose precision – example of using the appropriate numerical method – a simple fortran code to find root – How does floating point arithmatic works in a computer – root finding for higher order algabraic equations – Newton-Raphson in one dimension.

Let us start with simple algebra. First few words of caution, all programs

that we write in this course, or rather in rest of your life, however small, must be commented. With this we start writing our first program,

A fortran program must begine with

```
program <name of program>
```

and end with

```
end program
```

This file must be saved and generally has the extension `.f90`. A comment line must begin with a "bang" `!`

```
! code to add two integers
```

We must make a distinction between integers and real numbers. The reason simply is that to make a computation we need to use a base of our number system. Whatever base we use some real numbers will need an infinite number of digits for a accurate representation in that base. For example the fraction 1/3 has an infinite representation 0.33333... in base 10. With a computer with finite memory, i.e., any computer, we can store a few of these infinite number of digits. Hence real numbers will have an incomplete description in the computer. Hence they must be treated differently in a computer.

## 5.1 Integers

In fortran, before you begin computation you must define each variable that you use, i.e., you must specify whether they are real numbers or integer (or characters e.g., alphabets but we are not getting into that now). Historically this was not necessary in fortran. There was an implicit convection that a variable name which started with the letter `i, j, k` (and some others which I dont care to remember) are integers, others were real numbers. We shall never use such convention but due to a mistaken respect for backward-compatiability fortran compilers still use this convention. But you can rule that out by saying, in the second line of your program,

```
implicit none
```

I strongly recommend that you use this in every piece of fortran code that you write.

Let us then start with two integers

```
integer :: i,j
```

Note the single space between `integer` and `:` above. You can specify as many integers as you like. But there are limits to the number of characters you can have in a line. The limit is NN number of characters for some compilers and NNN for some others. Historically in fortran this limit used to be 77 characters. If you need to have a line longer than the limit you can put the line continuation character `&` . In other words you can have lines like

```
integer :: i, &
          j
```

This applies to any line in the code.

Let us get back to the main part of the code after this digression. Once you define your integers by the command above they are also initialized. A good compiler initializes number to zero. A bad compiler to any particular value. Let us find out what our compiler does. To do that we write out the two integers we have just defined

```
write(*,*) 'integers', i,j
```

This is one of the simplest way to write something out from a fortran program. We shall deal with far more complicated input and output later, but for now this will suffice.

So the program looks like

```
!*********************************!
program add_integers
! add two integers
integer :: i,j
  write(*,*) 'integers',i,j
end program
!*********************************!
```

In my computer with the gfortran compiler, I get

```
integers  1074051392   137394604
```

So it is always prudent to initialize your variables. Which can be done by

```
!*********************************!
program add_integers
! add two integers
integer :: i=0,j=0
  write(*,*) 'integers',i,j
end program
!*********************************!
```

## 5.2 Exercise

1. Some compilers comes with the option of initializing all the integers in the code with a fixed value that you can set. For example gfortran comes with the option $-$`finit` $-$ `integer` $=$ `n`. Use this option with the first version of the program `add_integers` to initialize `i` and `j` to any value you want.

2. Integer arithmatic: In fortran the basic arithmatic operations are

   ```
   !addition
   k=j+j
   !subtraction
   k=i-j
   !multiplication
   k=i*j
   !division
   k=i/j
   !exponentiation
   k=i**j
   ```

   Write a small code that does all these operations and write the result. Then experiment with this code to see how large an integer you can represent. Can you guess how large this value will be ? If you want bigger integers you can use the definition

   ```
   integer(8)
   ```

   or in the old style (still followed by some compilers)

   ```
   integer*8
   ```

3. *Integer division:* Note that in the division above when you divide 2 by 3 you do not get a fraction but 0. Can you think of interesting use of this idea ?

4. *modulo function:* In fortran implemented by the command

```
k=mod(i,j)
!or
k=modulo(i,j)
```

   The two definitions are the same for positive `i` and `j`, v.i.z.,

$$k = i - \lfloor \frac{i}{j} \rfloor \tag{1}$$

   But for negative values of `i` or `j` the convention is that in the former case `k` has the same sign as `i` and in the latter case it has the same sign as `j`. Write a code that calculates the modulo function and check that the above is true.

## 5.3 Floating point numbers

In a computer we must use finite number of memory to store a number. The representation of real numbers in a computer is thus necessarily incomplete. One of the ways, rather the most common way, real number arithmatic is done in a computer is by representating them by floating points. For example the number $1/10$ has *a* floating point representation as $1.00 \times 10^{-1}$. This particular floating point representation uses three digits and one exponent. In general the representation may look like,

$$\pm d_0.d_1 d_2 \ldots d_{p-1} \times b^e \tag{2}$$

where we use a representation with $p$ "significands" or significant digits, base $b$ and exponent $e$. The representation has the following properties:

1. First there must necessarily be a maximum and minimum value of $e$, $e_{\min}$ and $e_{\max}$ this determines what is the biggest and the smallest number we can represent.

2. Next and more importantly you must realise that for any base $\mathtt{b}$ there exists (rational) number which requires infinite number of significant digits. For example the number $\mathtt{1/3}$ needs infinite number of digits in base $\mathtt{10}$. Hence in base $\mathtt{10}$ for any finite number $\mathtt{p}$, $\mathtt{1/3}$ always lies between two floating points.

3. Note also that the floating point representation of a number is not even unique. The number $\mathtt{1/10}$ can, for example, be represented in two different ways $\mathtt{1.00E-1}$ and $\mathtt{0.01E0}$ in the floating point representation with $\mathtt{b} = \mathtt{10}$ and $\mathtt{p} = \mathtt{3}$. If we demand that the leading digit will be nonzero then we have what is called a *normalized* representation which is unique. Unfortunately that implies that you cannot store $\mathtt{0}$ in such a system. An obvious way to store zero is as $\mathtt{1.0E}e_{\min}\mathtt{-1}$. Hence if we have $\mathtt{k}$ bits to store the exponents, only $2^{\mathtt{k}} - \mathtt{1}$ values of the exponent are possible.

Due to (2) above most real numbers when represented by floating point number will have an error. This process of constructing floating point numbers from real number is called *rounding* or *rounding off*. The error is called *rounding error*. Let us say we want to use a floating point representation with $\mathtt{p} = 3$, $\mathtt{b} = 10$, $e_{\max} = 2$ and $e_{\min} = -2$. We shall use this representation to illustrate several important issues in floating point arithmatic. Now let us try to represent the number $\mathtt{0.3141}$ in this system. The floating point representation will be $\mathtt{3.14E-1}$. The rounding error is $\mathtt{0.0001}$. So the *relative* rounding error is $\frac{0.0001}{0.3141} \approx \mathtt{0.00031}$ A different way of writing the same error is to say that it is $\mathtt{0.1}$ *units in last place*, or in short `ulp`. In terms of this unit, the maximum rounding error is $\mathtt{0.5}$`ulp`.

**Theorem 1** The relative error and the `ulp` are related by

$$\frac{\mathtt{1}}{\mathtt{2}}\mathtt{b}^{-\mathtt{p}} \leq \frac{\mathtt{1}}{\mathtt{2}}\mathtt{ulp} \leq \frac{\mathtt{b}}{\mathtt{2}}\mathtt{b}^{-\mathtt{p}} \tag{3}$$

where we are using a system of floating point representation with base $\mathtt{b}$ and number of significant digits to be $\mathtt{p}$.

This implies that an error that is fixed in `ulp` can vary by a factor of $\mathtt{b}$. This factor is called *wobble*. Note that the maximum rounding error is always less a relative error of $\epsilon = \frac{\mathtt{b}}{\mathtt{2}}\mathtt{b}^{-\mathtt{p}}$. This number is called the *machine precision*. The relative error of a calculation is often given in units of $\epsilon$.

Now let us consider floating point arithmatic. The basic unit is addition. An addition in floating point is done by moving the decimal point till both the

numbers being added have the same $e$. Then they are added. For example let us add $3.75E2$ to $1.45E-1$. The addition will proceed as follows.

$$\begin{aligned} & 3.75E2 \\ +1.45E-1 = 0.00145E2 &\approx 0.00E2 \\ & 3.75E2 \end{aligned} \tag{4}$$

Whereas the actual number would be $3.75145$ which when rounded off to three significant digits will be exactly $3.75$.

Now consider the following subtraction. Find $x-y$ where $x=10.1$ and $y=9.95$. The actual answer is $0.15$. In floating point operations

$$\begin{aligned} x = 10.1 &\approx 1.01E1 \\ y = 9.95 &\approx 0.99E1 \\ x-y \approx 0.02 &= 2.0E-2 \end{aligned} \tag{5}$$

The error is $13\mathtt{ulp}$ and wrong in every digit !

**Theorem 2** For a floating point representation with base $b$ and number of significant digits $p$ the relative error can be as large as $b-1$.

Now let us look at a slightly improved method of subtraction. In this method the smaller number is rounded off with one more digit. And the result of the subtraction is rounded off the the usual number of digits. Let us see what happens in exactly the same problem:

$$\begin{aligned} x = 10.1 &\approx 1.01E1 \\ y = 9.95 &\approx 0.995E1 \\ x-y \approx 0.015 &= 1.5E-2 \end{aligned} \tag{6}$$

In this case the answer is exact ! The answer being exact is a lucky break, but in generaly by adding of one extra digit, called the guard digit, the accuracy in subtraction can be vastly improved.

**Theorem 3** If $x$ and $y$ are two floating-point numbers with base $b$ and number of significant digits $p$, and if the subtraction is done with a guard digit ($p+1$ digits) the th relative rounding error is less than $2\epsilon$.

In other words the guard digits is useful when subtracting two numbers which are almost equal to each other. But the guard digit unfortunately is not a magic bullet. If the two numbers themselves are results of other

floating point operations, then the guard digit will not help. To illustrate let us look at the roots of quadratic equation,

$$ax^2 + bx + c = 0 \tag{7}$$

The solutions of which is,

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a} \tag{8}$$

Let us take $a = 0.08$,$b = 2.75$ and $c = 23.43$. In this case we have

$$\begin{aligned} b^2 &= 7.5625 \\ ac &= 1.8744 \\ 4ac &= 7.4976 \\ b^2 - 4ac &= 0.0624 \\ \sqrt{b^2 - 4ac} &= 0.24979 \\ x_1, x_2 &= -2.500, -2.999. \end{aligned} \tag{9}$$

Let us see what we get by floating-point operations,

$$\begin{aligned} a &\approx 8.00\mathrm{E}-2 \\ b &\approx 2.75\mathrm{E}1 \\ c &\approx 2.34\mathrm{E}2 \\ b^2 &\approx 7.56\mathrm{E}1 \\ ac &\approx 1.87\mathrm{E}1 \\ 4ac &\approx 7.48\mathrm{E}1 \\ b^2 - 4ac &\approx 8.00\mathrm{E}-1 \\ \sqrt{b^2 - 4ac} &\approx 8.94\mathrm{E}-1 \\ x_1, x_2 &= 1.86\mathrm{E}1, 3.64\mathrm{E}1 \end{aligned} \tag{10}$$

So the error is $0.894 - 0.25 = 0.64$ which is $64\mathtt{ulp}$. Note that in this process the actual subtration of $4ac$ from $b^2$ (once each of them were used in their floating point representation) was exact. Using guard digits would not have improved this calculation at all.

Here the subtraction has merely brought fourth the error which were already inherent in the two terms that were subtracted. Problems such as

these must be dealt with in a case-by-case basis. In the present case there are several possible solutions. One of them is described in Ref. [8], but below I descibe my favourite which is due to Ref. [1].

The equation to solve is

$$0.08x^2 + 2.75x + 23.43 = 0 \tag{11}$$

Consider this equation as a linear equation with the quadratic term as a mere perturbation. But to proceed with this method let us first try an even simpler equation that can solved by hand in this technique.

$$x^2 - 10x + 1 = 0 \tag{12}$$

Which has the roots, (upto 4th decimal places) $9.8989$ and $0.1010$. Write this in the following form,

$$x = \frac{1}{10} + \frac{x^2}{10} \tag{13}$$

Now let us try iterative method. First ignore the second term on the right. This gives us our first approximation,

$$x_1 = 0.1 \tag{14}$$

Now the second one

$$x_2 = 0.1 + 0.1 * (0.1^1) \tag{15}$$

and so on

$$x_j = 0.1 + 0.1 * x_{j-1}^2 \tag{16}$$

Let us try to put this in a fortran code. To begin with let us forget about the beginning and end of the program but write its core first.

```
x = 0.1 + 0.1*x*x
```

This has to be iterated many times. First question is how many times ? As we dont know yet let us try it for $10$ times. In fortran such a repeatative operation is done by a `doloop` constructed as follows:

```
do i=1,10
  x = 0.1+0.1*x*x
enddo
```

We also want to know how $x$ changes after each iteration:

```
do i=1,10
  x = 0.1+0.1*x*x
  write(*,*)'i,x',i,x
enddo
```

But this also needs a first guess for x. Let that be 0

```
x=0.
do i=1,10
  x = 0.1+0.1*x*x
  write(*,*)'i,x',i,x
enddo
```

Now put a beginning and an end:

```
program root_quadratic
implicit none
integer :: i
real :: x
x=0.
do i=1,10
  x = 0.1+0.1*x*x
  write(*,*)'i,x',i,x
enddo
end program
```

Let us see what the output looks like:

```
 i,x           1  0.10000000
 i,x           2  0.10100000
 i,x           3  0.10102011
 i,x           4  0.10102051
 i,x           5  0.10102051
 i,x           6  0.10102051
 i,x           7  0.10102051
 i,x           8  0.10102051
 i,x           9  0.10102051
 i,x          10  0.10102051
```

Now note that if we want accuracy upto second place of decimal we may as well stop after the first iteration. Accuracy upto fourth place of decimal is

obtained after second iteration. So we have pretty good convergence. Now let us try to make our code more user-friendly. Let our equation be of the standard quadratic form. The we have to input three number $a$, $b$ and $c$. And the iteration procedure will go as

$$x_j = -\frac{c}{b} - \frac{a}{b}x_{j-1}^2 \tag{17}$$

With this addition our code looks like,

```
program root_quadratic
implicit none
integer :: i
real :: x
real :: a,b,c
a = 1.
b = -10.
c = 1.
x=0.
do i=1,10
  x = -(c/b)-(a/b)*x*x
  write(*,*)'i,x',i,x
enddo
end program
```

So far so good. Now let us try this on the quadratic we were working on.

$$0.08x^2 + 2.75x + 23.43 = 0 \tag{18}$$

which as $a = 0.08$, $b = 2.75$ and $c = 23.43$. This time we get the output

```
 1   -8.5200005
 2  -10.631721
 3  -11.808247
 4  -12.576282
 5  -13.121101
 6  -13.528387
 7  -13.844138
 8  -14.095569
 9  -14.299929
10  -14.468740
```

The iteration is no way as fast as the previous one. What is going on ? First let us try to see where the roots are. Let us calculate this function.

```
program root_quadratic
implicit none
integer, parameter :: N = 100
integer :: i
real :: x,xmin,xmax,dx
real :: a,b,c
! The coefficients of the quadratic equation.
a=0.08
b=2.75
c=23.43
! First plot the function.
! For this we need a minimum, a maximum and
! an interval.
xmin=-10.
xmax=10.
dx=(xmax-xmin)/float(N)
! Now generate the function for N points
! and write it to a file.
! First open a file
open(unit=10,file='quadratic')
do i=1,N
  x = xmin+dx*float(i-1)
  y = a*x*x + b*x + c
  write(10,*)x,y
enddo
close(10)
! Now go into iterative method of finding the
! root
x=0.
do i=1,10
  x = -(c/b)-(a/b)*x*x
  write(*,*)'i,x',i,x
enddo
end program
```

Now look at the tabulated data in the file 'quadratic' Notice these two points

```
 -18.900000      3.17995623E-02
 -18.799999      5.19947615E-03
 -18.699999     -1.98005978E-02
 -18.600000     -4.32002284E-02


 -15.700000     -2.58000903E-02
 -15.599999     -1.19998469E-03
 -15.499999      2.50001326E-02
 -15.400000      5.27999885E-02
```

This shows that the equation has one root between $-18.799999$ and $-18.699999$. And another root between $-15.599999$ and $-15.499999$. It is the smaller root to which our iterations were converting to. Let us check whether they were really converging by increasing the number of iterations from 10 to 100. The end of the iterations look like

```
i,x          86  -15.594783
 i,x          87  -15.594830
 i,x          88  -15.594872
 i,x          89  -15.594910
 i,x          90  -15.594944
 i,x          91  -15.594975
 i,x          92  -15.595004
 i,x          93  -15.595030
 i,x          94  -15.595054
 i,x          95  -15.595075
 i,x          96  -15.595094
 i,x          97  -15.595111
 i,x          98  -15.595127
 i,x          99  -15.595141
 i,x         100  -15.595155
```

Indeed we are converging, but the convergence was much slower than the first problem we solved. Can you improve the convergence ?

While we were solving this equation, we have actually found out a much more general universal method of root finding, The well known Newton-Raphson method. By just tabulating the function and looking at its values we have found the limits on both the roots. We can then just refine `dx` above

and find the root upto a higher accuracy. Let us write a code that does this. It will turn out that such a code will be applicable to higher order polynomial equations and even transcendental equations.

Before we end let us make a last note on rounding. We have noted that to subtract two floats which are close to each other it is useful to have a guard digit and then round off. The question then is how exactly do we round off. Till now we have been just ignoring the digits which are greater than $p$. But a better method is to take them into account and use them to change the $p$-th digit of our float. An obvious way is to round to $0$ if the $p+1$ this digit is less than $5$ and round to $1$ otherwise. Another school of thought says that half the time $5$ should be rounded up and rounded down other half. One way of doing this is by demanding that the rounded number will have its least significant digit to be even. Now let us look at an example

$$x = 1.00, y = -0.555 \tag{19}$$

Look at the following sequence

$$\begin{aligned} x_0 &= x_1, \\ x_1 &= (x_0 - y) + y, \\ x_2 &= (x_1 - y) + y \\ &\cdots \\ x_n &= (x_{n-1} - y) + y \end{aligned} \tag{20}$$

We get the following sequence by rounding up

$$\begin{aligned} x_0 &= 1.00 \\ x_1 &= (1.00 + 0.555) - 0.555 \approx 1.56 - 0.555 = 1.0005 \approx 1.01 \\ x_2 &= (1.01 + 0.555) - 0.555 \approx 1.57 - 0.555 = 1.015 \approx 1.02 \\ &\cdots \end{aligned} \tag{21}$$

This makes $x$ increase by $0.01$ at every iteration. Now let us round to to even. Then the sequence looks like

$$\begin{aligned} x_0 &= 1.00 \\ x_1 &= (1.00 + 0.555) - 0.555 \approx 1.56 - 0.555 = 1.0005 \approx 1.00 \\ &\cdots \end{aligned} \tag{22}$$

There is no problem at all.

## 5.4 Exercise

*Hint: Answer to all these exercises and in particular the proofs of the theorems are given in Ref. [5]. The point this exercise is to make you read this beautiful paper from which most of the discussion about floating point above is taken.*

1. Use the representation $\mathtt{p} = 3$, $\mathtt{b} = 2$, $e_{\max} = 2$ and $e_{\min} = -1$. How many (normalized) floats can you represent using this representation ? Write down all of them.

2. In the previous floating point represention what is the relative error corresponding to `0.5ulp` ?

3. Prove Theorem 1.

4. Prove Theorem 2

5. Prove Theorem 3

6. In fortran if you define a number to be `real` it implies that the computer uses a floating point representation where $\mathtt{b} = 2$, $\mathtt{p} = 23$, $e_{\max} = 126$ and $e_{\min} = -127$. What is the largest and the smallest number you can write in this system ? This number is a 32 bit or 4 byte number. Show how that is consistent with the value of $\mathtt{b}$,$\mathtt{p}$,$e_{\max}$ and $e_{\min}$ above.

7. Show by explicit example that floating point addition is not associative, i.e., $(\mathtt{a} + \mathtt{b}) + \mathtt{c}$ and $\mathtt{a} + (\mathtt{b} + \mathtt{c})$ are not equal.

8. Assume you are doing $\mathtt{N}$ floating point operations in your code. Each operation contributes maximum $\epsilon$ relative error. What would you expect to be the total maximum relative error, and why ?

## 5.5 References:

- Chapter one of Ref [1]
- Chapter on errors in Ref. [8]
- Chapter on root finding in Ref. [8]
- Ref. [5]

# 6 Lecture 2

Arrays and pointers – matrix multiplication – correct way of accessing an array in Fortran – how slow can it be if we access it the wrong way – profiling a fortran code – page faults, what do they mean – a function for matrix multiplication – multiplying matrices of any dimension – object oriented coding, what does it mean – operator overloading – how much memory does a subroutine takes

### 6.0.1 Iterative maps and roots

As an interesting aside consider the following equation.

$$\lambda x^2 + (1 - \lambda)x = 0 \tag{23}$$

where $0 \leq \lambda \leq 4$. And assume we want to solve this numerically. One of the ways could be to write the equation in the form used in previous lecture

$$x_{n+1} = \lambda x_n(1 - x_n) \tag{24}$$

In a more formal language the above equation is an iterative map. The "fixed point" of this map is the solution of Eq. 26. Let us try the iterations for various values of $\lambda$ and see what solutions we get.

### 6.0.2 Definition of arrays

Let us start with arrays. In fortran an one dimensional array `a` of `N` elements is defined by

```
integer, dimension(N) :: a
real, dimension(N) :: a
double precision, dimension(N) :: a
double complex, dimension(N) :: a
```

Here `N` must be an integer which has been given a value earlier. For example the following is not a valid syntax

```
integer :: N
real,dimension(N) :: a
N=10
```

This generates the following error in my compiler (gfortran)

```
real,dimension(N) :: a
               1
Error: Variable 'n' cannot appear in the expression at (1)

real,dimension(N) :: a
                     1
Error: The module or main program array 'a' at (1) must have constant shape
```

Note a few point about the errors.

1. We called our variable `N`, the compiler calls it `n`. Fortran make no distinction between upper and lower case, which is very unfortunate.

2. It does not allow you to define an array whose dimension is not known beforehand. So what if you try this

   ```
   integer :: N
   N=10
   real,dimension(N) :: a
   ```

   This generates the error :

   ```
   real,dimension(N) :: a
                         1
   Error: Unexpected data declaration statement at (1)
   ```

   Because all the definition statements must be before you start assigning numbers to variables. So the only way to do this is,

   ```
   integer,parameter :: N=10
   real,dimension(N) :: a
   ```

   This is where the construct `integer, parameter` becomes useful.

3. You can use dynamic array allocation in fortran as another solution. This we shall not use extensively to begin with, but to be pedantic let us mention the syntax here

```
integer :: N
real,allocatable,dimension(:) :: a
N=10
allocate(a(N))
```

Note that you have to specify at least the structure of `a`, i.e., whether it is one dimensional, two dimensional or multidimensional. Note further that every allocation should be accompanied by a `deallocate` command at the end of the program after `a` has been used.

```
deallocate(a)
```

You must not mention the size of `a` in the deallocation statement.

### 6.0.3 Arrays and pointers

Now that we know how to define the arrays let us find out how it actually works in a computer. In the language C there is a very close connection between an array and what is known as a *pointer*. I think is is a very useful thing to know although the use of pointers does not come naturally to fortran. Here I strongly recommend that you read chapter 5 of the book, The C programming language by Kernighan and Ritchie. Unfortunately, conceptually the most important thing in the relation between pointers and arrays cannot be demonstrated with usual fortran pointers because the address arithmatic is not supported. Here I shall demonstrate something below, which is not very useful for future coding, also may not be avialble for all fortran compilers, but conceptually meaningful. The following demonstrations work with the gfortran compiler but the idea is universal.

To begin, when we define a number in a computer it is reserved a space in memory, e.g., the statement

```
real :: a
```

reserves a space in the memory large enough to hold a 32-bit float. This space has an address which the computer knows. That is how it manipulates this number. In other words when we write

```
a = 10.54
```

The computer write the float 1.054E1 to the address it has previously reserved for a. A pointer is this address.

Let us see how it works,

```
program pointer
implicit none
real,target :: a=1.,b=2.,c=3.
```

In fortran if you want the pointer to a particular number you have to tell the compiler beforehand. That is why you define that number as a target.

Next define your pointer

```
real :: pointee
pointer(ptr,pointee)
```

This is done in two steps. First define a real, which in this case is pointee. And then by the next command define the ptr, pointee pair. The ptr is the address and the pointee is the number that is stored in this address.

We give a code below which we shall discuss in class

```
program pointer
implicit none
integer,parameter :: N=10
real,target :: a=1.,b=2.,c=3.
real,target,dimension(N) :: z
integer,target,dimension(N) :: zi
double precision,target,dimension(N) :: zd
real :: pointee
integer :: pointeei
double precision :: pointeed
integer :: i
pointer (ptr,pointee)
pointer (ptri,pointeei)
pointer (ptrd,pointeed)
!----------------------
  do i=1,N
     z(i) = 10.+float(i)
     zi(i) = 20+float(i)
     zd(i) = 30.+float(i)
```

```
  enddo
  ptr = loc(a)
! ptr points to a
! Hence pointee is now a
  write(*,*)'ptr,pointee,a',ptr,pointee,a
  b = pointee
  write(*,*) 'a,b',a,b
! b now has the value of a
  pointee = 0.
  write(*,*)'ptr,pointee,a',ptr,pointee,a
! a is now zero
  ptr = loc(z(1))
! ptr now points to the first element of z
  write(*,*) 'ptr,pointee,z(1)',ptr,pointee,z(1)
! Does incrementing pointers work ?
  ptr = loc(z(1))+1
  write(*,*) 'ptr,pointee,z(1)',ptr,pointee,z(1)
! apparently not !
! let us try the following
  ptr = loc(z(2))
  write(*,*) 'ptr,pointee,z(2)',ptr,pointee,z(2)
! let us then try adding 4
  ptr = loc(z(1))+4
  write(*,*) 'ptr,pointee,z(2)',ptr,pointee,z(2)
! so far so good now let us try integers
! let us now try to write out the whole array
  ptr = loc(z(1))
  do i=1,N
     ptr = loc(z(1))+(i-1)*4
     write(*,*) 'ptr,pointee,z(i),i',ptr,pointee,z(i),i
  enddo
! now let us increment ptr a bit further
   ptr = loc(z(1))+11*4
!   write(*,*) 'ptr,pointee,z(11)',ptr,pointee,z(11)
   write(*,*) 'ptr,pointee,z(11)',ptr,pointee
!  DANGER ! we get a number !!
! and a different number we we run the code twice !!
!  write(*,*) 'ptr,pointee,z(i),i',ptr,pointee,z(i),i
```

```
! Does it work for integers
  ptr = loc(zi(1))
  write(*,*) 'ptr,pointee,zi(1)',ptr,pointee,zi(1)
! Does not work because pointee is not an integer
! let us try the following
  ptri = loc(zi(1))
  write(*,*) 'ptri,pointeei,zi(1)',ptri,pointeei,zi(1)
! now try arithmatic on that
  do i=1,N
     ptri = loc(zi(1))+(i-1)*4
     write(*,*) 'ptri,pointeei,zi(i),i',ptri,pointeei,zi(i),i
  enddo
! perfectly ok.
! what happens with double precision ?
  ptrd = loc(zd(1))
  write(*,*) 'ptrd,pointeed,zd(1)',ptrd,pointeed,zd(1)
! now try arithmatic on that
  do i=1,N
     ptrd = loc(zd(1))+(i-1)*4
     write(*,*) 'ptrd,pointeed,zd(i),i',ptrd,pointeed,zd(i),i
  enddo
! Does not work, becasuse we are now in double precision
! so we need to do the following:
  ptrd = loc(zd(1))
  write(*,*) 'ptrd,pointeed,zd(1)',ptrd,pointeed,zd(1)
! now try arithmatic on that
  do i=1,N
     ptrd = loc(zd(1))+(i-1)*8
     write(*,*) 'ptrd,pointeed,zd(i),i',ptrd,pointeed,zd(i),i
  enddo
!--------------
end program
```

Now that we understand how pointers and arrays work together, let us look at two dimensional arrays. Here is a piece of code that defines and uses two dimensional arrays.

```
program array2d
```

```
implicit none
real :: pointee
pointer (ptr,pointee)
real,dimension(2,2) :: mat
integer :: i
!----------------------------
mat(1,1) = 1; mat(1,2) = 2
mat(2,1) = 3; mat(2,2) = 4
write(*,*)'mat(1,1)',mat(1,1),'mat(1,2)',mat(1,2)
write(*,*)'mat(2,1)',mat(2,1),'mat(2,2)',mat(2,2)
! point to the (1,1) element of the matrix
ptr = loc(mat(1,1))
write(*,*),'ptr,pointee,mat(1,1)',ptr,pointee,mat(1,1)
!now let us increment
do i=1,4
  ptr = loc(mat(1,1))+4*(i-1)
  write(*,*),'ptr,pointee',ptr,pointee
enddo
! Fortran stores arrays in a column major way.
end program array2d
```

### 6.0.4 Matrix multiplication

Now that we know how to define a matrix, let us step back and look at the usages of matrices. In physics one of the first things we do to a matrix is to multiply it with another matrix.

The simplest code would be the following,

```
program matrix
implicit none
real, dimension(2,2) :: a,b,c
a(1,1) = 1.; a(1,2) = 2;
a(2,1) = 3; a(2,2) = 4;
!
b(1,1) = 1.; b(1,2) = 0;
b(2,1) = 0; b(2,2) = 1;
! and if c is the result of their multiplication
c(1,1) = a(1,1)*b(1,1)+a(1,2)*b(2,1)
```

```
c(1,2) = a(1,1)*b(1,2)+a(1,2)*b(2,2)
c(2,1) = a(2,1)*b(1,1)+a(2,2)*b(2,1)
c(2,2) = a(2,1)*b(1,2)+a(2,2)*b(2,2)
end program
```

Clearly this involves 8 floating point multiplications. To estimate how computationally intensive the matrix multiplication is (sometimes called the computationally complexity of the problem) we consider only the number of multiplications not additions which are cheaper compared to the additions. Clearly to generate one element of the result matrix each element of a row of `a` must be multiplied with each element of a column of `b`. If we consider only square matrices then there will be N multiplications. And there are $N^2$ elements in the result matrix, hence the total number of multiplications are $N^3$. So the computational cost of the problem rises as a power law of the size of the matrices. We shall see several examples estimations of computational complexities in forthcoming lectures.

### 6.0.5 Subroutines

Let us now try to make our code more systematic. We want to do the multiplication inside a `subroutine`.

```
program matrix
implicit none
real, dimension(2,2) :: a,b,c
a(1,1) = 1.; a(1,2) = 2;
a(2,1) = 3; a(2,2) = 4;
!
b(1,1) = 1.; b(1,2) = 0;
b(2,1) = 0; b(2,2) = 1;
! and if c is the result of their multiplication
!c(1,1) = a(1,1)*b(1,1)+a(1,2)*b(2,1)
!c(1,2) = a(1,1)*b(1,2)+a(1,2)*b(2,2)
!c(2,1) = a(2,1)*b(1,1)+a(2,2)*b(2,1)
!c(2,2) = a(2,1)*b(1,2)+a(2,2)*b(2,2)
call matmul(a,b,c,2)
write(*,*)'a',a
write(*,*)'b',b
write(*,*)'c',c
```

```
end program
!************************
subroutine matmul(x,y,z,nsize)
implicit none
real,dimension(nsize,nsize),intent(in) :: x,y
real,dimension(nsize,nsize), intent(out) :: z
integer, intent(in) :: nsize
integer :: i,j,k

do i=1,nsize
  do j=1,nsize
    z(i,j) = 0.
    do k=1,nsize
      z(i,j) = z(i,j) + x(i,k)*y(k,j)
    enddo
  enddo
enddo
endsubroutine matmul
!************************
```

Note the following points :

1. `intent(in/out)`
2. nsize
3. what is the right way to access arrays ?

### 6.0.6 Optimised coding: introduction

To understand how to access arrays let us first try to *profile* our code. This is way to measure the performace of our code. Let us first add some timing to our code.

```
program matrix
implicit none
integer, parameter :: nmatrix=512
real, dimension(nmatrix,nmatrix) :: a,b,c
real :: time
integer :: nmax=1000000,imax
```

```
a=0.
b=0.
a(1,1) = 1.; a(1,2) = 2;
a(2,1) = 3; a(2,2) = 4;
!
b(1,1) = 1.; b(1,2) = 0;
b(2,1) = 0; b(2,2) = 1;
! and if c is the result of their multiplication
!c(1,1) = a(1,1)*b(1,1)+a(1,2)*b(2,1)
!c(1,2) = a(1,1)*b(1,2)+a(1,2)*b(2,2)
!c(2,1) = a(2,1)*b(1,1)+a(2,2)*b(2,1)
!c(2,2) = a(2,1)*b(1,2)+a(2,2)*b(2,2)
call cpu_time(time)
write(*,*)'time',time
do imax=1,nmax
   call matmul(a,b,c,2)
enddo
call cpu_time(time)
write(*,*)'time',time
!write(*,*)'a',a
!write(*,*)'b',b
!write(*,*)'c',c
end program
!*************************
subroutine matmul(x,y,z,nn)
implicit none
real,dimension(nn,nn),intent(in) :: x,y
real,dimension(nn,nn),intent(out) :: z
integer, intent(in) :: nn
integer :: i,j,k
do i=1,nn
  do j=1,nn
    z(i,j) = 0.
    do k=1,nn
      z(i,j) = z(i,j) + x(i,k)*y(k,j)
    enddo
  enddo
enddo
```

```
endsubroutine matmul
!*************************
```

This gives us the difference in time.

Instead of putting commands inside the code, we could find out about the timing from the unit `time` command which for me shows

```
> time ./a.out
 time  1.99999996E-02
 time  0.28001699
0.248u 0.044s 0:00.37 75.6%     0+0k 0+8io 0pf+0w
```

The fields are respectively,

1. 0.248 seconds of user CPU time
2. 0.044 seconds of system (kernel) time used on behalf of user
3. 0.37 seconds real time (wall clock time)
4. 75.6% total CPU time (user+system) during execution as a percentage of elaps
5. 15 Kbytes of shared memory usage and 3981 Kbytes of unshared data space
6. 0 block input operations and 8 block output operations
7. no page faults
8. no swaps

This gives some information. Some more in obtained by the following technique.

```
gfortran -pg <programname.f90>
gprof
```

## 6.1 Links

1. https://computing.llnl.gov/tutorials/performance_tools/#Overview
2. http://www.ncsu.edu/itd/hpc/Documents/sprofile.php

# 7 Lecture 3

matrix inversion – order of an algorithm – futility of using analytical neat formulas in a computer – matrix diagonalization – simpler methods of obtaining the largest eigenvalue The scanned copy of the lectures are at this location http://www.nordita.org/~dhruba/teach/scientific_compu/lecture3.pdf

References:

- Chapter on matrices in Ref. [8]
- Ref [1]

# 8 Lecture 4

Basic ideas of parallelisation – optimisations that the compiler does – memory and speed, one at the expense of other – evaluation of functions (sines, cosines, elliptic, Bessel) - examples of use of recursion relations – iterating in the correct direction – another example of specific tools for specific job

The scanned copy of the lectures are at this location `http://www.nordita.org/~dhruba/teach/scientific_compu/lecture4.pdf`

# 9 Lecture 5

Using existing tools – organising your code – makefile – C and fortran – calling C libraries from fortran – installing and linking libraries – static and dynamic linking – using LAPACK for matrix manipulations – using GSL – how to write your own libraries – fortran input and output – formatted input, namelists – formatted and unformatted output – hdf5, what it is all about.

Let us start by first looking at an earlier code that we wrote. This is the code the looks for the largest eigenvalue of a matrix.

```
!*********************************************
program eigvalue
implicit none
integer, parameter :: ndim=3
integer, parameter :: niter=200
real,dimension(ndim,ndim) :: mat
real, dimension (ndim) :: vec
integer :: iiter
!-----------------------------
mat(1,1) = 5.; mat(1,2) = -2.; mat(1,3) = 0.
mat(2,1) = -2.; mat(2,2) = -3; mat(2,3) = -1
mat(3,1) = 0.; mat(3,2) = -1.; mat(3,3) = 1.
vec(1)=1.;vec(2)=0.;vec(3)=0.;
```

```
write(*,*)'mat(1,1)',mat(1,1),'mat(1,2)',mat(1,2),'mat(1,3)',mat(1,3)
write(*,*)'mat(2,1)',mat(2,1),'mat(2,2)',mat(2,2),'mat(2,3)',mat(2,3)
write(*,*)'mat(3,1)',mat(3,1),'mat(3,2)',mat(3,2),'mat(3,3)',mat(3,3)
write(*,*)'before:vec',vec(1),vec(2),vec(3)
! multiply several times
do iiter=1,niter
  call mul_mat_vec(mat,vec,ndim)
  write(*,*)'iiter,vec',iiter,vec(1),vec(2),vec(3)
enddo
!done
end program eigvalue
!*******************************************************
subroutine mul_mat_vec(mij,a,k)
implicit none
real,dimension(k,k) :: mij
real,dimension(k) :: a
integer, intent(in) :: k
integer :: i,j
real,dimension(k) :: b
real :: norm
!-----------------------------------------
do i=1,k
  b(i) = sum(mij(i,:)*a(:))
enddo
norm = sum(b(:)*b(:))
write(*,*)'norm',norm
a=b/sqrt(norm)
norm = sum(a(:)*a(:))
write(*,*)'norm1',norm
!-----------------------------------------
endsubroutine mul_mat_vec
```

Last few lines of the output shows:

```
iiter,vec         198  0.97058779     -0.23497540      5.24008386E-02
 norm   30.076361
 norm1  0.99999994
 iiter,vec         199  0.97058779     -0.23497541      5.24008349E-02
 norm   30.076361
```

```
 norm1  0.99999994
 iiter,vec         200  0.97058779     -0.23497540      5.24008386E-02
```

Which shows that the iterations have converged. It further shows that the eigvenvalue corresponding to the largest eigenvector is $\sqrt{(30.07636)} \approx 5.48$.

Now let us try to write organise it in two files. First we make a file called `eig2.f90` and the `mul_mat_vec.f90`. The first one contains the `program` above and the second one contains the `subroutine`. Now compile by

```
gfortran eig2.f90
```

I get the following error:

```
/tmp/cc4puXLw.o: In function ‘MAIN__’:
eig2.f90:(.text+0x461): undefined reference to ‘mul_mat_vec_’
collect2: ld returned 1 exit status
```

This implies that the compiler cannot find the subroutine it looks for, namely `mul_mat_vec_`. This is obvious because the subroutine is now in a different file. What we need to do now is the following steps

```
gfortran -c eig2.f90
gfortran -c mul_mat_vec.f90
\begin{verbatim}
This process creates two ‘‘object’’ files
\begin{verbatim}
eig2.o mul_mat_vec.o
```

This two files can now be linked by

```
gfortran eig2.0 mul_mat_vec.o
```

And this creates the executable file `a.out`. All this process can be automatised by the unix command `make`. For this to work we must first create a `Makefile`. For this problem the simplest `Makefile` would look like:

```
default:
\tab gfortran -c eig2.f90
\tab gfortran -c mul_mat_vec.f90
\tab gfortran eig2.0 mul_mat_vec.o
```

Here the symbol

```
\tab
```

denotes the "tab" character in your keyboard. This is absolutely necessary in the syntax of the `Makefile`. Once you have this file you can use the command by simply saying `make` which will then compile your code by the steps above. A more systematic `Makefile` for the same problem is given below

```
include config.local
EXECUTABLES=eig.x
OBJ_FILE=  eig2.o mul_vec_mat.o
eig.x: $(OBJ_FILE)
        $(FC) $(OBJ_FILE)  -o eig.x
eig2.o: eig2.f90
        $(FC) -c eig2.f90
mul_vec_mat.o: mul_vec_mat.f90
        $(FC) -c mul_vec_mat.f90
#-----------------------------------
clean:
        rm -f *.x *.o a.out
```

Here the file `config.local` contains the following lines

```
FC = gfortran
FFLAGS = -O3
```

Now that we have seen the rudiments of how `make` works we have the background to use and appreciate external libraries. But before you proceed note a few points from above

- When you compiled `eig2.f90` without the $-$`c` option the compiler looked for a subroutine with the name `mul_mat_vec_` although in your code the same subroutine was called `mul_mat_vec`. Note the trainling underscore ! Fortran compilers, by default, puts a trailing underscore after its subroutines ! Again it varies from compiler to compiler. This can create untold misery which you try to interface your fortran code with C as we shall do below.

- To tell your compiler that the file you are compiling may not contain all the subroutines you need, but just to make the object file and not do the linking you have to use the option $-$`c`.

- Note the `tab` character in `Makefile`.

### 9.0.1 GSL

Let us begin with the GNU Scientific Library. This is a library that allows you to get values for many functions of mathematical physics. We have shown in earlier lectures how non-trivial it is to accurately calculate these mathematical functions. The GSL is hence of great potential use for us. But unfortunately it is in C. Let us try to see how we could use that.

As a first step download `gsl`. Read its manual. This of course takes some time ! For the very impatient, this is what you should do (But please do read the manual)

```
./configure --prefix=<DIRNAME>
make
make install
```

This installs the library in the directory given by $<$ `DIRNAME` $>$. Now let us try to first write a C code that uses this library. Let us say we are trying to calculate Bessel function typically denoted by $J_\ell(x)$ for a particular value of $\ell = 0$ and $x = 2.$.

```
#include <stdio.h>
#include <gsl/gsl_sf_bessel.h>
/* compile with gcc test_gsl.c
-lgsl -lgslcblas -L<DIRNAME>/lib -I<DIRNAME>/include */
     int main (void)
     {
       double x = 2.0;
       double y = gsl_sf_bessel_J0 (x);
       printf ("J0(%g) = %.18e\n", x, y);
       return 0;
     }
```

I am not going to explain the C syntax above. But most of it is obvious. Note that we need to include the header files in the second line. This code need to be compiled with the following command

```
gcc test_gsl.c -lgsl -lgslcblas -L<DIRNAME>/lib -I<DIRNAME>/include
```

The $-$`L` tells the compiler where to find the library. The $-$`I` tells the compiler where to find the hearder files it wants to include. In a very crude

representation this library is a collection of object files. So your command compilers the code you wrote and then links it with this already compiled subroutine. Now let us try to see how we could do this from fortran. In fortran the function call in C above should be made into a subroutine. What is done is to first write a piece of C code which interfaces between the fortran code and the C library. Such code are typically called "wrapper". Here is an example

```
/* c2f_gsl_sf_bessel_Jl.c */
/*-----------------------------------------*/
#include <gsl/gsl_sf_bessel.h>

void wrapper_sf_jl_(double* y, int*l, double* x){
   *y = gsl_sf_bessel_jl(*l,*x);
}
```

This calles the C function called `gsl_sf_bessel_jl` which already exists in the library. This piece of code also make available a subroutine called `wrapper_sf_jl_` to be called from fortran. The fortran code that uses this looks like

```
program test_gsl
      integer, parameter :: N = 20
      double precision :: x, y,deltax
      integer :: l,i,j
      l = 2
      deltax = 1.0d0/dfloat(N)
      do i=1,N
        x = 0.0d0 + (i-1)*deltax
        call wrapper_sf_jl(y,l,x)
        write(*,*) x,y
      enddo
end program test_gsl
```

To use them all together you need to give the following command

```
gcc -c c2f_gsl_sf_bessel_Jl.c -lgsl
    -lgslcblas -L<DIRNAME>/lib -I<DIRNAME>/include
```

This creates an object file called `c2f_gsl_sf_bessel_Jl.o`. Next compile your fortran code

```
gfortran -c test_gsl.f90
```

Next they are to be linked

```
gfortran test_gsl.o c2f_gsl_sf_bessel_Jl.o -lgsl
    -lgslcblas -L<DIRNAME>/lib
```

This creates the file `a.out` whose output look like:

```
0.0000000000000000         0.0000000000000000
  5.00000000000000028E-002  1.66636906828625437E-004
  0.10000000000000001       6.66190608445568766E-004
  0.15000000000000002       1.49759079189717937E-003
  0.20000000000000001       2.65905607952738590E-003
  0.25000000000000000       4.14809773936112517E-003
  0.30000000000000004       5.96152486862021932E-003
  0.35000000000000003       8.09545103937938867E-003
  0.40000000000000002       1.05453023922702661E-002
  0.45000000000000001       1.33058271610718183E-002
  0.50000000000000000       1.63711066079934124E-002
  0.55000000000000004       1.97345673464680987E-002
  0.60000000000000009       2.33889950253352297E-002
  0.65000000000000002       2.73265493454095607E-002
  0.70000000000000007       3.15387803766147279E-002
  0.75000000000000000       3.60166461411082356E-002
  0.80000000000000004       4.07505314251498246E-002
  0.85000000000000009       4.57302677798690840E-002
  0.90000000000000002       5.09451546685796841E-002
  0.95000000000000007       5.63839817158693565E-002
```

Obviously the process can be easily automatised by using a `Makefile`. I shall leave that for you to do.

Note finally that there can be further improvements to this piece of code. To make it run for any fortran compiler we need to take into account the fact that some compiler put one underscore, some put two and some none. This can be taken care in the following, possibly not unique, way (This piece of code is taken out of the pencil-code )

```
/* Wrapper C functions for the gsl library */
#include <gsl/gsl_sf_bessel.h>
#include <gsl/gsl_sf_legendre.h>
#include <math.h>

#include "headers_c.h"

void FTNIZE(sp_besselj_l)
     (REAL* y, FINT*l, REAL* x) {
   *y =  gsl_sf_bessel_jl(*l,*x);
}
/* ---------------------------------------- */
void FTNIZE(sp_bessely_l)
     (REAL *y, FINT*l, REAL* x) {
   *y =  gsl_sf_bessel_yl(*l,*x);
}
/* ---------------------------------------- */
void FTNIZE(sp_harm_real)
     (REAL *y, FINT *l, FINT *m, REAL *theta, REAL *phi) {
  REAL Plm;
  FINT ell = *l;
  FINT emm = *m;
  REAL fi = *phi;
  REAL x =  cos(*theta);
  if(emm<0){
    Plm = gsl_sf_legendre_sphPlm(ell,-emm,x);
    *y = pow(-1,emm)*Plm*cos(emm*fi);}
  else{
    Plm = gsl_sf_legendre_sphPlm(ell,emm,x);
    *y = (REAL)pow(-1,emm)*Plm*cos(emm*fi);}
}
/* ---------------------------------------- */
void FTNIZE(sp_harm_imag)
     (REAL *y, FINT *l, FINT *m, REAL *theta, REAL *phi) {
  REAL Plm;
  FINT ell = *l;
  FINT emm = *m;
  REAL fi = *phi;
```

```
  REAL x =  cos(*theta);
  if(emm<0){
    Plm = gsl_sf_legendre_sphPlm(ell,-emm,x);
    *y = pow(-1,emm)*Plm*sin(emm*fi);}
  else{
    Plm = gsl_sf_legendre_sphPlm(ell,emm,x);
    *y = (REAL)pow(-1,emm)*Plm*sin(emm*fi);}
}
```

The code above is the wrapper. Note that it includes a header file called `headers_c.h` which actually defines what `FTNIZE` means. The header file, also from pencil-code is given below:

```
/*                              headers_c.h
                               ------------
*/

/* $Id: article.tex,v 1.13 2010-12-01 08:22:18 dhruba Exp $
   Description:
     Common headers for all of our C files. Mostly required to get the
     number of underscores and single vs. double precision right.
*/

/* Choose single or double precision here (typically done from the Makefile) */
#ifdef DOUBLE_PRECISION
#  define REAL double
#  define FINT int /* should this be long int? */
#  define NBYTES 8
#  define GSL_PREC GSL_PREC_SINGLE
#else
#  define REAL float
#  define FINT int
#  define NBYTES 4
#  define GSL_PREC GSL_PREC_DOUBLE
#endif

/* Pick correct number of underscores here (2 for g95 without
   '-fno-second-underscore', 1 for most other compilers).
   Use the '-DFUNDERSC=1' option in the Makefile to set this.
```

```
*/
#if (FUNDERSC == 0)
#  define FTNIZE(name) name
#elif (FUNDERSC == 1)
#  define FTNIZE(name) name##_
#else
#  define FTNIZE(name) name##__
#endif

/* End of file */
```

To the best of my knowledge this part of coding in the pencil-code was done by Wolfgang Dobler. Which underscore is chosen is chosen by an option in the `Makefile` . The example of the `Makefile` is not given here, you should try to write it.

### 9.0.2 Exercise

1. Use the `gsl` library and the routine to find roots by bisection that you have written in Lecture 1 to solve the following problem.

   For $\ell = 10$, $r_{\min} = 0.7$ and $r_{\max} = 1.$, find $\alpha$ such that the following equations are solved,

   $$a_\ell j_\ell(\alpha r) + b_\ell n_\ell(\alpha r) = 0 \tag{25}$$

   for $r = r_{\min}, r_{\max}$. There are infinite number of solutions for $\alpha$, first the first three in increasing order.

   This equation arises while trying to solve the force-free equations in spherical coordinates. For further details see the following references: [3], [7]

# 10 Lecture 6

Integration – Monte-Carlo method

Let us try to improve the code for finding out the eigenvalue of a matrix, that we wrote earlier further. First we would like to input the matrix we want to work on. Let us first limit ourselves to the square matrices. Then we also need to tell the code what the size of the matrix is. The simplest to

do this use to use an input file and dynamic memory allocation. Let us first generalise the arrays we have used. We now rewrite the top of the code as:

```
integer :: ndim,niter
real,allocatable,dimension(:,:) :: mat
real, allocatable, dimension (:) :: vec
integer :: iiter
integer :: i,j
!-----------------------------
open(unit=1,file='mat.in',status='old')
read(1,*)
read(1,*) ndim,niter
allocate(mat(ndim,ndim))
allocate(vec(ndim))
read(1,*)
do j=1,ndim
  do i=1,ndim
    read(1,*) mat(i,j)
  enddo
enddo
read(1,*)
do i=1,ndim
  read(1,*) vec(i)
enddo
close(1)
```

Here we assume that the input file `mat.in` is written in the following way,

```
ndim niter
3      20
mat
5.
-2.
0.
-2.
-3
-1.
0.
-1
```

```
1.
vec
1.
0.
0.
```

Here we are not reading in the lines which contains statements like `mat` or `vec`. There is of course a better way to read in input parameters. This is done by using the fortran `namelist` concept. First, in the code we write

```
namelist /input/ &
  niter,ndim
```

This creates a namelist called `input`. The corresponding input file is read in the following manner:

```
open(unit=1,FILE='mat.in',FORM='formatted',STATUS='old')
read(1,NML=input, IOSTAT=ierr)
close(1)
```

The input file must now look as follows

```
&input
 ndim=3,niter=2
/
```

The advantage of this way of input is that

- The input data can come at any order
- it is clear which quantity is being set to exactly what value.

The matrix and the initial vector are now read from the following way

```
open(unit=2,file='matrix.dat',status='old')
do j=1,ndim
  do i=1,ndim
    read(2,*) mat(i,j)
    write(*,*) mat(i,j)
  enddo
enddo
close(2)
```

```
open(unit=3,file='inivec.dat',status='old')
do i=1,ndim
  read(3,*) vec(i)
enddo
close(3)
```

Note that our way of reading these assumes that the numbers are given after each newline(`\n` ) character. One can also put all of them on one line. Which case it can be read by

```
read(1,*)(vec(i),i=1,ndim)
```

### 10.0.3 Introduction to modules

Let us now take this chance to introduce a concept of fortran generally found very useful, the concept of `module`. A `module` is a collection. It could be a collection of variables which are then shared between different subroutines. In this case they are very much like `commonvariables` used in fortran 77. But that is hardly the beginning of the use of `modules`. We can also put subroutines themselves in a module. Then include the module in another code and use those subroutines. Let us begin with an illustration.

In the present case we would like to send our an error message if we do not find the correct input file. This could be done, for example, in the following way

```
open(unit=1,FILE='mat.in',FORM='formatted',STATUS='old')
read(1,NML=input, IOSTAT=ierr)
if(ierr/=0) call fatal_error('input data','problem with reading common input data'
close(1)
```

If the namelist is not read correctly it returns the integer `ierr` to be not equal to zero. We can then write a subroutine called `fatal_error` to be

```
  subroutine fatal_error(location,message)
!
    character(len=*) :: location
    character(len=*) :: message

    write(*,*)'FATAL ERROR: ',location,' ',message
    STOP 1
endsubroutine fatal_error
```

Obviously we could put such a subroutine in a different file. Compile it and link it. But we suspect such a subroutine may be useful in almost all code we write. And there could a whole collection of such subroutines. So we put them in a file called `technical.f90` and write a module in the following way

```
module technical
implicit none
public
integer,parameter :: labellen=25
double precision, parameter :: pi=3.14159265358979324D0
contains
!======================
  subroutine fatal_error(location,message)
!
    character(len=*) :: location
    character(len=*) :: message

    write(*,*)'FATAL ERROR: ',location,' ',message
    STOP 1
endsubroutine fatal_error
!=======================
  subroutine warning(location,message)
!
    character(len=*) :: location
    character(len=*) :: message

    write(*,*)'WARNING',location,message
    write(*,*)'will however proceed'
endsubroutine warning
!=========================

!==========================================
end module technical
```

Note that here we have declared the whole module to be `public` which implies that all the subroutines in this module are available to any other program or subroutine that uses this module. This is done by saying the following

```
!*********************************************
program eigvalue
use technical
implicit none
```

and the corresponding `Makefile` looks like

```
include config.local
EXECUTABLES=eig.x
OBJ_FILE=  eig4.o mul_vec_mat.o technical.o
eig.x: $(OBJ_FILE)
$(FC) $(OBJ_FILE)  -o eig.x
eig4.o: eig4.f90 technical.o
$(FC) -c eig4.f90
mul_vec_mat.o: mul_vec_mat.f90
$(FC) -c mul_vec_mat.f90
technical.o: technical.f90
$(FC) -c technical.f90
#-----------------------------------
clean:
rm -f *.x *.o a.out
```

### 10.0.4 Integration

Now that we have gone through the boring technical part, let us get back to the interesting algorithmic part. Let us consider integrating functions. But first a pause about the technical part. We can anticipate that the integration shall go by the following steps.

- We need to know the integrand and the limits of integration.
- We need to evaluate the function at different points in the interval
- From these function evaluations we find out the integral by some algorithm.

Of these the algorithm should be independent of the function we evaluate. So the implementation of this part can be done separately. Maybe in a module which shall then contain several such implementations for different algorithms. There must be driver program. And another piece of code that evaluates the function itself. We should keep these in mind while designing our code for integration.

### 10.0.5 Pieces of useful code

I have no idea how to integrate some simple ideas frequently used in fortran with the rest of the course. But as have already written some code I guess you shall be able to appreciate their use here.

So first a way to write a function with an array argument, where you do not need to transfer the dimension of the array (from pencil-code)

```
!***********************************************************************
    function step(x,x0,width)
!
!  Smooth unit step function centred at x0; implemented as tanh profile
!
!  23-jan-02/wolf: coded
!
      real, dimension(:) :: x
      real, dimension(size(x,1)) :: step
      real :: x0,width
!
      step = 0.5*(1+tanh((x-x0)/(width+tini)))
!
    endfunction step
!***********************************************************************
```

Next a piece of code to check if a file exists or not before opening it, again from the pencil-code

```
!***********************************************************************
    subroutine initialize_messages
!
! Set a flag if colored output has been requested.
! Also set a flad if fake_parallel_io is requested.
!
      inquire(FILE="COLOR", EXIST=ltermcap_color)
      inquire(FILE="FPIO", EXIST=lfake_parallel_io)
!
    endsubroutine initialize_messages
!***********************************************************************
```

To check if a float has become NAN (not a number)

```
if(ISNAN(x)then
```

The scanned lecture notes describing the algorithms can be found here `http://www.nordita.org/~dhruba/teach/scientific_compu/lecture6.pdf`.

# 11 Lecture 7 and 8

Integration of ODEs – different schemes – the classic Runge-Kutta – conservation laws, symplectic algorithms – subroutines – coupled ODE solver – dynamical memory allocation – stiff equations – exponential integrators

Most of this lecture is taken from the two main books we use, namely Ref. [1] and Ref. [8]. So instead of lecture notes I just put up a list of refenreces.

### 11.0.6 References

- For predictor-corrector method see chapter 5 Ref. [1]
- For Runge-Kutta methods and in particular the adaptive stepsize controlled RK4 see Ref. [8] chapter 15.
- For exponential integrators see Ref. [4]
- For conservative schemes see the appendix of Ref. [2]

# 12 Lecture 9

Interpolation – fitting a function

# 13 Lecture 10

FFT – basics of algorithm – ways of minor improvement – how are the data stored – memory and speed – installing and linking to FFT libraries, e.g., FFTW

# 14 Lecture 11

spectral solver for the Poisson equation in Cartesian coordinates – spectral solver for Burgers equation – coupling integration of ODEs with FFT – use of fortran modules

# 15 Lecture 12

PDE solvers – finite-difference – elliptic and hyperbolic PDEs – basic of hydrodynamic problems – advection test – phase errors – numerical dissipation – upwinding

# 16 Lecture 13

architecture dependent computation – GPUs – FFT and finite-difference in GPUs – modularity of code – adapting CPU codes to GPUs – a second look at parallelisation

# 17 Lecture 14

Random numbers – basic idea – generating random number arrording to a given distribution – properties of random number generators – algorithms for minimisation

# 18 List of problems

1. *Logistic map* Consider the following quadratic equation:

$$\lambda x^2 + (1 - \lambda)x = 0 \tag{26}$$

where $0 \leq \lambda \leq 4$. And assume we want to solve this numerically. One of the ways could be to write the equation in the form used in previous lecture

$$x_{n+1} = \lambda x_n (1 - x_n) \tag{27}$$

In a more formal language the above equation is an iterative map. The "fixed point" of this map is the solution of Eq. 27. Eq. 27 is the famous

logistic map. Iterate this map for about $1000$ transient iterations and then find out what it converges to. For small $\lambda$ you shall converge to $0$. As $\lambda$ increases the the converged quantity will be non-zero. For even higher $\lambda$ you shall see appearence of two-cycle, i.e, two solutions which map into each other under the action of the map. For even higher values of $\lambda$ higher order oscillation will appear, eventually for $\lambda = 4$ the map will turn chaotic.

2. Write a code to find roots by bisection and apply this to solve the quadratic equation discussed in Lecture 1. The code should look for desired accuracy and stop the iterations accordingly.

3. *Iterative root finding.* Use newton's method to find the root of the following equation:

$$x^3 = 1 \tag{28}$$

The solutions are the complex cube roots of unity, $1$, $\omega$ and $\omega^2$. First look at the lattice of points in the complex plane between $-1 \leq x \leq 1$ and $-1 \leq x \leq 1$. Start iterations from each of these points and find out to which of the three roots your iterations converge to. Color the initial points accordingly. The pattern that emerges is a fractal.

4. *Largest eigenvalue of a matrix* Write a code to find the largest eigenvalue of a matrix. The code should read in input parameters via namelist, use Makefile to compile and different subroutines should be in different files in a modular fashion.

5. *Roots of transcendental equations* Use the `gsl` library and the routine to find roots by bisection that you have written in Lecture 1 to solve the following problem.

For $\ell = 10$, $r_{\min} = 0.7$ and $r_{\max} = 1.$, find $\alpha$ such that the following equations are solved,

$$a_\ell j_\ell(\alpha r) + b_\ell n_\ell(\alpha r) = 0 \tag{29}$$

for $r = r_{\min}, r_{\max}$. There are infinite number of solutions for $\alpha$, first the first three in increasing order.

This equation arises while trying to solve the force-free equations in spherical coordinates. For further details see the following references: [3], [7]

6. *ode-solver* This is a collaborative project. Check out the ode-solver code from the googlecode svn server. The needs to have the following possibilities:

   - A model for single harmonic oscillator.
   - Runge-Kutta 2nd order integrator
   - Runge-Kutta-Fehlbarg integrator
   - A module for extrapolation using polynomials
   - A Bulrisch-Stoer solver.

   For the details of the solvers see Ref. [8]

7. *Fermi-Pasta-Ulam problem* Find out about the famous FPU problem from the scholarpedia (not wikipedia). Use the ode-solver above to solve for the FPU$\alpha$ model with $32$ coupled oscillators with periodic boundary conditions. Integrate till you see the first recurrence. Perform this integration with several integrators above and comment on their usage.

8. *Fourier Transform* Install FFTW in your computer. Then in real space start with the function

$$f(x) = \sin x + \sin 10x \tag{30}$$

   Use $64$ grid points in one dimensions. Use FFTW to perform FFT and to find out the spectra. See that your spectra matches with the input function.

9. *Random numbers* Write a code to use the random number generator by subtraction method as described as `ran3` of Ref. [8] Plot the probability distribution function of the random number you get and also calculate the correlation function. Finally plot in three dimension the time-series of random number shifted by one.

10. *Random numbers and logistic map* Use the logistic map with $\lambda = 4$ as a random number generator. Describe how good or bad this is as a random number generator as per the different tests described in class. Also compare the time of execution of this compared to `ran3`. The

different random number generators should be in the same module and a different master code should call them. The testing routines should be in a different modules.

# 19 Acknowledgements

In the designing and execuing this course I have received very constructive help from Nicolo, Matthius and Chi-Kwan. Nicolo has even attened some of the lectures, contributed some of the codes given above and make numerious helpful suggestions inside and outside the class.

Also I should acknowledge the two individuals who had taken time out of their busy schedules to teach me the basics of computer programming, when I was a fully computer illiterate student, namely (Sidhu) Siddhartha Shankar Ghosh and Chinmay Das. I must also thank my long time friend Sumit Sachdev first showed me how to use a computer.

# References

[1] F. Acton, Numerical methods that work, Harper and Row Publishers, New York, Evanston and London, 1970.

[2] J. Bowman, C. Doering, B. Eckhardt, J. Davoudi, M. Roberts, J. Schumacher, Link between dissipation, intermittency, and helicity in goy model revisited, Physica D : Nonlinear Phenomena 218 (2006) 1–10.

[3] S. Chandrasekhar, P. Kendall, On force-free magnetic fields, Astrophys. J. 126 (1957) 457–60.

[4] S. Cox, P. Matthews, Exponential time differencing for stiff systems, Journal of Computational Physics 176 (2002) 430–455.

[5] D. Goldberg, What every computer scientist should know about floating-point arithmetic, Computing Surveys.

[6] D. Knuth, The Art of Computer Programming, Addison-Wesley publishing company, Reading, Massachusetts.

[7] D. Mitra, R. Tavakol, A. Brandenburg, D. Moss, Turbulent dynamos in spherical shell segments of varying geometrical extent, Astrophy. Journal 697 (2009) 923.

[8] W. Press, B. Flannery, S. Teukolsky, W. Vetterling, Numerical Recipes in Fortran, Cambridge University Press, Cambridge, 1992.