COSMOS 2022, Lab 02

Even if you are quite familiar with C, the discussion of ‘molecular dynamics’ (MD) (coming up soon!) will provide a physics context which I think will be quite novel to you. Furthermore, MD is one of the most widely used and powerful techniques to simulate collections of *classical* particles. It’s a tool well worth mastering!

Immediately below is a summary list of Lab01 and then some follow-on projects/programs from Lab01. Details of the latter are on the following pages. The items denoted by the word ‘extra’ are some optional things. If you do not get to them, no problem, you will still understand everything we wanted you to learn. If you get to them, you’ll see a few additional interesting math and physics ideas.

Important! Please work with each other.

Lab01 revisited, part one! Type in, compile, and run the four codes provided to you: hello.c, add.c, count.c, decision.c. Study their structure and understand what they are doing and why. Try to think about every little thing: why are curly brackets where they are, etc?

Lab01 revisited, part two! Type in, compile, and run the three additional codes in Lab01 (page 09, division subtleties; pages 10-11 more complex control statements).

Lab01 revisited and extended: Write a set of codes to solve the quadratic equation. First ignore the possibility that $b^2 - 4ac < 0$ and that you will be asking the computer to take the square root of a negative number. Then, as a second iteration, put in an ‘if’ statement to handle the case when the discriminant is negative. Finally, as a third iteration, handle the case when the discriminant is zero. I provide some additional notes here, page 2.

[extra, page 3 below] If you have extra time, write a code which computes the point of intersection of two lines $y = ax + b$ and $y = cx + d$. This will actually connect to some of the things we willl learn/use in studying matrices!

[extra, pages 4-5 below] Although you may often use a calculator to do them, you have been carefully taught how to do the basic arithmetic operations addition, subtraction, multiplication, division on your own. Have you ever wondered how more complex, higher level arithmetic calculations are done? You can see in the codes in this optional exercise! (You’ll be learning something called ‘Newton’s method’ from calculus, although you do not need calculus at all to write the code!)

[extra, pages 6-8 below] Write some codes to explore the generation and use of random numbers. This is a *super important* topic in computational science! All of the incredibly powerful ‘Monte Carlo’ methods use them.

Write a (first) code to solve the quadratic equation. This is a super-charged version of add.c: You will need to declare, and scan in, three variables a, b, c and then implement the quadratic equation and output the two roots. Important note: 'math.h' tells the computer how to compute $\sqrt{x}$ using the command

sqrt(x)

but you also need a slightly more complex compilation command:

```
gcc -o quadratic.e quadratic.c -lm
```

the -lm links your code to the math library.

I am not clear why the designers of C decided including powerful math operations in math.h needs also to be supplemented by -lm.

For the moment, do not worry about taking the square root of a negative number.

Test your code for

$$y = 3x^2 - 20x - 7$$

That is, set $a = 3, b = -20, c = -7$. What are the expected results? (Factorize the polynomial!) Does your code give the expected results?

Run your quadratic equation code for $a = 3, b = 2, c = 7$. What happens?

Write a second quadratic equation code which prints a warning if $b^2 - 4ac < 0$. You will need a control statement to handle the two cases. $b^2 - 4ac \geq 0$, $b^2 - 4ac < 0$. You will need the ideas of decision1.c. **Please** do follow my suggestion of making a copy of your first, working, quadratic equation code before modifying!

Write a (third) code to solve the quadratic equation which deals with all three cases $b^2 - 4ac > 0$, $b^2 - 4ac = 0$, $b^2 - 4ac < 0$. That is, have the computer print an indication that there is only a single root when $b^2 - 4ac = 0$ and give its value. You will need an appropriate control statement. See the more complex decision codes for clues. Unlike version 2, which was really needed, version 3 is not so essential: you could in principle just count on the user to notice for herself that the two roots are identical when $b^2 - 4ac = 0$.

[extra]

If you have extra time, write a code which computes the point of intersection of two lines $y = ax + b$ and $y = cx + d$. First you need to do a little algebra to get the formula for the intersction point. Then you use that formula in your code (much like the quadratic equation code). There is a special case in this code too (which also involves an 'illegal' arithmetic operation)! Can you spot it? What is its geometric significance? Can you write a second code to handle it?

Actually, the condition for the special case is something we will come back to when we discuss matrices! It can be framed more formally as the vanishing of the determinant of the coefficient matrix of the set of linear equations you are trying to solve.

[extra]

Try this code out:

```
#include <stdio.h>
#include <math.h>
int main()
{
    int j;
    double x=1.,A;
    printf("Enter A:   ");
    scanf("%lf",&A);
    for(j=0; j<20; j=j+1)
    {
       x=x/2.+A/(2.*x);
       printf("%5i %12.8lf\n",j,x);
    }
    return 0;
}
```

• Notice you can assign values to variables when you declare them!

• Figure out what it does by running for several values like $A = 25, 64, 144, 0.01$. What does this code compute?

• It's interesting that the code does not get the result in a single step. Instead it cnverges to the right answer (very fast!). this is called solving a problem *iteratively*.

• Modify the code to use an arbitrary starting value for x (by scanning in x instead of setting it to $x = 1$). Does the code still work?

• What happens if you start with $x < 0$. Pretty crazy! Or is it?

[extra]

Figure out what this code does:

```
#include <stdio.h>
#include <math.h>
int main()
{
    int j;
    double x,A;
    printf("Enter initial x, A:   ");
    scanf("%lf %lf",&x,&A);
    for(j=0; j<20; j=j+1)
    {
       x=2.*x/3.+A/(3.*x*x);
       printf("%5i %12.8lf\n",j,x);
    }
    return 0;
}
```

• Can you generalize these two codes to compute fifth roots? You need to discern the appropriate pattern in the equations for updating x inside the loops.

[extra] If you have extra time, write these codes which examine the generation and use of random numbers.

First type in this code, compile, and run, to see how random numbers are generated.

```
#include <stdio.h>
#include <math.h>
#include <stdlib.h>
int main()
{
        double R;
        int i,N;
        unsigned int seed;
        printf("\nEnter number of iterations and seed\n");
        scanf("%i %u",&N,&seed);
        srand(seed);
        for(i=0;i<N;i=i+1)
        {
            R=(double)rand()/RAND_MAX;
            printf("%12.8lf \n",R);
        }
        return 0;
}
```

• This code generates and prints N ‘random’ numbers between 0 and 1.

• Run it and see how it works. Try, for example, $N = 20$ and seed=12345.

• What happens if you run the code again with the same $N = 20$ and seed=12345?

• What happens if you run the code again with the $N = 20$ and seed=66763?

• Can you think of advantages to being able to generate the same sequence of ‘random’ numbers?

• Computers do not generate truly random numbers. Rather they use an specific algorithm which generates numbers which ‘look random’: they are equally likely to be anywhere between 0 and 1 and the numbers which are generated are not related to the previous numbers in any way. Think about ways you might test whether the random numbers obey these criteria! We will explore some thoughts below.

Write a code to compute the first five 'moments' of the random numbers. Given N numbers $R_1, R_2, R_3, \cdots R_N$, their 'moments' are defined as:

$$\begin{aligned}
\text{moment1} &\equiv \frac{1}{N}\big(R_1 + R_2 + R_3 + \cdots R_N\big) \\
\text{moment2} &\equiv \frac{1}{N}\big(R_1^2 + R_2^2 + R_3^2 + \cdots R_N^2\big) \\
\text{moment3} &\equiv \frac{1}{N}\big(R_1^3 + R_2^3 + R_3^3 + \cdots R_N^3\big) \\
\text{moment4} &\equiv \frac{1}{N}\big(R_1^4 + R_2^4 + R_3^4 + \cdots R_N^4\big) \\
\text{moment5} &\equiv \frac{1}{N}\big(R_1^5 + R_2^5 + R_3^5 + \cdots R_N^5\big)
\end{aligned}$$

- You might find the 'pow' function useful: $x^5 = \text{pow}(x, 5)$.
- Try running with $N = 1000000$ (one million). Can you detect the pattern in the moments? Can you prove the pattern? The argument for the value of first moment is fairly simple, but what about the higher moments?

A super famous problem in mathematical physics is the 'random walk'. (Shirley actually alluded to it in her first presentation: Einstein's 'Brownian motion' problem!) The most simple version goes as follows:

Start a particle off at the origin $x = 0$. Generate a random number $0 < R < 1$.

If $R < 0.5$, move the particle to the left: $x = x - 1$.

If $R > 0.5$, move the particle to the right: $x = x + 1$.

Repeat N times. What is the final position?

Write a code to generate a random walk. Print out the final position.

Write a code to generate a bunch of random walks of length N steps. Figure out the average of the square of the distance from the origin. How does this grow with N? The answer is a very important result in the theory of 'diffusion'!

The random walk problem has close connections to the binomial distribution. Think about it and ask me, Eli, or Chunhan.

The random walk problem is the same as modeling the result of flipping a fair coin N times.

A ‘biased’ random walk has *unequal* probabilities of going left and right:

If $R < p$, move the particle to the left: $x = x - 1$.

If $R > p$, move the particle to the right: $x = x + 1$.

Biased random walks are good models of gambling!

Write a code that models going to Las Vegas with \$100 and making \$5 bets. If $p = 0.53$, what is the chance you will lose all your money if you place 200 bets? Can you write a code to figure that out? This sort of problem is sometimes called a walk with an ‘aborbing wall’: if the particle (your wallet) hits $x = -100$ the walk is over!

Testing random number generators

Random numbers are used *a lot* in computational science. In fact, every research project my group does uses them extensively. It’s super important to know we have a “good” random number generator. Did you think about how to test the one we are using? We will discuss some ideas in class.