I. INTRODUCTION

Welcome to Numerical Methods and Programing course! This is a 5 credit course and nothing is assumed! We will have about 3 theory classes, where some techniques will be introduced to you (this could either be a blackboard lecture or a presentation - depending on the material) and a hands-on session. We will have a lab every week between 2:00 pm and 5:00 pm in DKC. The class has been divided into two groups and one session will be on Tuesday (TA: Swetambar Das) and another will run on Thursdays (TA: Raj Manna). The labs will be graded and there will be assignments once in 15 days that will count towards your final grades.

There are excellent books available on computational physics: I will be mostly following:

1. Computational Physics Problem solving with computers by Rubin H. Landau and Manuel J. Paes
2. Computation Physics lecture notes by Hjorth-Jensen (on-line notes).

Other references include:

- Numerical Recipes in C (There is also a web version)
- Numerical Mathematical Analysis, J.B. Scarborough, John Hopkins
- Introductory Methods of Numerical Analysis, S.S. Sastry, Prentice Hall of India
- Numerical Methods for Engineering, S.C. Chapra and R.C. Canale, McGraw-Hill (1989).
- Electromagnetics and Calculation of Fields, Nathan P-Ida and J.P.A Bastos, Springer-Verlag (1992).
- M.K. Jain, S.R.K. Iyengar and R.K. Jain, Numerical Methods for Scientific and Engineering Computation, Wiley Eastern (1992).

The main goal of these lectures is to give a general idea of the tools required to solve problems on a computer. Why do we need computers in Physics? Very often we encounter problems in physics which cannot be done analytically. For example, we could come across a function, which is a solution of a differential equation that does not have a simple analytic form, and is therefore defined on a grid with N points i.e.. $f(x) \approx f_i$ defined on We would be required to either find its roots, integrate it over some interval, differentiate it etc. Sometimes we might have to solve equations which are very complex. Then the tool for handling such problems is a computer.

How does a computer work? The computer does what it is told! If these instructions are clear and precise, the result is what one would expect. It needs codes to start up, to use its hardware and in general for everything! The codes are nothing but a set of instructions to the computer and should be in a language that it can understand, i.e., *machine language*. The heart of a computer is the operating system and its nucleus is the kernel. The operating system (OS) is a set of instructions for the computer to use its hardware and also communicate with the user. The operating system has several shells and the user interacts with the computer/OS through these shells and most of the time, a user does not talk to the kernel. There are lots of OS available: UNIX, DOS, VMS etc. We of course will be using Linux which is based on UNIX and is a free OS.

Most of the programs we come across in Physics are not written in machine language, but rather a high-level language such as FORTRAN, C, C++ etc. These languages then need to be translated to the machine language and this translation is provided by a compiler. For this reason, they are also called compiled languages. There are also other softwares such as Mathematica, Maple, Matlab etc which are called Interpreted languages and do not require a compiler. In these softwares, the "instructions" written in some specific syntax is usually translated to the machine language line by line, whereas programs written in C, C++, FORTRAN are translated all at once using the compiler and are usually faster. In this course, for the most part we will use C.

The compiler translates a code written in a high-level 'human-readable' form into machine language by going through the code several times and understanding the logic. Therefore, it is essential before writing a code, or discussing a particular language syntax, to come up with a clear logic as far as possible. Here are some guidelines that help us attack a problem:

- Begin by understanding what needs to be done, what parameters are required etc.

- Plan how to execute the code even before you start typing in a language of your choice.
- Try a flow chart! It helps!

Once you have a clear logic in mind, we can proceed to writing out a code in any language of your choice. Here is an example: If we wish to calculate area of a circle how do we plan the code?

- We need the formula: πr^2 where r is the radius of the circle.
- We need to input r.
- We need to define π as accurately as possible.
- Write out the result.

Here is a sample C code:

```
1  /*FILE: area.c
2
3  Programer: Sunethra Ramanan suna@physics.iitm.ac.in
4
5  Date: 22 Dec 2011
6
7  Version: Original
8
9  Revision-History:
10
11 Comments:
12 22 Dec 2011: This code calculates the area of a circle. The input radius is obtained
13 as a user input.
14
15 NOTES:
16 Compile using gcc -o area area.c This generates an executable called area. Run this
17 executable using ./area at the prompt. You could also use a makefile.
18
19 Todo:
20 1. Hard code radius instead of an interactive input
21 2. Change the code such that r is now a grid starting from some r_min to r_max for
22 example 0. to 5. with a step size of 0.1. Generate the r grid. Using a for loop, loop
23 through all values of r and calculate the area. Print the output to screen.
24 3. Now declare a file pointer, open a file called area.dat and write the output to
25 file.
26 */
27
28 #include <stdio.h>
29 #include <math.h>
30 #include <stdlib.h>
31
32 /*****************************************************************************/
33 int
34 main(void)
35 {
36   const double pi = 4. * atan(1.);              /*defining pi*/
37   double r;                                     /*radius of the circle*/
38   double area;                                  /*stores the value of area*/
39
40   /*getting input*/
41   printf("Enter radius of the circle:");
42   scanf("%lf", &r);
43
44   /*calculate area*/
45   area = pi * r * r;
46
47   /*print output*/
```

```
48    printf("area of circle is:%f \n", area);
49
50    return(0);
51 }
```

The code inputs area from the user, calculates the area and prints out the output on screen. Notice that there are sufficient comments throughout the code. Although commenting a code might seem irrelevant for such simple examples, one needs to consider the fact that human memory is very fallible and it is easy to loose track of what a variable stores and as a result the code will rapidly become unusable by others and very soon by the programmer himself/herself. Hence good coding ethics involves commenting the code and also documenting the programmer details, revision history, version and the modifications.

Clearly the logic that is implemented above is just one way of doing it. There are several modifications that one can think of for this code. For example: One can avoid an interactive entry of the radius and code the value in, have an array for radii and calculate area as a function of the radius, push the part of the code that calculates the area into a sub-routine and the main code calls this sub-routine etc. The advantage with the last point is that the code is now modular and hence re-usable. We will see several instances where this re-usability will be a boon!

Before all the fancy sophistications of the code, it is important to check the code against an analytic result. What would be the simplest check that requires minimum effort from the user? If we set $r = 1$, then the area is π. This is of course the most basic check. Checking a code is mandatory and picking out simple enough examples is essential.

Here are some golden rules while writing a code:

1. Keep the logic very simple.
2. While defining variables use obvious labels. For example: if you need a variable for potential, you are better off calling it potential, pot, or v instead of x or y. This way by just looking at the variable it is obvious what it stands for.
3. Never use goto or computed goto in a code. The goto statements asks the code to go back to a particular line number and this can definitely change if you modify something before it. YOU WILL NEVER BE ABLE TO KEEP TRACK. Plus it is not a good thing for the compiler to go back and forth through goto's.
4. Always document a code! Never rely on your memory! From personal experience, I still use codes from my graduate school days and it is re-usable only because it is documented.
5. Make the code as modular as possible. The main code should ideally take inputs, perform some code specific stuff and any general calculation, such as integrations, matrix inversion etc, should be called, where each of these sub-routines are individual codes that are separately compiled and linked.
6. Once a code is written, do a dry run. You could use a de-bugger to step through the code. You will see examples in your lab sessions.

II. PRECISION AND ACCURACY

A. Number Representation

A computer stores everything as either 0 or 1, which is called a bit. A string of 8 bits is called a byte and 1KB = 1024 bytes. Although we have powerful computers, they are still finite when it comes to how much data it can store and process. Since a computer stores data in the form of bits, one can ask how many integers can be stored with N bits. The answer is 2^N integers. The sign of the integer usually occupies one bit, therefore integers on an N bit machine are in the range $[0, 2^{N-1}]$. Therefore a 32 bit machine can store a maximum value of $2^{31} = 2 \times 10^9$ integers and for unsigned integers $2^{32} \approx 4 \times 10^9$. This is indeed small when we know that we can have 10^{23} molecules in a room! *What are the largest integer limits for a 64 bit machine?*

Real numbers are usually represented in the following general way (in base 10 representation):

$$x_{10} = \pm r \times 10^n, \qquad 1/10 \leq r < 1 \tag{1}$$

where r is the mantissa and n is the exponent. For example a number -4.333 is represented as

$$-4.333 = (-1)^1 \times 0.4333 \times 10^1 \tag{2}$$

The analogous notation in base 2 is:

$$x_2 = (-1)^{\text{sign}} \times \text{mantissa}_2 \times 2^{\text{expfld}-\text{bias}} \tag{3}$$

The bias is introduced so that the exponent is always positive and we do not have to give a bit to store the sign. In base 2 the mantissa is written as:

$$\text{mantissa}_2 = \text{m}_1 \times 2^{-1} + \text{m}_2 \times 2^{-2} + \text{m}_3 \times 2^{-3} + \cdots \text{m}_\text{n} \times 2^{-\text{n}} \tag{4}$$

For single precision, n is usually 23 and m_i can be either 0 or 1. With the exponent of 8 bits and a bias of $127_{10} = 01111111_2$ and a bit for the sign, the computer uses 32 bits to store a number in single precision. Note that the mantissa in Eq. 4 is a choice and in fact in the IEEE standards, which is what most of the current computers follow, Eq. 4 becomes:

$$\text{mantissa}_2 = \text{m}_1 \times 2^{0} + \text{m}_2 \times 2^{-1} + \text{m}_3 \times 2^{-2} + \cdots \text{m}_\text{n} \times 2^{-\text{n}} \tag{5}$$

and this form is called the normalized form. The size of the number stored depends on the form for the mantissa. Let us look at an example. A base 10 number like 0.5 is represented as:

$$0.5 \equiv 0\ 0111\ 1111\ 1000\,0000\,0000\,0000\,0000\,000 \tag{6}$$

where the first place is the sign, the next eight digits are the exponent, which is the binary equivalent of 127_{10} and the remaining digits are for the base 2 mantissa. Note that 0.5 can be written in two ways: $(1 \times 2^{-1})2^0$ or $(1 \times 2^{-2})2^1$ where the terms in brackets is the mantissa. In order to have a unique representation, m_1 is normalized to 1.

Thefore for a 32 bit machine in single precision, which usually assigns 1 bit for the sign, 8 for the exponent and 23 for the mantissa, the largest number is:

$$0\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 111 \tag{7}$$

and the smallest is:

$$0\ 0000\ 0000\ 1000\,0000\,0000\,0000\,0000\,000 \tag{8}$$

In base 10 the largest real number is:

$$2^{128} = 3.4 \times 10^{38} \tag{9}$$

and the smallest number is:

$$2^{-128} = 2.9 \times 10^{-39} \tag{10}$$

Let us do an example to see how a computer stores a number (single precision). Consider the following binary sequence

$$1\ 0111\ 1101\ 1110\,1000\,0000\,0000\,0000\,000 \tag{11}$$

What number does this represent? We see that the first bit on the left is 1, which means it is a negative number. The mantissa is:

$$\text{mantissa}_2 = 1/2 + 1/2^2 + 1/2^3 + 0/2^4 + 1/2^5 + 0/2^6 + 0/2^7 + 0/2^8 + \cdots = 0.90625 \tag{12}$$

The exponent is expfld - bias. For a bias is 127_{10} and the decimal equivalent for the binary expfld is 125. Therefore the exponent is $125 - 127 = -2$. Therefore the base 10 number is -0.90625×10^{-2}.

The precision is determined by the smallest number in the mantissa. Therefore for the single precision number the precision is $2^{-23} \approx 10^{-7}$. Real numbers can also be declared as doubles in C. For doubles, 64 bits are used to represent a number as opposed to 32 bits, which is single precision. Using double precision increases the precision as

well as the range of numbers that can be stored. In this case, 1 bit is reserved for the sign, 11 for the exponent and 52 for the mantissa. The bias for double precision numbers is set to 1023_{10}. *What are the largest and smallest numbers in double precision? What is the precision? How dependent are these results on the way we define the mantissa? - We will see an illustration of the last question in the hands-on sessions.*

If we try to store and use a number larger than the largest number allowed for that machine, then we get an *overflow* error. If the number is smaller than the smallest number an *underflow* error occurs. Usually when there is an underflow, the answer will be incorrect as the number will be automatically set to zero. Overflows are disastrous!

While representing base 10 numbers in binary form it turns out that some numbers use only a finite number of bits, while some require infinite bits. Those that can be represented by finite number of bits are called machine numbers. Obviously there are only finite number of them! For example:

$$0.25_{10} = 0.01_2 \tag{13}$$

while

$$0.2_{10} = (0.00110011001100110011001101\cdots)_2 \tag{14}$$

Suppose there is only enough storage to keep only 8 digits, then 0.25 will be accurately stored, while for 0.2_{10}, only 0.00110011 is stored which is 0.199219 and therefore deviates from the actual number. The maximum deviation is called machine precision. Any number z is related to its computer representation z_c by,

$$z_c = z(1+\epsilon) \tag{15}$$

where $\epsilon \lesssim \epsilon_m$ and ϵ_m is the machine precision. Machine precision can be formally defined as that value ϵ such that

$$1 + \epsilon = 1 \tag{16}$$

in a given representation, which could be either a float or a double. Machine precision is not the smallest float or double that can be represented. It arises because of approximating the mantissa. Repeated operations such as multiplication, subtraction etc, accumulate the error depending on how the numbers are combined. While combining two numbers, if either or both is smaller than the machine precision, the answer will be obviously wrong (either equal to 0 if both are small numbers or equal to the larger number). We will explore these concepts a little more in detail in the hands-on sessions.

B. Errors

We have already seen that a computer can be limited in several ways although it is very powerful and can perform complex operations. The finiteness results in a limit on the largest and the smallest number a computer can store and also on the precision of the number stored. The machine precision, which is the smallest number a computer can safely store and use is around 10^{-6} or 10^{-7} for single precision real numbers, while for double precision, it is around 10^{-15}. For scientific computations, it is customary to use double precision.

While designing and executing a program, there are several errors that a programer should be aware of.

1. **Blunders**: These are obvious errors that arise due to mistakes while typing an equation, leaving out parts of syntax etc. If there is an error in the syntax, the compiler will catch it. For run time errors that could arise from typographical errors the programer should design obvious checks that would help catch these bugs. Use of a de-bugger such as gdb helps in catching these errors.

2. **Random errors**: These occur due to random events as the name suggests, such as cosmic rays, someone pulling out a plug, fluctuation in power etc. While a shorter calculation is usually reliable, codes that take over a day or week, should be re-run several times before the result can be trusted.

3. **Approximation errors**: This arises due to simplifications of the mathematics that would allow a problem to be solved on a computer. For example, infinite series sums are replaced by finite sums, variable functions by constants. For example:

$$\sum_0^\infty \frac{x^n}{n!} = e^x \tag{17}$$

$$\sum_{0}^{N} \frac{x^n}{n!} \backsimeq e^x + \varepsilon(x, N) \tag{18}$$

where $\varepsilon(x, N)$ is the absolute error. The more terms retained in the series minimizes this error. This error is also called truncation error. This error decreases if $N \gg x$. If $N \sim x$ then the error is large.

4. **Roundoff errors**: These arise due to the finiteness of the machine's storage. Only machine numbers get stored exactly. Sometimes in calculations, this error can build up significantly that we start getting garbage from the computer. For example, if we store a number $112233445566778899 = 1.1223344556677889 9 \times 10^{19}$, the exponent is stored completely as it is a small number. The mantissa is stored as two words, one has the most significant part and the second has the least significant part. In this example, the part 1.12233 is stored correctly while the second part is approximated. It is inevitable that errors get introduced in the least significant part of a number and it is important that we know how to keep this error under control.