COSMOS 2022, Computational Lab 01

Introductions

- Students- name, high school (location), favorite book, movie, or sport
- Eli
- Justine
- Morgan
- Richard

Course Overview

- Building an STM (Shirley will provide details)
- Computational Quantum Mechanics

Toolset:

Linux operating system

Programming in C (mostly) and python (a bit)

Matrices

Scientific Goals:

* Classical trajectories: 'Molecular dynamics' (mass on spring and 'Kepler problem')

  Mass on spring $\leftrightarrow$ LRC circuit (connect to Shirley)

  Kepler: there is already an open (unsolved) problem: Kirkwood gaps!
* Energy bands in one dimension, van Hove singularities (vHs).
* Energy bands in two dimensions, vHs and cuprate superconductors.
* Energy bands in two dimensions, graphene (semi-metals and Dirac electrons).
* Flat bands
* Spreading of a QM wave packet
* Tunneling (connection to STM work!)
* Perfect quantum state transfer

Why computation and why C?

Only possible to solve QM problems with 1-2 particles with pencil and paper!

How fast are computers at doing arithmetic?

With desktop computers can solve $\mathcal{O}(10^2)$ QM particles.

With supercomputers can solve $\mathcal{O}(10^3\text{-}10^4)$ QM particles.

Why matrices?

* Schroedinger Equation (need a lot of calculus and 'special functions')
* Feynman path integrals (also sophisticated mathematically; need Monte Carlo)
* Heisenberg matrix mechanics

Two key points:

- Modern research can be understood with relatively simple math and programming!!!
- Will explain everything from the beginning.

Some more talk

The properties of solids play a key role in technology. The names of early periods of human civilization reveal as much: the 'Stone Age', the 'Bronze Age', the 'Iron age' etc. Many key advances have come about through the manipulation of materials. Adding a few tenths of a percent of carbon to iron produces steel, which is a much stronger material than iron alone.

Why?

Starting about 100 years ago, people discovered what is inside materials: atoms composed of protons, neutrons, and electrons, and developed the theory which describes them: quantum mechanics. The last century has been marked by an explosion of new materials: semiconductors (as with steel, these are produced by doping Si with small numbers of impurities) make possible all of modern electronics, superconductors which are crucial to many medical imagining devices, graphene, carbon nanotubes, neodymium magnets in microphones, giant magnetoresistant materials which are now in all disk drives, etc etc.

New materials are almost always discovered experimentally. The reason is that while we do know the equations that govern the atoms in a solid, which in principle should allow us to compute their properties, in practice those equations are far too difficult to solve to be useful. This is illustrated by the challenge in going from the solvable quantum physics problem of the calculation of the energy levels of the hydrogen atom: one electron and one proton; to the *unsolved!* problem of the Helium atom (two electrons orbiting a nucleus). A 1 cm^3 solid contains roughly 10^{23} electrons, yet we cannot solve the quantum equations for two electrons!

You might think computers would help, and they do. Physicists and quantum chemists can now solve systems of between ten and a thousand electrons, depending on how strongly the electrons interact and how low the temperature is.

This cluster is about the first step in calculation of the energy levels of electrons which determine the properties of a material. We will focus on non-interacting electrons, a problem which, as we will see, is doable if we know a bit about the mathematics of matrices and a bit of coding. Because we will not include the interactions between electrons, we cannot get the final answer for the energy levels of a real solid. Nevertheless, we will see that taking the first step can explain a lot of interesting things. We'll focus on what solving this first step teaches us about some materials that are the focus of modern research: cuprate superconductors, graphene, Kagome metals, flat band systems, and so on. You'll learn a lot of advanced concepts in 'condensed matter physics': van Hove singularities, Fermi surface nesting, Dirac spectra, and so on.

If we are going to use computers, we should know a little bit about their capabilities.

How fast is a laptop computer?

10^9 'flops'.

What's a flop?

floating point operation per second

A computer can do one billion multiplications each second. That's impressive, but it is pretty easy to overwhelm a processor in doing advanced problems in computational science.

How fast is a the world's most powerful supercomputer?

10^{17} 'flops'.

A good habit to get into in coding is to count the number of operations. Then, if you are running on a laptop, you can divide by 10^9 to get the execution time in seconds!

We will learn to code in C. We'll begin with relatively simple codes which illustrate some high school mathematics and allow us to learn basic programming elements: input/output, loops, control statements, $\cdots$ in a familiar context. Then we'll do some further practice with a very powerful (but simple!) technique called 'molecular dynamics' (MD) to compute the tractories of *classical* objects. Some of our applications of MD will review sine and cosine functions and (surprise!) also give us a nice entryway into understanding electronic circuits via a very interesting mathematical analogy to the motion of a mass attached to a spring.

We will learn coding in C through doing a bunch of examples. You might think of this as learning a foreign language by listening to someone speak and imitating, rather than doing a formal study of vocabulary and grammar. We adopt this approach in the interest of proceeding rapidly, but also because it works! One thing that makes it effective is the relatively limited vocabulary of C and also the rigid syntax of coding languages.

Before showing our first four programs, let me review the steps in developing a code:

[1] Use an editor like "notepad" (or vscode) to type the instructions which constitute your code into a file. We will use C.

[2] "Compile" the code. You need a C compiler like gcc. This will produce an "executable": a version of your code you can run. Unless you tell the compiler otherwise, the executable will be called either "a.out" or "a.exe" depending on what operating system you use.

[3] Eliminate any errors found by the compiler, and recompile until none remain.

[4] Run the code.

C (and fortran, java, python, $\cdots$) are 'high level' languages which look a lot like English. A computer of course does not understand English. The compiler translates your 'almost English' C code into a 'low level' language that the computer can understand. Top-level computer scientists learn such 'machine language'.

Turn it over to Morgan and Justine to explain basics of linux and using an editor.

Then our first four programs.

Type in the four codes discussed on the following pages. Compile and run them.

The objectives here are to learn how to use an editor, how to compile, debug, and execute a code (all aspects of linux), and some C syntax.

hello.c

```
/* Print 'Hello, world' on the screen */
#include <stdio.h>
int main()
{
    printf("Hello, world\n");
}
```

One purpose of writing this code is to ensure you know how to use the editor, and that gcc can find the directory in which you are saving your codes. It does, however, introduce you to several simple things:

- In C, one can 'include' various useful 'libraries' at the top of the code. 'stdio' (standard input/output) defines commands like 'printf' (print to screen) and 'scanf' (scan (read) from screen).
- Comments can (and should!) be included in your code and are demarked by $/*$ and $*/$.
- You tell the compiler where (the main part of) your code begins and ends by with the header 'int main()' and encapsulating it with a pair of curly brackets { }. Curly brackets will also be used to mark the boundaries of other things like loops, conditional statements and user-defined functions.
- 'printf' is used to print to the screen. You can print text like 'Hello, world' or, as we will see, numerical values of variables you have computed.
- Most lines in a C code end with a semicolon ;
- $\backslash n$ tells the computer to go to a new line after printing.

add.c

```
/* This program reads in two numbers and adds them */
#include <stdio.h>
#include <math.h>
int main()
{
    double x,y,z;
    printf("Enter x:  ");
    scanf("%lf",&x);
    printf("Enter y:  ");
    scanf("%lf",&y);
    z=x+y;
    printf("x+y=%12.8lf \n",z);
}
```

We have learned a few more things:

• 'math.h' is a library containing 'advanced' arithmetic operations (square root). More below.

• You must 'declare' all variables you plan to use in a C code.

• There are different variable types:

'int' declares an integer.

'float' declares a real number of 32 bit precision (more on this later!)

'double' declares a real number of 64 bit precision (more on this later!)

Why might the requirement to declare variables be a good idea?

It is *very* easy to mis-type a variable name, and some typos can be deviously hard to discern e.g. 'x1' instead of 'xl'. With the declaration requirement, you will be alerted to typos when you compile. Without it, your code will look good to you and just give wrong results, which can often be super-challenging to track down. In general the mantra is that 'compile-time errors are much preferable to run-time errors.'

• 'scanf' is used to read numbers from the screen.

• When you scan in a variable, C needs to be reminded what type the variable is. 'lf' stands for 'long float' and is used for double precision variables. Note: This reminder is an annoying feature of C. I am unclear why the developers did not just adopt the convention that scanned variables always have the same type as their declaration. Such a convention is used in other languages. The way C does things opens the door to insidious errors (not caught in compilation, unfortunately!) where you mis-match the variable type, usually with horrendous consequences. Be aware of this as one thing to look out for when your codes give nonsense!

• When you read in a variable, you need to precede its name with an ampersand &

• You can format your output. The 12.8 tells the computer to assign 12 spaces for writing the variable, 8 of which will follow the decimal point. As with scanf, printf needs to be reminded of the variable type: 'lf' for long float (double).

count.c

```
/* this is a program to print out the first 10 non-negative even integers */
#include <stdio.h>
#include <math.h>
int main()
{
    int j;
    for(j=1; j<20; j=j+2)
    {
        printf("\n%i %i",j,j*j);
    }
}
```

- Here we encounter 'int' to declare an integer variable.
- We also encounter loops, probably the most important coding construct. Loops enable you to harnass the power of computers by doing operations over and over. The commands to be repeated are enclosed in curly brackets.
- The 'for' loop has a quite sensible syntax: You tell the computer the first value of the 'loop variable': $j = 1$. You tell the computer the final value of the 'loop variable': $j < 20$. You tell the computer how much to change the 'loop variable' each time the loop is executed: $j = j + 2$.
- In changing j we encounter the structure $j = j + 2$. This equation does not make sense in an algebra standpoint. It is best thought of as $j \leftarrow j + 2$: The computer takes the *current* value of j, adds 2 to it, and then *reassigns* j the new value.
- The 'for' loop is one of the C commands which is **NOT** followed by a semi-colon. If you do put a semicolon at the end of the line, the computer thinks this is the end of the loop, and the commands within the curly brackets will be executed only once. This is a very easy mistake to make, since you get in the habit of terminating all lines with semicolons. It is also hard to find since a semicolon is such a tiny symbol.
- Note: Unlike python, where indentation is a key part of the syntax, indentation in C is cosmetic: Use it to help yourself read your code more easily. In python, indentation plays a similar role to the curly brackets in C in demarking where things begin and end.

decision1.c

```
/* This program reads in two numbers and decides which is bigger */
#include <stdio.h>
#include <math.h>
int main()
{
    float x,y;
    printf("Enter x and y:  ");
    scanf("%f %f",&x,&y);
    if (x>y)
       printf("The first number was bigger\n");
       else
       printf("The second number was bigger\n");
}
```

- Here you are learning a 'control statement': Depending on the relative values of the variables x and y the code will do two distinct things.
- Notice you can scan in two variables in a single line!
- x and y are 'floats': i.e. 'single precision' (32 bit) variables.

Some final comments:

• It is *very useful* to give your codes names that suggest their purpose: hello.c, add.c, $\cdots$.

Why do I make this recommendation?

I find many students give useless names like 'program1.c'. Then, a few days into the class, they have absolutely zero idea what is in their many files.

• In fact, it is very useful also to name your executable in a way which connects it to the parent C code. You should be able to do this by expanding

gcc -o count.c

which produces the uselessly named a.out or a.exe, with

gcc -o count.e count.c

This tells the compiler to name the executable count.e. Depending on your operating system you can also do this in two steps:

gcc -o count.c

mv a.out count.c

• We will often write several versions of codes, adding functionality or improving in some way. I **strongly encourage** you to make a copy of any working code you have and modify the copy rather than the original.

Why do I make this recommendation?

It is possible to break codes when improving them. It is sometimes difficult to revert to the previous version (undo your changes). You are then left without even the original working code!

My own naming convention is to append a number to the code name to designate new versions. We will see this in a few minutes.

Please work with each other! Discuss problems you are having. Seek help. Provide your insight. Learning is a group effort.

We will walk around the room and provide help if needed.

Have fun!

Additional Activities

[01] There are a couple of ugly aspects to the output of 'count.c' Can you make the output look prettier? Note that in formatting an integer, you only need to specify the number of spaces to be allocated, since there is no decimal point.

[02] Type in the following code:

```
#include <stdio.h>
#include <math.h>
int main()
{
   float x;
   x=1./2.;
   printf(" x is  %7.4f\n",x);
   x=1/2;
   printf(" x is  %7.4f\n",x);
}
```

Can you explain what is happening? This exercise illustrates a fairly common coding bug and one which is often hard to find. If $x = 11$ and you have the line,
$y = 1/2 * x$;
What value will y have?

[03] 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?

Review the 'geometric interpretation' to the different cases of the quadratic formula.

A more complex control statement:

```
#include <stdio.h>
#include <math.h>
int main()
{
   double x;
   printf(" Enter x:  ");
   scanf("%lf",&x);
   if (x>0)
   {
      printf("x is positive\n");
   }
   else if (x==0)
   {
      printf("x is zero\n");
   }
   else
   {
      printf("x is negative\n");
   }
}
```

• Notice the double equals sign ==. Programming languages distinguish the single equals sign which assigns a value (e.g. $x = 2$) and the logical equals (e.g. $x == 2$) which asks a question: Does the variable x have the value 2?

Another code illustrates combining logical operators.

```
#include <stdio.h>
#include <math.h>
int main()
{
   float x,y;
   printf("Enter x and y  ");
   scanf("%f %f",&x,&y);
   if (x < y && x>0 ) {
      printf("x<y and x>0\n");
   }
   else
   {
      printf("either x was not less than y, or else x was not positive\n");
   }
}
```

- Two ampersands && demand that both the logical statements be true.

This code illustrates the 'not equals' logical statement:

```
#include <stdio.h>
int main()
{
   float x,y;
   printf("Enter x,y  ");
   scanf("%f %f",&x,&y);
   if (x!=y)
   { printf("The numbers are not equal \n"); }
   else
   { printf("The numbers are equal \n"); }
}
```

- not equals is denoted by $!=$ (This sort of makes sense.)