Comptuational physics: syllabus and goals for the course

The goals for this course are to develop in you the ability to write simple computer programs to solve physics problems that require computers (or which are much more easily solved by computers, in the case of things which you can do with paper and pencil, but which must be done in a repetitive manner). In order to do this, we will learn some basics of the Linux operating system and the C programming language.

We will aim to get you to the level where you don't need any more formal programming classes unless you want to use advanced techniques like parallel programming. This is not the same as saying that we will get you to the level where you are fully proficient in all types of programming in C. That takes more practice than would be reasonable to expect in a one semester class. One you reach a certain level, however, you should be comfortable with looking things up in a book or on google, and trying them out when you are confronted with situations where you don't know exactly what to do next.

The other main aim of the course is to teach you a bit of numerical analysis. Like with programming, in the course of a semester, we cannot do everything. Additionally, some types of numerical analysis involve having a higher level of mathematical background than what is in just the prerequisites for this course. Thus in many cases, I will mention some advanced topic by name in the notes, but I will not teach it in detail. My goal with this is just that if, several years from now, you are faced with a problem you don't know how to solve, but that sounds like a more challenging version of what you did in this course, there might be a chance that you can look up a buzzword in my lecture notes, and then investigate the topic yourself in detail.

We will cover the following topics and solve the following computational problems:

1	August 26-30	Why we do computational physics
2	September 2-6	Some basics of Linux and gnuplot
3	September 9-13	Basic ideas of programming, and initial programs in C
4	September 16-20	Pointers, functions, and arrays in C
5	September 23-27	Fourier analysis
6	September 30-October 4	Root finding
7	October 7-11	Integration I: Simple numerical integration schemes
8	October 14-18	Integration II: Simple differential equations
9	October 21-25	Integration III: Monte Carlo integration
10	October 28-November 1	Curve fitting
11	November 4-8	Make-up time
12	November 11-15	Final project lab sessions
13	November 18-22	Final project lab sessions
14	November 25-26	Final project lab session
15	December 2-5	Final project lab session

The use of week 11 as "make-up time" is to allow for the possibility that some topic may take a little bit longer to teach than I have anticipated. If all goes well, we'll cover another topic, with an unassessed assignment. For the first 10 weeks' lessons there will be assignments due one week after the last lecture on the topic. In each week, there will be one lecture and one lab session. I will be present at the lab sessions to give you help with the assignments. I strongly recommend that you make a try at the assignments before the lab session, so that then you can approach me quickly with well-targeted questions, after you have seen where you have gotten stuck.

Homework assignments

Week 1: None
Week 2: Making simple plots
Week 3: Simple algorithm development – long multiplication
Week 4: Making a calculator
Week 5: Find the pulsar; essay due
Week 6: Solving polynomials and other functions
Week 7: Solving integrals: I
Week 8: How far does a baseball fly?
Week 9: Hyperspheres
Week 10: Modeling of data
Week 11: No assignment
Week 12-15: Projects from the list of projects, to be given out soon

The final project will be due on December 9, which is during finals' week. There will be a range of choices for the final projects.

Comptuational physics: grading policies

For this course you will hand in 9 short, homework-length computer-work assignments, one short essay, and one project-length assignment.

In each week, there will be a main assignment and a challenge problem. To pass the course, you must complete all the main assignments successfully. This means getting the right answers to the problems using a computer program you have written yourself. I will not give any D's in the course, except possibly in the case where a student completes all the regular assignments and turns in a very poor final project; if you successfully complete all the assignments, you will get at least a C-, while if you do not successfully complete all the assignments, you will get an F or an incomplete, depending on the circumstances of why you have not completed the assignments.

You can then boost your grade from a C- for the bare minimum job all the way up to an A+ for a truly exceptional job by doing the following:

1. Turning a well-written pseudo-code (if you don't know what pseudo-code is yet, relax – you'll learn soon enough)
2. Writing a comprehensive and yet concise report on your work that you turn in with your code
3. Commenting your code properly
4. Writing good, structured code
5. Completing the challenge problem
6. Coming up with an idea of your own of how to extend the problem I have set you in a way that is not trivial

As one example of a way that you can always show some extra effort, you may try different approaches to writing the program that both work, and determine which one runs faster, or in some cases, which gives more accurate answers. The purposes of giving higher grades for doing the challenge problem or for coming up with an idea of your own are to encourage you to exercise some creativity by indulging your own sense of curiosity, and to get you in a position to learn more. At the same time, I don't want to have a course that students who make a good effort struggle to pass.

The purpose of these extensions beyond the basics are *not* just to give better grades to students who spend *more time* working on the project; they should be about going to deeper or broader levels of knowledge, and not just about doing some extra busy work. The challenge problems will be things that require you to give some more thought to the problem assigned. To earn an A+ on an assignment, you will have to come up with something on your own that is at least as difficult as the challenge problem and you will have to execute it properly (with a few exceptions, for cases where I assign a challenge problem that has some very difficult parts to it). If you receive an A+ in the course, it will demonstrate some curiosity and creativity, in a way that will make it

very easy for me to write you good letters of reference in the future, since I will have something specific to point to. Please bear that in mind if you find the course a bit easy (if, for example you already have some significant programming experience).

Applied Science and Technology requirement

This course counts for the core requirement for Applied Science and Technology, which will apply to those of you who started at Tech on or before Fall 2012. As a result, I need to give you an assignment related to ethics and technology. It will be to write a short essay about the free and open source software movements.

Collaboration in the course

I strongly encourage students to discuss programming and physics with one another as part of this course *but* you may not copy sections of one another's computer codes, and you should not be looking at another student's screen while you are working on your own code. You may look at another student's screen in order to get an idea of what they have done to make the program work, but then you should go back to your own program and make sure that you understand what to do before using the information you have just obtained. As a rule of thumb, make sure that you could re-do the assignment locked in a room by yourself.

Use of programs from the web and from books and journal articles

For the regular assignments for the course, you must write your entire programs yourself. You may use the programs I give you as a starting point, but you may not download programs from the web and modify them, and you may not copy code from your fellow students. These assignments are set up to drag you through the basics of scientific programming in C. If you use programs I give you, you should write your own pseudocode to explain how they work.

The final project is different – its goal is to ensure that you have some feel for how to do science with a computer program. Thus, once you get to the final project stage of the course, you should feel free to make use of functions that you take from any source you wish, but not to make use of full programs to do the job for you. If you use code from the web or a book, you must do the following: (1) add a comment to the function that states exactly where you found the code and (2) add comments throughout it that give me evidence that you understand how it works OR run a series of tests that show that it does what you think it is doing. The first is an issue of academic honesty, and while the second is merely good scientific practice. While it's important not to spend too much time re-inventing the wheel, it's also important to make sure that you're not using black boxes for vital things without being sure that they work.

I will even suggest that you may find the book *Numerical Recipes in C* helpful in particular. That book gives reasonably good explanations of a broad range of topics in numerical analysis, and it includes computer codes that work for many of these topics.[1] The book is available for free online. It represents

[1]The licensing rules for using the codes are fairly restrictive, so bear that in mind if you decide to use them. It turns out that the GNU Scientific Library has free version of almost all

its authors' best compromises between writing a book which is accessible and which advanced the state of numerical analysis in physics. There are better methods for a lot of things, but they are often hard to learn and implement and not always worth the effort – but you should bear in mind later in your career if you run into a tough problem that there may be better approaches in the applied and computational mathematics literature.

Late work and failed assignments

Work will be penalized by one full letter grade if it is up to one week late. It will be penalized by two letter grades if it is more than one week late. Any work that is turned in which successfully completes the assignment will be given a grade of at least a C-. If your first version of the assignment that you turn in is totally unsatisfactory, then you may re-submit after receiving some comments and get a C- grade on the assignment. All assignments must be received by three days after the end of lectures for the semester, or you will receive either an incomplete or an F for the course. You may choose to turn in all the assignments on that deadline if you are happy with getting a C- for a grade, and you are confident that nothing will have to be re-done, but I strongly recommend against that approach.

Since this is a relatively small class, I can be flexible about deadlines in the case of illness or serious issues in your personal life, but it will definitely be to your advantage to get caught up as soon as possible if you have such problems, since this is a course where we will continue to build on what we do throughout the course.

Distribution of points from assignments

The first 10 weeks of the course will each have assignments, except for the first week. One of these does not lend itself to being graded, and will just be a pass/fail assignment. I will then drop your two lowest marks on the eight assignments that are graded, as long as you pass all the assignments. In this way, you won't have to be overly stressed if you have midterms or other tests for some other course, and don't have time to complete the challenge problems every week. After that, these will all carry equal weight, and will, together make up 66% of your final grade. Then, there will be a final project which will be worth the remaining 34% of your grade. For the final project, because there will be no opportunity to re-submit, you will receive a passing grade for a good attempt at the problem that shows that you made some progress in solving it.

The essay will be graded pass-fail. That is, you must turn in an acceptable essay to pass the course, but beyond that, it will not affect your grade. It will be due toward the end of week 5, so that you will have some time to make revisions if I find your essay not to be acceptable.

(perhaps all) of the programs in Numerical Recipes, with the only licensing restriction being that you cannot then incorporate the routines into software that is sold without the right for people to copy and pass it on.

Comptuational physics

Motivations for learning computing as a physicist

There are a variety of reasons why anyone completing a physics degree should develop a certain level of skill at computer programming. The two main academic reasons are to be able to solve problems that simply cannot be solved with just paper and pencil, and to be able to work with the large data sets that are becoming increasingly more common in most areas of physics. The other major reason, of course, is that people who know physics and can do at least at little bit of computer programming are in high demand in industry.

Numerical solutions to physics problems Many of you may have already heard terms like *closed-form* or *analytic* solution in physics classes. A function is said to be written in closed form if it has been expressed in terms of a finite number of "well-known" operations that lend themselves to calculation. "Well-known" has different meanings, but generally means the functions that you learned before you took your first calculus class – addition, subtraction, multiplication, division, logarithms, roots, exponents, and trigonometric functions and their inverses. The term *analytic* also includes some solutions to different kinds of differential equations – these are functions which most mathematicians and physicists will have heard of, and which may pop up if you try to get an answer to something from e.g. Wolfram Alpha, but which will not be represented by a button on a $20 calculator.

Whenever a solution can be expressed in closed-form, this is ideal. One can then easily make plots of the functional form of the solution and develop an intuition about how properties change with respect to one another. For example, when I write that:

$$y = y_0 + v_{y,0}t - \frac{1}{2}gt^2 \tag{1}$$

$$x = x_0 + v_{x,0}t \tag{2}$$

many of you immediately realize that whatever object we're talking about follows a parabola in space, and if I told you to prove that, by finding an expression for y in terms of x, you could solve equation 2 for $t = \frac{x-x_0}{v_{x,0}}$, and substitute into equation 1. This is the power of a close-form solution – you can integrate it into other work very easily.

Historically, it was quite useful to get equations expressed in analytic form where closed-form solutions were not possible. The reason for this is that there were books that tabulated the values of special functions like the Bessel function or the error function (which is the integral of a Gaussian, something we will come back to later in the course). One could then compute the numerical values of analytic form solutions to problems much more readily than one can compute the value of a non-descript, unsolvable integral.

In recent years, numerical solutions have become much more prominent in physics. When we say we make a numerical solution to a problem, what we mean is that we go back to the origins of calculus. Instead of using some pre-set methods for solving integrals, we treat the integral as what it is – the area under

a curve. In the simplest approach, we put a bunch of little rectangles along the function and add up the areas of the rectangles. Often other shapes will give much better answers with fewer shapes, but we will get to that later in the course.

The use of numerical solutions also us to re-visit some "classic" physics problems. Before the advent of computers, physicists would often "cheat" to solve problems by making approximations. A famous example is the small angle approximation, in which $\sin\theta$ or $\tan\theta$ is set to be equal to θ for values of $\theta << 1$ (remember: the angle must be in radians!). We can also add in air resistance to our projectile problems, for example.

Now, an important point to bear in mind is this: being able to program a computer in order to do a numerical problem does not absolve you from having to learn math. There are several major reasons for this, even looking ahead as a professional physicist (i.e. apart from your grades in classes). First, remember that the closed form solutions to problems are always preferrable to any other class of solution, because it's so much easier to work with them. Second, in many cases, if you do some clever algebra before you write your computer program, you will be able to set up a program that takes many fewer cycles on the computer's processor to run. In some cases, you'll want to use these advantages to get better precision, and in others just to get the job done faster. Third, you always want to be able to come up with some quick analytic expression to check your computer code against. Coding mistakes are far more common than algebra mistakes.

There are, of course, cases where our computers are still not powerful enough to do everything we want to do, and so we have to fall back on classical approaches, or, in many cases, we need both to make approximations *and* use the computer to solve the approximate equations.

Big data

In many fields of science, data sets are growing in size literally expoentially, and the data sets in these fields are almost entirely digital. This has made it both necessarily, and straightforward to analyze the data on computers, often with considerably less human intervention than was the norm in the recent past.

Just to give some examples from my own field of astronomy: a large number of projects now focus on detecting transient sources of light in the sky. These may be supernova explosions, fast moving asteroids, outbursts from black holes sucking in gas from their surroundings or any number of other things.

As recently as the mid-1990's, surveys of large parts of the sky were still done using photographic plates – basically something like old-fashioned camera film, in which light interacts with silver causing chemical reactions to allow detections of light. After being exposed on a telescope, the plates then needed to be developed chemically.

Now, the data for projects done on optical telescopes are generally collected using charged coupled devices (CCDs). These are digital detectors, which immediately convert the photons they detect into electrons, which can then be counted to measure the brightnesses of the objects being examined. The original camera for the Hubble Space Telescope (which was launched in 1990) contained

an array of 4 800×800 pixel chips (i.e. about 2.5 Megapixels). More recent projects make use of cameras with many Gigapixels. The data flow rate from radio telescopes has increased similarly – the Very Large Array in New Mexico[1], for example, is in the process of having its data rate increase by a factor of about 20, as its electronics are being upgraded from the original 1970's era electronics to modern equipment.

At the present stage of transient search experiments, the detection of an event that is a transient must be made by some automated computer routine. There are simply too many images being taken every night, with too many stars in each image, to have a human look at the data and see which objects have varied. At the present time, the total number of *transients* is still small enough that humans can examine the data to look at them. However, as even bigger projects are built, this, too may become difficult, and people are working on writing computer programs that would classify the transients based on the data. Similar types of advances in data processing are going in most branches of physics and, indeed, branches of all sorts of other science and engineering fields.

What we will do in the course

So, having set some motivations out for doing a course in computational physics as part of a standard physics degree, let's set out the actual goals of the course.

Working in a Unix-based environment

There are a variety of operating systems for computers. Most of you are probably most familiar with using Windows or MacOS. MacOS is a Unix-based operating system, although many people who use it do not take advantage of its power.

The advantages of operating systems like Windows are also its disadvantages. Most things are done by point-and-click. This approach makes it easier for the user performing simple tasks to get started, but makes it more difficult to do sophisticated things. In Unix, one can open a "shell" window, and interact at a more basic level with files on the computer in a more straightforward manner. This makes Unix the operating system of choice for a lot of the things that most physicists do – if, for example, you want to perform the same analysis on a large number of different data files. It is also generally true that there is a better suite of free software available for Unix systems than for Windows. This difference ranges from being mildly important to absolutely crucial depending on one's field of endeavor.

Plotting data

Graphical representation of data is a vital tool to (1) understanding what's happening in physics and (2) conveying that to other people. Throughtout the course, you will plot your data. Part of your grade will be based on choosing good ways to present your results. For example, if you want to show that two quantities are related by a power law function, plotting the logarithms of

[1]This is the big Y-shaped set of 27 radio dishes that you may remember from the movie Contact.

both will generally be more illuminating than making linear plots of both. For exponential relations, you will want to take the log of only one quantity.

Understanding some basic numerical methods

Much of what you will do in this course will involve going back to the roots of calculus and just adding up numbers repeatedly in order to do integrals. However, there are schemes which can be far more efficient than others for doing this sort of things. There are also cases where setting up the equations incorrectly on the computer can lead to big errors, because the computer rounds numbers off.

Some beginning of understanding error analysis and model fitting

We will look at how computers can be used to estimate parameter values from experimental data. For example, suppose you tossed a ball in the air, and measured its height a large number of times. How would you know (1) what g is? And (2) what the initial velocity was with which you threw the ball? And how do you take into account the uncertainties on the measurements that you made? You can do this using a computer quite easily, and it is quite hard to do *in a precise, systematic manner* by hand.[2]

Understanding the process of writing a computer program – and that it's not that hard

This is probably the most important thing to do in this course. As computers have become more sophisticated, and software has become one of the biggest sectors of society, tech-savvy students are often much *more* intimidated by computer programming. The things that computers can do are much more sophisticated now than they were 20 years ago when I was taking courses like this one. Richard Feynman once said to the effect that people think physics is difficult because they try to start at the end – they try to understand quantum gravity without understanding how to solve pulley problems. Computer programming can appear difficult in the same manner.

[2]You can, of course, get a decent guess by hand.

Getting started with Linux

We will be using the Linux operating system in this course. The chief advantages of Linux are the ease with which you can do complicated things, and the large amount of high quality free software available. You will also probably find that certain things run a bit faster and more smoothly, and there is also the advantage that there are almost no viruses being written for Linux because it isn't used by enough people to justify the effort by the virus programmers[1]. The advantages relative to MacOS are quite a bit smaller than the advantages it has relative to Windows. Relative to MacOS, the main advantage is that the hardware on which it runs is cheaper.

Linux has a much wider range of command line tasks available than Windows does. It is also a little bit more straightforward to make what are called "scripts" in Linux than in Windows, although this is possible in both operating systems. We will not get into scripting in the course, but you may eventually want to do it. It is a handy way to do things like run the same program repeatedly with different parameter values, or run through a long list of tasks. Installations of Linux also have a window manager, and you can still do most of the same thing by pointing and clicking in Linux that you can do in Windows when pointing and clicking is easier.

The first thing to do with Linux is to open up a terminal window. [figure out how we do that in that particular flavor].

Next, let's also open a web browser. Go to my course web-site:

and download the file helloworld.cc

Now, let's move the file. In Linux, we use the term *directory* to refer to a place where a group of files are located. It's essentially the same as a folder on a Mac or a Windows machine, but we just use a different name for it. In the window managers, they'll be represented with icons that look like folders.

To change directories to the directory called *directory*, we type:
`cd` *directory*

Programming languages

First generation languages

Computers work with binary codes. Sending different sets of "bits" to a computer's central processing unit will give it different instructions about how to move other bits around in its memory or how to apply mathematical operations to the bits in its memory. The most "direct" way to program a computer, then, is to feed it a series of binary codes that tell it exactly what to do. This is how the earliest computers, which were mechanical devices, worked, and for some very specific applications, binary coding is still used. Machine language programming is called a "first generation programming language".

Second generation languages

The next step was to develop tools that allowed programmers to write programs in something resembling plain English, and to be able to translate those programs into machine code. These "second generation programming

[1] And, also, Linux users tend to be the most computer-savvy computer users.

languages", are often referred to as Assembly (since the code needed to be assembled into machine language). Assembly is generally machine-specific – that is, the same program will not work on a PC and a Mac. This has major disadvantages, of course, in that it requires the programmer to write separate programs for different types of computers. It has some major advantages for some very specific types of programming, in that it makes it possible for the programmer to do some non-standard things that will allow the program to run an the very fastest possible speed on that particular system. Getting the graphics processing to run smoothly in video games is one particular application of assembly language that still exists.

Compiled versus interpreted languages

Computers will only actually run machine language codes. There are two ways to generate these – through the use of compiled languages and interpreted languages. Compiling a computer program means taking the full program and converting it from a language similar to plain English into machine code. Interpreting a program means doing this line-by-line. The programming process is often a bit easier for an interpreted language – especially the process of going through and finding mistakes. The big disadvantage of interpreted languages is that they are often 30-100 times slower for the same task – for certain types of tasks they spend far more time translating the same line of your program into machine language than they do executing the tasks you're interested in.

Third generation languages

The major breakthrough was the development of third generation programming languages. These are languages that look something like plain English *and* can be transported from one operating system to another. They are written with the aim of being broadly applicable to a range of types of programs, rather than with a few specific applications in mind. In this course, we will use the C programming language. It is one of several third generation languages widely used in physics. The other two are Python and FORTRAN.

Why C?

The C language is sort of a compromise choice among the three languages. I wanted to give you some experience with a compiled language, so that if you eventually need to work on a problem that will be very computationally intensive, you can do it. I also wanted to give you experience to a language that is widely used outside academia, so that you will have better job prospects when you finish your degree if you decide not to go to graduate school.[2]

Python has some properties that allow it to do certain mathematical tasks

[2]FORTRAN has historically been a little better for certain kinds of computational projects than C because it gives the user a bit less freedom to manipulate the computer's memory – meaning that the compiler can assume certain things to be true and not use computational power checking that they are. There are now ways to force C compilers to assume the same things, removing much or maybe all of FORTRAN's advantages. FORTRAN is still a little easier to program for a lot of tasks, and there are some very valuable old programs in FORTRAN that are still modified and used. It's not dead yet. C has a much better system set up for user-interfaces, which is its main advantage – and it also caught on with a greater level of popularity in computer science departments in the 1980s and 1990s, which is why it's so widely used now.

in ways that are much easier to program than in C or FORTRAN (e.g. you can apply an operation to a vector in one line of code, instead of having to go through all the elements of the vector), but it is much slower once it's running than FORTRAN or C. Eventually if you learn both C and Python, you can decide which you will use based on how long you expect your program to spend running. Generally speaking, C or FORTRAN programs will run about 30-100 times faster than python programs for the same job. If the job will take an hour in python, then it will take a minute in FORTRAN or C. If you know you only need to run the program once, and it will take an extra half hour to write the program in FORTRAN or C than in Python, then you are probably better off with Python – you can take a coffee break or work on another project while it's running. On the other hand, if you will have to run that same program 100 times, it may be worth that extra half hour of your human time to write the program in FORTRAN or C.

Fourth generation languages

Fourth generation programming languages also exist. These are languages designed to be very easy to program, but for a fairly narrow range of tasks. Some examples include MATLAB, which is designed for engineering purposes, and R/S which are languages optimized for statistical analysis. Both of these can be of some value to physicists, but whether they are in a particular case depends on how big your computational project is and how well matched it is to the computer language you're using. For example, MATLAB has a scripting language, which is an interpreted computer language, and it has some compiled routines. If you need to do tasks that can be done mostly by MATLAB's compiled routines, then the relative ease in developing the program, coupled with what are sometimes better algorithms for doing the numerical tasks, may make it advantageous to use MATLAB. If you need to do most of your tasks under the MATLAB scripting language, which is interpreted, then you will probably find your programs running extremely slowly. The other disadvantage of many of the 4th generation languages is that they have expensive licensing agreements (although there are some open-source, free alternatives for the most prominent ones, which can provide nearly all of the same functionality).

Different levels of languages: an analogy to eating

I'll lay out a good, if slightly imperfect analogy for you.

1. First level languages are like cooking food that you grew yourself. It's pretty hard to do anything complicated, but you really know what's going on with everything.

2. Second level languages are like picking food, then cooking it. It's a lot easier than the former, and you get pretty much all the same choices.

3. Third level languages are like cooking for yourself after buying food at the grocery store. You might lose out a little bit in terms of just how well you can do what you want, but it's a lot easier process.

4. Fourth level languages are like eating at a restaurant. It's a lot less work for you, but it costs more. You also have a lot less control over how things

are done. Sometimes the expertise of the person doing the work for you is high enough that it allows you to try something you couldn't try otherwise. Sometimes it just takes longer and isn't any better. There are some things which are 4th level languages that are aimed at specific applications in the pure sciences which are free – but if it's something engineers would also be interested in using, it will often carry a charge. There are also knock-off versions of many of these languages which are free.

Which type of language you should use is something that you will eventually have to choose for yourself in a variety of different situations. For a lot of you, after you settle into a career, you'll find a fourth-level programming language like MATLAB to be what you use most of the time. That's fine, if the MATLAB routines exist for the purposes you have in mind. You'll generally find that you can get the programs written faster in MATLAB than in some other language.

I'm not going to teach you MATLAB, however, for a few reasons. First, you might end up going to a job or to a graduate school which doesn't have a MATLAB license. Second, you might end up needing to write programs to do the kinds of things MATLAB isn't optimized for. And third, it's generally easier to start with more a more general programming language and adapt to a specific one than the other way around.

At the same time, I'm not going to teach you assembly language (i.e. a second level language). The uses of assembly language these days are mostly limited to optmizing codes for a very small range of specific applications. If you think you might like to work in the video game industry, you should look into that, but it's not really something that is of use to many physicists – I've never used it myself.

C vs C++

We will use the C++ compiler g++ for turning our programs into machine code. The C++ compiler is "backward-compatible" with C programs – i.e. it can compile programs written in standard C. It also has structures set up to allow the use of a technique called object-oriented programming. The object oriented approach can sometimes be better for very large group projects, but is generally a bit more burdensome for doing small projects on your own or in small groups. I will not teach any object-oriented techniques to you, so effectively I will be teaching you to program in C, even though we will be using the C++ compilers.

An aside: LaTeX

Many of you may be impressed with how nice the equations look in the notes I will send around to you. I am using a software package called LaTeX for writing them. It is the standard way to write scientific papers for most physicists, astronomers, mathematicians and computer scientists. For papers with a lot of math in them, it can be a lot easier to work with than the Microsoft equation editor. I am not going to teach LaTeX in class, because it's one of these things that you can learn by doing once you know what it is, and it's not really computational physics, but if you want to see a few sample copies of the original files of my notes so that you can start teaching it to yourself, please feel free to

ask me for them. I didn't start learning it myself until grad school, so don't feel like this is something you really need to do.

How to approach a computational problem

A lot of people find computer programming difficult, especially when they first get started with it. Sometimes the problems are problems specifically related to writing programs – difficulties in dealing with the exact syntax the computer demands from the programmer. A lot of the time, though, this is not the case, and the problem is more basic. One major source of difficulty that beginning programmers will often have is that they try to write the computer program before they have properly thought about what they are trying to do. This sort of impatience is human nature, especially if you're learning to program in a computer lab, while sitting in front of the computer terminal. It's also very bad practice and can lead to a lot of frustration.

As you first get started with programming, you should follow a few steps before you start typing on the computer. As you become more experienced, you may decide to take a more relaxed approach, and for simple tasks, that more relaxed approach may lead to your getting the job done faster. For complicated tasks, though, you should always follow certain elements of "good practice".

Subroutines and pseudocode

We often refer to a finished computer program as a piece of "code." The essence of a computer program is that it is a means of encoding an algorithm – a step-by-step procedure for performing some function, usually a calculation – into a language that a computer can "understand."[1] Very often, what will happen for beginning (and even experience) programmmers is that the computer code is a correct implementation of an incorrect algorithm.

We need to remember that, without programming, computers can do only a small handful of things – some basic input and output tasks, some shifting around of data from one location in the computer's memory to another, and some basic arithmetic. Once we remember that the computer can do only these things, but can do them perfectly,[2] we understand the level of detail in which we need to develop our algorithms in order to get them to work out.

What you need to do, then is to take the larger task you wish to accomplish, and break it down into a set of smaller tasks. Eventually you will wish to get to tasks so small that the computer can understand them. You will have the most success in achieving your goals if you do this *before* you sit down at the computer and start typing in a program.

There are a few key "buzzwords" for this sort of algorithm development – *structured code*, *subroutines*, and *pseudocode*.

First, let's discuss "structured code". The nature of a structured code is that it is built up from a main body with a set of subprograms, called subroutines. The main body exists almost exclusively for calling the subroutines to be run – and sometimes for determining whether enough of the subroutines have been run. For very simple programs, one can get away with doing a bit more in the

[1] As we will discuss later, we actually need to take our codes written in a "high level" programming language and convert them into machine language before the computer can really "understand" them.

[2] We will discuss round-off errors in computer arithmetic later in the course – computers actually don't do certain types of math perfectly

main body, but as programs get more complicated, the main body really needs to be kept as simple as possible, with the complexity moved to the subroutines.

The modularity of structured codes leads to several key advantages. During the programming process, it becomes possible to split tasks up among several programmers, if a code is especially complicated. Additionally, certain subroutines are useful for a variety of purposes, and one can often find these subroutines already written (e.g. the book *Numerical Recipes* contains a lot of computer codes that can be applied to various tasks, mostly relevant to scientific computing). There is also a package called the GNU Scientific Library that contains libraries of routines written for use with C/C++ programs.

Consider the program `compute_sines.cc`. You can look at this code and see immediately that it first calls a subroutine `readinvalues` and then calls a subroutine `printoutsines`. You do not need to be an expert in computer programming to figure out what the basics of this program are, especially since I have put comments in the subroutines. You can also see that if we wanted to allow the user to decide whether to take a sine or a cosine wave, we could have put in another subroutine, almost like the `printoutsines` routine, but with a cosine function in it and called it `printoutcosines`[3]. We then could have included sine or cosine as an option in the routine `readinvalues`, and a "control structure" in the main body to use the user's choice to decide which of the subroutines. The main body of the code would get only slightly longer, even as the code got much longer.

The other main advantage is that two programmers could work simultaneously on this program – one could write the input subroutine and one could write the output subroutine, and then the full program could be put together later on, and a third programmer could be added to write the cosine routine, if that were deemed valuable. This program, and indeed, most of what you are likely to do as an undergraduate, is simple enough that the amount of time spent discussing exactly how to separate the work and how to make sure that the pieces fit together at the end might not be justified by the additional in person-power to the task[4] – but you can at least see how a nice structured program makes this much easier to do.

Pseudocode

So: how do we get to the point of writing a nice structured program, and coding it in correctly? This, plus some basic understanding of some numerical methods, is the whole point of our course. A lot of it will just take practice. People who spend hours and hours writing computer programs tend to become good at it. So, think about this once in a while when you are doing your homework for other classes, or are working on lab reports, or anything else that requires some mathematical analysis – and maybe perform a computational

[3] We could call it `Fred` too, if we wanted to, but we generally like to give subroutines names that help describe what they do. But the program would still work with `Fred`

[4] There is an influential book in software engineering called *The Mythical Man-Month* which suggests that sofware projects should generally be done with the fewest people possible because having too many people working on the project means that too much time is spent communicating about the project and not enough time is spent doing it.

solution to something, or use a computer program to model your data.

Let's also remember the words of NFL Hall of Famer Vince Lombardi, who said, "Practice does not make perfect. Only perfect practice makes perfect." There are some ways to approach writing a computer program that will allow you to do a better job, right from the start.

The most important of these is writing a "pseudocode" before you start programming. A pseudocode is a description of what your program will do that is written out in plain English (with some mathematical notation where that's easier to understand than plain English). A good pseudocoding approach will start from a top-down approach – give a list of fairly vague tasks that are "black boxes". Each of those black boxes is then specified further later on. The top page then becomes the main body of the program, and each of the black boxes becomes a subroutine. Some of the subroutines may need further subroutines, of course.

In class, we will discuss the example of writing a pseudocode to make a peanut butter and jelly sandwich.

Debugging

Writing computer programs is not easy for most people at first, but it's something that people generally become better at as they get more experience. A lot of what you have to do is "trial and error" type work, but there are some strategies that can make it go more quickly and less painfully.

There are two results of making a mistake in writing your program. The first result is that the program fails to be compiled. This is usually the easy kind of problem to fix, because the compiler usually gives you some advice in the form of an error message, telling you what went wrong and in which line of the program it went wrong. Often, in C, the problem will be as simple as a missing semi-colon. Occasionally, the real problem will be elsewhere in the code and will show up when it causes a fatal error somewhere else.

The other problem which can arise, which is harder to fix, is when your program compiles and runs, but gives the wrong answer. In this case, there are three main classes of mistakes:

1. Mistakes in designing the algorithm – that is, your plan for how the computer should follow simple steps and get the answer you're looking for is incorrect.

2. Mistakes in coding the algorithm into the computer – that is, the idea you have about how to break the problem up into simple steps is correct, but you have made some error in terms of translating that idea from pseudocode into actual computer code.

3. Typographical errors – these can include typographical errors in writing the computer code, and also sometimes errors in the input data set you send the computer.

When in doubt, print it out!

When you are trying to figure out where you have made a mistake in your program, you should print out the results of the program to the screen, or to

a file that you can look at, or even send to a printer to get a paper copy. Not only that, you should print out a lot of things that you wouldn't normally need to see – e.g. the intermediate results of calculations; lines that say you entered a loop, and that you exited a loop; etc. Print out as much stuff as possible, and then follow the problem through a simple example that you can calculate by hand. Find where what you're doing by hand differs from what the computer is giving you. If at all possible, print out so much stuff that you locate the mistake to within a few lines of computer code.

Getting started

We will now write the first C program that most people write, "Hello World", called `helloworld.cc` in the course web page.

```
/* Hello World program */

#include<stdio.h>

main()
{
printf("Hello World\n");

}
```

There are several key syntax things to notice about this program. First, I have set apart the first line as a comment that tells me something about what the program does. Second, I have included the standard input-output library. Third, I have a section starting with `main()` – this is the main body of the program. Fourth, I have printed something out within the main body of the program. Note that there is a `\n` in the `printf` statement. That signifies to go to a new lines afterwards.

Next, let's try to move to a structured programming approach. Let's make that printout go inside a function.

```
/* Hello World program -- with a function */

#include<stdio.h>

void helloworld()
{
printf("Hello World\n");
return;
}


main()
{
helloworld();
}
```

Now we have `void helloworld()` before the main body of the program. This is a function. The function is of "type" void.

An aside: data types

Functions and variables have "types" in C. The basic types are `char, int, float`, and `double`, for characters (i.e. things which may not be numbers), integers, floating point numbers (i.e. real numbers), and double precision floating

point numbers.[5] It is fine to use a `float` to store an integer, but if only integer operations will be done, that will waste memory and CPU time.

Back to functions

A void function is a function that doesn't return a value. Other functions sometimes will return values and store them in variables in the main program. Functions can also have variables that are passed to the function. There aren't any of those for the `helloworld` function, either, but if there were some, they would go in the parentheses.

Next, we see that the main body of the program calls `helloworld`. Then the program ends.

Reading in input from the user

OK - now let's say we want to print out "Hello World" a bunch of times, and we want to let the user decide how many times. Then we need to figure out how to take input from the user. We use the `scanf` command here.

Take a look at the program `helloworld3.cc` now:

```
/* Hello World program -- with a function */

#include<stdio.h>

void helloworld()

printf("Hello World\n");
return;



main()
{
int i,numtimes;
printf("How many times do you want to print out a message?\n");
scanf("%i",&numtimes);
for (i=1;i<=numtimes;i=i+1)
{
helloworld();
}
}
```

There are a few things to notice here. First, we have to declare the time of the variable `numtimes`[6]. Second, when we use `scanf` to read in a variable, we have to put ``%i'' inside the quotes and an ampersand before the `numtimes`.

[5] We'll get to the idea of precision later on, but double precision variables use twice as much of the computer's memory, and sometimes take longer to be used in computations, but they are less susceptible to round-off errors.

[6] We also declare a type for `i` which we will use just a bit later.

The ‘‘%i’’ is there to tell the computer what type of variable to expect. The ampersand is there for technical reasons that we'll discuss next week, because it's something called a pointer.

Control structures

The next thing that we want to do in programs is to be able to have some process where the computer can use some criteria to decide whether to keep repeating a statement or not to. There are three of these in C: the `for` statement, the `while` statement and the `do while` statement.

When we have a case where we want to do something a specific number of times the `for` statement is the easiest way to go. We have a first line with the starting point for a variable, the ending point for the variable, and the way we change the variable as we go through the loop.

`for` *(initial;final;increment)*
{
set of tasks
}

Then we have a set of curly braces which include all the tasks that we complete. At the end of the set of curly braces, we go back to the beginning of the loop, after incrementing the variable by the amount specified in the *increment* statement.

Sometimes we wish to have conditions that aren't easily specified in a `for` loop. We then have a couple other options: the `while` loop and the `do while` loop. These are almost identical to one another – the only difference is that the `do while` loop runs through one time before checking whether the condition to continue running is met. The syntaxes are:

`do`
{
set of tasks
}
`while` *(condition is true)*;
and

`while` *(condition is true)*
{
Set of tasks
}

There are many reasons for physics-related tasks why you might prefer to use a `while` type loop rather than a `for` loop. For example, suppose you were computing the trajectory of a projectile, and you wanted to stop your calculation when the projectile hit the ground. If you already knew ahead of time how long

the projectile would be in the air (e.g. if you were neglecting air resistance) then you could do this with a for loop – but then you probably wouldn't need a computer to solve the problem. If you don't know how long the projectile will be in the air, then you wouldn't know what to make the maximum value of the loop.

Conditional statements

The other kind of control structures we'll want to work with are "conditionals" – i.e. tasks that we only execute one time, and only if certain conditions are met. In principle, we can always do this with a `while` loop, and the other ways of coding things I will show you here just make things easier.

The workhorse command is the `if` command. The `if` command allows also for `else` and `else if` statements, which, again, are unnecessary, but often quite convenient. The following snippet of code could be used in a computer blackjack game:

```
   if (total==21)
{printf''Blackjack!''}
else if (total>21)
{printf''Busted!''}
else if (total<21)
newcard(deck[]);
```

There is also a syntax involving the commands `switch` and `case`, which can make the coding for some tasks a bit tidier and easier to read (but really not any easier to program), but which is not valuable enough to be worth the time to teach in this class. If you have a menu of tasks you wish to run through, you should feel free to google the syntax for using these commands and decide if you wish to use them.

Pointers and arrays

There are two ways of working with data in C that go beyond what you would normally think of as a single variable. These are pointers and arrays. They are two very different concepts, but we will treat them together.

We already briefly alluded to the concept of the pointer last week when we talked about how to read input from the screen. A variable on a computer has two numbers associated with it – the location of the variable in the computer's memory, and the value of the variable, in the way that we normally think of it.

When C passes a variable to a function, it passes the value of the variable, but not the memory location. What this means is that you cannot change the value of the variable within a function. The way that you can work around this is to use pointers to the location in memory. Then inside the function, you can tell the computer to look up the value stored at that memory location, rather than the memory location itself. This manner of dealing with variables stems from the fact that C was originally developed with the goal of writing operating systems, rather than the goal of doing numerical calculations – for running an operating system, it is useful to be able to work with specific memory locations. It takes a bit of getting used to, but after you get used to it, it's not so bad.[1]

A pointer to a variable's memory location is specified by putting an ampersand in front of it. The value of the variable stored in a memory location is specific by putting an asterisk in front of the pointer.

Some basic rules on how to deal with pointers:

1. When you're in the main program, and you want to send a variable to a function, but don't want to change it, use the variable name. This is the simplest case – everything is intuitive.

2. When you're in the main program, and you want to send a variable to a function and do want the function to be able to change the value, send a pointer.

3. When you're then in the function and you wish to work with the variable to which you have a pointer, you need to remember that it is a pointer that you have and not the variable itself.

Here is an example, the program `read_numbers.cc`. One of the lines is just in there to print out both the pointer and the data value, and to show you that the pointer is generally a very large integer which is meaningless unless you have a lot of expertise in dealing with the innards of operating systems.

```
   /* Simple program to read in number */
/* Really just demonstrates pointers*/
#include<stdio.h>
void read_start_vals(float *x, float*y,float *v, float *theta)
```

[1] This is the reason `scanf` needed an ampersand in front of the variable in the previous week's lecture. Even though we didn't write the function `scanf` ourselves, it is still a function, and if you want the program to change the value of the variable by reading something from the command line, you need to use a pointer there.

```
{
printf("What is the initial value of x in m?\n");
scanf("%f",x);
printf("What is the initial value of y in m?\n");
scanf("%f",y);
printf("What is the initial value of theta in degrees?\n");
scanf("%f",theta);
theta=*theta*pi/180.0;
/*This is to convert theta into radians, since C does trig in rads
*/
printf("What is the initial value of v in m/s?\n");
scanf("%f",v);
printf("For x, the memory value is %li and the actual value is return;
}


main()
{
int i,imax;
float x,y,v,theta,vx,vy,t,fx,fy;
read_start_vals(&x,&y,&v,&theta);
printf("%f %f %f %f \n",x,y,v,theta);
}
```

Arrays

The other type of data that we have to learn to deal with is data stored in arrays. An array is basically the computer equivalent of a vector or a matrix (or tensors when you eventually get to them). It's a set of numbers sorted in a particular way and stored in a single variable.

There are a few things to remember. First, in C, the vectors start from element number 0, rather than from element number 1. In many other programming languages vectors start from element 1. Second, the way that we pass elements to functions is as the whole array, without using symbols indicating they're pointers. When we have the whole array, the compiler assumes that it is a pointer to the first element of the array, and then you can work with the variables as though you had passed a pointer. However, when you send a single element of an array to a function, you need to use a pointer symbol. Take a look at this program, which reads in two arrays of user-defined size, and then takes their dot product:

```
   /*This is a simple program to read two vectors and compute their
dot products */
#include<stdio.h>
#include<math.h>
#include<stdlib.h>
#define MAX_DIM 10

void readinvalues(double vector1[], double vector2[],int *dimensions)
{
/*At the top, we just use vector1[] because C knows we want to use
*/
/*a pointer to the whole array */
int i;
do
{
printf("How many dimensions do the vectors have?\n");
scanf("%i", dimensions);
}
while (*dimensions>MAX_DIM);
for (i=0;i<*dimensions; i=i+1)
{
printf("Enter vector 1, element %i\n",i);
scanf("%lf",&vector1[i]);
}
for (i=0;i<*dimensions; i=i+1)
{
printf("Enter vector 2, element %i\n",i);
scanf("%lf",&vector2[i]);
/*Down here, we need to specify that it's a pointer to vector2[i] */
/*because it's a single element we're passing to another function*/
}
return;
}

double dotproduct(double vector1[],double vector2[],int dimensions)
{
int i;
double dp;
dp=0.0;
for (i=0;i<dimensions;i=i+1)
{
dp=dp+vector1[i]*vector2[i];
/*Here, we leave vector1[i] alone, rather than putting an */
/*asterisk before it, because we are not passing it to */
/*another function.  */
printf("i,vector1,vector2= %i %lf %lf\n",i,vector1[i],vector2[i]);
```

```
}
return(dp);
}

   main()
{
int dimensions;
int i;
double vector1[MAX_DIM];
double vector2[MAX_DIM];
double dp;
readinvalues(vector1,vector2,&dimensions);
dp=dotproduct(vector1,vector2,dimensions);
printf("The dot product is:  %lf\n",dp);
}
```

You can also specify multi-dimensional arrays – for example to do matrix calculations. It can also be a convenient way to keep the position and velocity for an object – that could be a 2×3 array, with:
`x[0,0] =` x
`x[0,1] =` y
`x[0,2] =` z
`x[1,0] =` v_x
`x[1,1] =` v_y
`x[1,2] =` v_z

Choices like this are often a matter of taste, rather than a matter of what is the best way to do a programming problem. The choice to put both the position and velocity into the same variable is likely to make it a little bit easier to get the code written without making a mistake, but a little bit harder for someone knew coming to the problem to change the code if you do not comment it well.

Fourier analysis

This week, we will learn to apply the technique of Fourier analysis in a simple situation. Fourier analysis is a topic that could cover the better part of a whole graduate level course, if we explored it in detail, but it's also something where in a single lesson, along with a bit of the work from the student, you can go from zero knowledge to being able to do some useful things – you will be exposed to the Fourier transform here, and will develop a tool you can use to compute Fourier transforms, but you should expect that you will have to do a lot more learning if, e.g. you run into a research problem that requires a deep knowledge of Fourier transforms. The Fourier transform is one of the most commonly used mathematical procedures across a wide range of scientific and technical disciplines. It is a process which converts a function into a set of sines and cosines.

Often, we perform what is called a discrete Fourier transform. A function is a continuous set of values. Of course, no experiment will give us a truly continuous set of values, so instead, we will want to have a recipe that can take a set of discrete values, and find the set of functions that describe it. There will be some limitations to the process (e.g. we will not be able to figure out how something is varying when it varies much faster than speed at which we collecting the data).

I will give examples based on the assumption that some quantity is varying in time, but the math can be applied to spatial variations (or variations with respect to some abstract quantity) as well.

The simple, intuitive approach

There is a simple approach to figuring out what a Fourier transform means, which is something that can be used to compute a Fourier transform. There is a mathematical approach called *matched filtering* which can be used in its very simplest form here.

Let's consider some arbitrary function, $f(x)$. It ranges from -1 to $+1$ in value. How can we figure out what it is? We can take some other function whose value we know, and we can multiply the two functions together, and take the integral of the result. If the two functions are the same, then the value of the new function is $f^2(x)$. If the two functions are different the value will be something smaller. This is a trivial case, of course, because if we already know the function $f(x)$, we wouldn't try to represent it as a single function.

Let's consider a slightly different case, that we have a function $f(x)$ which we think is made of the sum of a set of sine waves (or, equivalently, with just a phase shift, cosine waves), but we don't know what the frequencies, normalizations or phases of the sine waves are. Then we wish to multiply the function by every sine wave possible, and see where the integrals come out big and where the come out small. This seems like a daunting job, especially when we consider the infinite number of phases we'd need to check.

We can actually get around the phase problem simply by using a little bit

of high school trigonometry.

$$\sin(a+b) = \sin a \cos b + \sin b \cos a. \tag{1}$$

Thus we can take:

$$\sin(2\pi f t + \theta) = \sin(2\pi f t)\cos\theta + \sin\theta\cos(2\pi f t). \tag{2}$$

We can then see that our original function can be expressed as a sine component with a particular frequency and a cosine component with the same frequency, and the relative normalizations of the two components will determine the phase. The phase issue is thus easily converted from an infinite set of possibilities, to just the requirement that we check the normalizations of all the sines and all the cosines.

Thus, for analytic functions we can produce an integral over the sines and an integral over the cosines, and find the coefficients of the sines and cosines. I will spare you the detailed math on this, since it's not the normal way of doing things.

Complex analysis

Instead, combinations of sines and cosines are most conveniently dealt with as complex numbers. A complex number can be written as $re^{i\theta}$, where r is the radius in the complex plane, and θ is the phase in the complex plane. In this way, the real component is $r\cos\theta$ and the imaginary component is $r\sin\theta$. It turns out that this approach leads to a far more efficient way for a computer to deal with taking Fourier transforms than other approaches do.

For taking the Fourier transform, what we want to do is to integrate our function times $e^{i2\pi ft}$:

$$S(f) = \int_{-\infty}^{+\infty} s(t) e^{i2\pi ft}. \tag{3}$$

Then, the real part of the output will give the coefficients of the cosines, and the imaginary part will give the coefficients of the sines.

So this method is fine for getting an idea of what the Fourier transforms of theoretical functions look like, but it's an integral from $infty$ to $+\infty$, so we can't use it for working with data. Real data will have a finite duration, and a finite sampling rate. What we want for working with data is to have a methodolgy that

Discrete Fourier transforms

A discrete Fourier transform is a transformation from a list of numbers to a list of amplitudes and phases of sine/cosine waves at different frequencies. The mathematical formulation is as follows:

$$X_k = \sum_{i=0}^{N-1} x_n e^{-i2\pi kn/N}, \tag{4}$$

where we have a series of N values of x_0 to x_{N-1} which are evenly spaced, and wish to get the coefficients for the sines and cosines at frequency k. If we have

real numbers only, we can show that we get no information from the last $N/2$ frequencies, so that we do not need to compute them. This expression above, then, is something that we could do on the computer. There is a process for setting up complex variables in C++, but in my opinion, it's not any easier than just putting the real part of the complex number into one variable and the complex part into another array, so I'm not going to spend time teaching it. You are welcome to learn it on your own and use it if you want to give it a try.

Also, it will turn out that the obvious way to turn equation 4 into a computer program turns out to be a really bad way to do it – it gives the right answer, but it is a factor of a few hundred slower than it needs to be.

Practical application on a computer

Now that we have been through some of the theory of Fourier transforms, we can consider how we will do them on the computer. Computer scientists, and scientists who use computers, often like to describe how fast an algorithm is based on how the number of operations on the computer scales with the number of data points, or steps of some other kind that will be used. Constant factors are ignored in this kind of analysis – the difference between taking $2N$ steps or N steps isn't what we concern ourselves with, but rather the difference between, for example, taking N steps and N^2 steps. The former causes us to have to wait a bit longer for the program to run, while the latter can leave us with a very limited range in the size scale of the problem we can attempt to solve.

Suppose we are applying some operation to all the elements on a line segment. That operation would take an amount of computer time proportional to N where there are N elements on the line segment. Suppose further that we were applying the operation to a square made of N line segments in each dimension – then we would end up with a process that takes N^2 time steps. If we then went into even higher dimensions, then we would see that the process goes as N^n, where N is the number of elements per dimension, and n is the number of dimensions. This will be an important motivation for learning to do Monte Carlo integration, a topic we will cover in a few weeks.

The Fourier transform would appear to be something that would take N^2 steps – we have to consider N data points and N frequencies. An clever algorithm, called the Cooley-Tukey algorithm was published in 1965 allowed the computation of Fourier transforms in $N\log N$ steps. We won't go into the details, but we will use the method. The original Cooley-Tukey algorithm works only for data set sizes which are powers of 2. In some cases, this can lead to requiring you to throw away nearly half your data, or to make up fake data points in order to be able to use the rest of your data. There are other algorithms that are a bit more flexible which are based on the same principles, but for this assignment, I will just give you a data set on which to work which has a power of 2 as its number of elements. I will also not explain how the Cooley-Tukey algorithm works, except to say that it essentially divides the data into smaller segments to deal with the Fourier transforms at the higher frequencies.

Using libraries

You should never re-invent the wheel, unless you think you can make a better wheel, or you're doing it as a learning experience. I am thus not going to ask

you to write a computer program that implements the Cooley-Tukey algorithm. Instead, I'm going to have you use a routine from the GNU Scientific Library that already does this. We will use the GSL libraries because they allow their libraries to be used freely under all circumstances[1]. There is one catch, which is that you cannot use them in proprietary software.

We will thus have to learn how to use libraries.

This very simple program is taken from the GNU Scientific Library web site:

```
#include <stdio.h>
#include <gsl/gsl_sf_bessel.h>
/*This program has been taken from the GSL web site */
/* http://www.gnu.org/software/gsl/manual/html_node/An-Example-Program.html#An-Example-Pr
*/
int
main (void)
{
double x = 5.0;
double y = gsl_sf_bessel_J0 (x);
printf ("J0(%g) = %.18e\n", x, y);
return 0;
}
```

It includes the library file `gsl_sf_bessel.h` on the second line of the program. There is then a call to a function `gsl_sf_bessel_J0` on a later line, which computes a particular Bessel function's value for $x = 5$. This is a function that we haven't defined – it's within the library file.[2]

There's another step we need to go through for making this work. We need to tell the compile at the time we compile the program that we want to use non-standard libraries. Generally, when you install a library on your computer, you will see some instructions about how to link the libraries at the time you compile. For this one, what you need to do is to add on `-lgsl -lgslcblas`, so you should type:

```
gcc test_gsl.cc -o test_gsl -lgsl -lgslcblas -lm
```

at the time you compile. You can also put these commands into a `makefile`, and then just type `make gsl_test` – this is something that you can teach yourself if you're interested. It will take you a bit of time to figure it out, but it will save you a bit of typing. If you get to the point of building complicated programs that involve a lot of subroutines in different files, you eventually need to learn this, but it's not necessary for a first course. If GSL is in a non-standard place on the computer on which you're working, you'll need to do a few other things to make it work.

[1] Except in some cases where you want to make money selling the software you have produced using their routines as part of your own code

[2] We've actually been using functions like this all along. If you're curious, or don't believe, me, try running `printf` without putting in `stdio.h`.

Numerical integration

Many times in physics we will encounter important and interesting problems for which we can write down the equations governing the system, but for which we cannot solve the equations in closed form. There are some very simple, well-defined functions for which the integral can be computed only numerically.

One *extremely* important example is the Gaussian:

$$g(x) = exp^{-\frac{(x-x_0)^2}{2\sigma^2}}. \tag{1}$$

The Gaussian is also sometimes called the normal distribution. It can be shown[1] that adding together large numbers of distributions almost always eventually produces a Gaussian distribution – this is called the central limit theorem, and you should encounter it in your first course on statistical mechanics, if not sooner.

One thing that is *almost* always distributed as a Gaussian is the background noise in an experiment.[2] That is to say, if you make a huge number of measurements of something, and plot the probability that each measurement will be in a small range between x and $x + \delta x$, that function will be a Gaussian in most real world cases. For this reason, if we want to estimate the probability that some measurement is really telling us about something besides the fluctuations in the background, then we want to be able to integrate out the Gaussian distribution.

Pitfalls for any summation process on a computer

Any time you are summing numbers on a computer, you need to be aware of round-off errors. The computer allocates only a certain amount of memory to writing down a number, and then records a number in binary notation which is as close as possible to the actual number. This is part of the reason we make integers a separate data type on the computer. The amount of memory allocated to a floating point number will vary from computer to computer.[3]

Morten Hjorth-Jensen from the University of Oslo has a nice set of notes on computational physics which is a little bit too advanced, in general for this course, but which provides a nice example of something we can do to see the effects of round-off error. We can just sum $(1/n)$ from 1 to 1,000,000 (or some other large number), and then take the same sum going in the opposite direction. I have written a version of the program myself, and it's called sum_1_n.cc on the course program web page.

On my laptop, I ran this program with regulat floating point precision, and got 14.357358 going forward (i.e. starting at 1), and 14.392652 going backward. I then changed the data type to `double` and I got 14.392727 for both numbers,

[1] Whenever a physicists or a mathematician uses the phrase "It can be shown" you should usually assume that whatever comes after that is (1) a well-established result and (2) probably requires about 15 minutes or more of algebra on the blackboard. You can almost always assume that "it can be shown" also means "it won't be shown".

[2] One notable exception is the rates of counting events when those rates are small. Then they follow something called the Poisson distribution. When the rates become large, the Poisson and Gaussian distributions converge to one another.

[3] `gcc -dM -E - </dev/null | grep FLT` will print this out for you for a standard `float`, and you can change `FLT` to `DBL` to see what happens with double precision numbers.

although the numbers weren't strictly equal at the machine accuracy level, but rather at the level at which I outputted them to the screen.

Addition is not normally the problem, unless you do an enormous number of addition operations, though – subtraction is much tougher for a computer to do correctly. Any time you subtract a number which is almost as big as the number you started with, you risk running into some problems. A nice example of this, which I take from Wikipedia, but check myself, is that of the quadratic equation. Suppose you are trying to solve a quadratic equation:

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

using the quadratic formula you learned in high school:

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}. \tag{3}$$

In some cases, if you try to do this on your calculator, you will get the wrong answer, and the same thing will be true on a computer in floating point arithmetic. The main source of problems will arise when $|b| >> c$ – when that's true, then one of the roots will be very close to zero and the other will be very close to $-b$.

In all cases, there are a few other formulae that we can use. We can see that the roundoff error in the $-b$ value will be pretty small, as a fraction. Then we can use a formula from the French mathematician François Viète, who proved that $x_1 x_2 = \frac{c}{a}$, where x_1 and x_2 are the roots of the quadratic. We can find the root that is computed accurately (i.e. with addition, rather than subtraction) and then the other root will be c/a times the reciprocal of the first root. Whether the root involving taking the positive or negative square root is more accurate depends on the sign of b. This is coded up in the function `viete_formula` in the program `solve_quadratic.cc`.

In summary:

1. Subtraction (or, more precisely, addition of numbers with different signs) is the process that causes the worst round-off error, but round-off error is always possible, even with addition

2. Doing large numbers of calculations also increases the chance of getting bad round-off errors

3. You can often reduce the errors using clever numerical methods. Also, you can often find the clever methods in an applied math book if you know where to look.

Methods for numerical integration

Now that we have discussed the issue of numerical round-off error, we can move on to doing numerical integration on the computer. The reason we first discussed the case of round-off error is that a lot of numerical integrations involve doing an extremely large number of additions, and so numerical round-off becomes something of which you should be aware.

Rectangular integration

The simplest method of doing numerical integration is called the rectangular rule method. This basically involves turning the basic expressions of calculus into a discrete summation (i.e. if you are integrating over x, instead of taking the limit as δx goes to 0, set δx to some small, finite value).

This method is simple to code and simple to understand; those are its advantages. Its main disadvantage is that for functions which change quickly, the step sizes in the integration need to be very small in order to allow it to converge quickly. The errors in one step are of order the square of the step size, and the overall error is of order the step size – so making the steps smaller to get faster convergenece of the integral to the correct answer helps only at the cost of a much larger number of steps being used. Using a very large number of steps increases the effects of numerical round-off error, and, usually more importantly, increases the amount of time the computer program takes to run.

Despite these disadvantages, this method is a good place to start learning, and works effectively for a variety of relatively simple problems. Mathematically, we just need to find the value of :

$$I \approx f(a) \times (b - a), \tag{4}$$

where I is the integral we wish to compute. Then, we can sum the values of a series of these rectangles up, with the widths made smaller and smaller to allow for more accurate computations.

Often, this rule is adapted into something called the trapezoidal rule by taking the average of the function values at the start and end of each step.

$$I \approx (f(b) + f(b))/2 * (b - a) \tag{5}$$

This is practically the same thing, especially as the number of steps taken becomes large. It becomes exactly the same rule if after going through the process one subtracts half value of the function at the start point and adds half the value of the function at the end point – so the most efficient way to compute the trapezoidal rule is actually to use rectangles with only the start point values, and then make the small adjustment at the end.

The errors in numerical integration procedures are usually written out as a coefficient, times the difference between the start and end values of the domain of the integral, times some nth order derivative of the function at some value between the start and end values of the domain. Or, as an equation, perhaps more simply, if we are integrating $f(x)$ from a to b, using steps of size Δ (in this case, the domain is broken into N segments of width $\Delta = (b - a)/N$):

$$E \leq \frac{(b - a)\Delta^2}{24} f^{(2)}(\xi) \tag{6}$$

gives the error on the integral. In this expression $f^(2)$ refers to the second derivative of f, and ξ is a number between a and b.

Simpson's rule

Simpson's rule is a simple way to compute integrals that gives exact values for polynomials up to cubic functions, and which converges to the correct answer much faster than Euler's method for most other functions. Simpson's rule is based on the idea that for each interval, the integral can be approximated by:

$$I \approx \sum \frac{b-a}{6}[f(a) + 4f(\left(\frac{a+b}{2}\right) + f(b)]. \tag{7}$$

Simpson's rule has an error that goes as $\frac{(b-a)^5}{2880} f^{(4)}(\xi)$. Thus for functions with very small third derivatives, the rectangular rule may actually converge faster, but otherwise Simpson's rule should converge faster – that is to say, usually fewer steps can be used with Simpson's rule to get the same level of accuracy as with the trapezoidal rule.

More terms can be added to the process in order to get even faster convergence. There exist things called Simpson's 3/8 rule (which has errors a factor of about 2.5 smaller, and uses four values per interval instead of 3) or Boole's rule (which uses 5 values per interval, and then scales with the sixth derivative of the function). We won't use these in class, but it might be useful for you to be aware of them at some time in the future.

Going beyond the simple approaches

If you have an especially hard integral to compute, and you wish to get the answer as exactly as possible with a reasonable amount of computer time, there are some things you can think about doing. First of all, there are some kinds of functions for which there are entirely different approaches which can be taken. If, for example, you have a function with only small, known, deviations from a polynomial, you can use something called *Gaussian quadrature*. If you have a function which varies very smoothly, or which has very small values, in some regions in x and which varies quickly, and/or takes large values in some regions, you may choose to have adjustable step sizes.[4]

[4]If your function is easily differentiable over a wide range of values, then you can figure out if this is likely to be a useful thing to do by computing its fourth derivative, since the error in Simpson's rule calculations scales with fourth derivative.

Numerical integration for solving differential equations

After integration, it is natural to consider how to find numerical solutions to differential equations on the computer. Simple equations of motion lead to 2nd order differential equations. In the case of constant acceleration, we see that:

$$v = at + v_o, \tag{1}$$

so,

$$x = \frac{1}{2}at^2 + v_0 t + x_0. \tag{2}$$

However, if the force and hence acceleration is related to the position or the velocity in any way, then we cannot just to simple integration in closed form (or at least not simple integration of the form that one typically learns in a first calculus class). [1]

There are still plenty of such problems that can be solved in closed form. Let's consider two problems that come up a lot in a first classical mechanics course – the ideal spring which follows Hooke's Law, and the simple pendulum. In an introductory classical mechanics course, we will usually use the small angle approximation, and say that $\sin\theta = \theta$, which then reduces the math for a pendulum's angular motion to being basically the same as the math for a spring's linear motions. It's a pretty good approximation, but it's not quite right, and the differences are something that you could measure in a lab fairly easily.

So, the equations of motion of a pendulum are:

$$\alpha = \frac{-g}{l}\sin\theta = \frac{d\omega}{dt}, \tag{3}$$

and

$$\frac{d\theta}{dt} = \omega. \tag{4}$$

For if we make the small angle approximation that $\sin\theta = \theta$, then we get:

$$\frac{d^2\theta}{dt} = \frac{-g}{l}\theta, \tag{5}$$

which we can solve by inspection to give a sine wave with frequency $\sqrt{g/l}$.

If we don't make the small angle approximation, there is no closed form solution to the differential equation.

[1] Historically, these problems were solved by perturbation analyses – one would solve a problem that was almost the problem that one wanted to solve, and then figure out how to make small changes to the solution based on the small deviations from the soluble problem. This kind of approach is still often useful for two purposes – (1) it can sometimes guide an efficient approach to finding the numerical solution to the problem and (2) it can sometimes produce simple formula which are approximately correct, and which can guide understanding of what really happening in a way that the outputs of a thousand computer simulations often cannot.

There are a few approaches to solving differential equations on a computer. The simplest, by far, is Euler's method, which is basically like the rectangular rule that we talked about for solving simple integrals last week. Instead of trying to solve a second order differential equation, what we do is we split it into two first order differential equations, and then solve each of them. We look at the force law to solve the equation to get the velocity, and then we look at the velocity to get the equation for the position.[2]

Let's lay this out in abstract terms. We have two variables, x and t. We want to solve for x as a function of t. Let's let the second derivative of x be a function of x itself, and maybe also of the first derivative of x. We define a new variable v to be $\frac{dx}{dt}$.

We just set:

$$x_{i+1} = x_i + \frac{dx}{dt} \times \Delta t \tag{6}$$

$$v_{i+1} = v_i + \frac{dv}{dt} \times \Delta t \tag{7}$$

$$\frac{dv}{dt} = f(x, v) \tag{8}$$

We can now solve this on the computer, as long as we can write down the expression for $f(x, v)$. As an example of how to do this, I have written a program to solve the pendulum problem with Euler's method.

```
   /* Pendulum program */
#include<stdio.h>
#include<math.h>
#define g 9.8 /* sets g for use throughout the program*/
#define deltat 0.001 /*sets a timestep value for use throughout the
program*/

   float euler(float x,float xdot)
{
/*This function just increments a variable by its derivative times
*/
/*the time step.  This is Euler's method, which is a lot.*/
/*like the rectangular rule for evaluating a simple integral.  */
x=x+xdot*deltat;
return(x);
}

   void read_start_vals(float *theta, float *length)
{
printf("What is the initial value of theta in degrees?\n");
scanf("%f",theta); /*theta is already a point so no & needed */
/*The next line converts theta into radians */
```

[2]For the pendulum, really, we're looking at torques, angles, and angular velocities, of course, but the math is applicable to a wide range of problems.

```
*theta=*theta*M_PI/180.0; /*M_PI is pi defined in math.h */
/* An asterisk is needed for theta because we want the value*/
/* at the memory address of the pointer, not the address itself*/
printf("What is the length of the pendulum in m?\n");
scanf("%f",length);
/*Like in the case where we read in theta, we already have a pointer
*/
/*So no ampersand is needed.*/
return;
}

   float angular_acceleration(float theta,float length)
{
float alpha,omega;
alpha=-(g/length)*sin(theta);
/* This is just first year physics, but before we make the small
angle approximation.  */
return alpha;
}

   main()
{
int i,imax;
float alpha,theta,omega,length,t,kineticenergy,potentialenergy,totalenergy,velocity;
FILE *fp;
/*Above lines define the variables*/
/*Mass is ignored here, since it falls out of all equations*/
/*To get energy in joules, etc.  would have to add a few lines of code*/
fp=fopen("./pendulumoffset","w"); /* opens a file for writing the output*/
t=0;
omega=0;
/*The two lines above make sure we start at zero.*/
read_start_vals(&theta,&length); /*Reads initial offset and length of
pendulum */
for (i=0;i<50000; i=i+1) /*Starts a loop that will go through 50000
time steps */
{
alpha=angular_acceleration(theta,length);/*Computes angular accel.
*/
omega=euler(omega,alpha);/*Changes the angular velocity by Euler meth.
*/
theta=euler(theta,omega);/*Changes angle by Euler meth.*/
potentialenergy=g*length*(1.0-cos(theta));/*Computes grav.  pot en.*/
velocity=omega*length;/* Computes linear velocity*/
kineticenergy=0.5*velocity*velocity;/*Computes kin.  en */
totalenergy=potentialenergy+kineticenergy;/*Computes total energy*/
```

```
t=t+deltat;/*Increments the time*/
fprintf(fp,"%f %f %f %f %f %f\n",t,theta,omega,kineticenergy,potentialenergy,totalenergy
Prints out the key variables*/
/*Note:  keeping track of the total energy is a good way to test a*/
/*code like this where the energy should be conserved*/
}

   fclose(fp);/*Closes the file*/
}
```

The big disadvantage of the Euler method is that it tends to give answers which get worse and worse with time. This is actually true for all numerical integration schemes for solving differential equations for nearly any problem. Problems like this one, for which the solutions are periodic are less troublesome than problems for which the solution could go off in any directions. First – in this case, it's obvious that the solution will be periodic, even if the solution is not a sine wave, so we have a sanity check [3]

The next level up in complication is a method to do something sort of like what we did with Simpson's rule, and to try to do something to take into account the curvature of the function for which we are solving. There is a family of methods called *Runge-Kutta* methods which can be used for solving differential equations more precisely than the Euler method can in the same amount of computer time. In this class, we will use the "classical Runge-Kutta" method, which provides a good compromise among simplicitly, accuracy, and computational time needed. It is also sometimes called the RK4 method, since it is a 4th order method in the seconds that the errors per step scale with the step size to the 5th power, so the total errors scale with the step size to the 4th power.

Let's first lay out the method for the simplest case, where we are solving a first order differential equation:

$$\frac{dx}{dt} = f(t, x), \tag{9}$$

along with which we will need a boundary condition, that at t_0 $x = x_0$.

Now, we integrate this out with something analogous to Simpson's rule:

$$x_{i+1} = x_i + \frac{1}{6}\Delta t(k_1 + 2k_2 + 2k_3 + k_4), \tag{10}$$

[3] You may have also noticed that I computed and printed out the energy in this system to make sure that it was conserved which is another sanity check.

where the k_n values are given by taking:

$$\begin{aligned} k_1 &= f(t_i, x_i) \\ k_2 &= f(t_i + \frac{1}{2}\Delta t, x_i + \frac{\Delta t}{2} k_1) \\ k_3 &= f(t_i + \frac{1}{2}\Delta t, x_i + \frac{\Delta t}{2} k_2) \\ k_4 &= f(t_i + \Delta t, x_i + \Delta t k_3) \end{aligned} \tag{11}$$

What is going on in this set of equations is that a series of estimates of the derivative of x with respect to t are being made. The first is the Euler method's approximation. The second makes an estimate of the value of x half way through the time step, based on using the value from k_1, and just adds half a time step to the time. The third makes another estimate based on adding half a time step, but now using the value of the position based on k_2, and the final one is a best guess at what's happening at the end of the time step.

So, this method is great for first order differential equations, but how do we apply it to second order differential equations? Now things get a bit more complicated. Let's take the case that $\frac{d^2x}{dt^2} = f(t, x, \frac{dx}{dt})$, which will allow for solving cases with something like air resistance present. Now, we need to vreak the second order differential equation up into two first order differential equations:

$$\begin{aligned} \frac{dx}{dt} &= v \\ \frac{dv}{dt} &= f(t, x, v) \end{aligned} \tag{12}$$

Then, we need to solve these two equations simultaneously.

We can start with writing down the solutions for the two equations in terms of k_i values for the second derivative, and K_i values for the first derivative – i.e. we will add up the k values to integrate out v, and the K values to integrate out x.

This gives is:

$$\begin{aligned} v_{i+1} &= v_i + \frac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4)\Delta t \\ x_{i+1} &= x_i + \frac{1}{6}(K_1 + 2K_2 + 2K_3 + K_4)\Delta t, \end{aligned} \tag{13}$$

and now we need a prescription for evaluating the values of the k's and the K's.

This is again similar to in the first order equation case:

$$\begin{aligned} k_1 &= f(t_i, x_i, v_i) \\ k_2 &= f(t_i + \frac{1}{2}\Delta t, x_i + \frac{\Delta t}{2} K_1, v_i + \frac{\Delta t}{2} k_1) \\ k_3 &= f(t_i + \frac{1}{2}\Delta t, x_i + \frac{\Delta t}{2} K_2, v_i + \frac{\Delta t}{2} k_2) \\ k_4 &= f(t_i + \Delta t, x_i + \Delta t K_3, v_i + \Delta t k_3), \end{aligned} \tag{14}$$

while evaluating the values of K_i is simpler:

$$\begin{aligned} K_1 &= v_i \\ K_2 &= v_i + \frac{\Delta t}{2} k_1 \\ K_3 &= v_i + \frac{\Delta t}{2} k_2 \\ K_4 &= v_i + \Delta t k_3 \end{aligned} \tag{15}$$

This gives us a recipe that is a bit harder to turn into computer code than the Euler method, but harder only in the sense of being more work, rather than genuinely conceptually harder. At the same time, it's much more accurate that the Euler method, which is why it's worth the effort.

Other methods

Just as Simpson's rule often provides the best compromise among accuracy, time to program, and number of CPU cycles needed for integrals, the RK4 method often provides the best compromise for differential equations. At the same time, there will often be better approaches both for integrals and differential equations.

In this course, we won't go to the lengths of actually executing these "better" approaches, but since you may eventually find yourself in a situation where RK4 isn't good enough, I figured that I should at least give some buzzwords you can look up to find more sophisticated approaches. The first thing people often try is RK4 with adaptive step sizes – that is, instead of making *Deltat* a constant, let *Deltat* increase when the higher order derivatives of the function are small. This is an approach that is useful for dealing with cases where the RK4 method is "good enough" in the sense that it will converge to the right answer given enough computer power, but where you want to get a precise answer more quickly than you can get it with fixed step sizes. There are a range of more and less clever ways to compute optimal step sizes. A few commonly used more advanced approaches are the Gauss-Legendre method, and the Bulirsch-Stoler algorithm. If you run into a problem where you need a more advanced method, you may find it helpful to know the terminology, but we will not discuss these methods in class.

Monte Carlo integration

In some cases, you will want to do integrations of very complicated functions in a large number of dimension (including "dimensions" that are not space nor time dimensions, but things like quantum states, or velocities, or any number of other quantities). If the number of these dimensions is at least 8, then you may be better off using a technique called Monte Carlo integration in order to get an accurate answer than using the standard approaches like the Runge-Kutta method or trapezoids.

Monte Carlo integration is the process of simply drawing a series of random numbers and determining whether they are within a region enclosed by your function. Let's suppose we want to evaluate $\int_0^1 xdx$. We know, of course that the answer is $\frac{1}{2}$, but suppose we didn't know this, or anything about geometry except for the formulae for the area of a square. We could draw a 1×1 square, and we would know its area is 1. Then, we could figure out what fraction of the square is below the function, and that would then give the integral of the function.

Without using any results from geometry or calculus, how could we go about figuring out what the fraction of the area under the curve is? It turns out that we can just draw two random numbers, one for the x value and one for the y value. Then, we can plot the point (x, y), and determine whether it is under the function or not. If we do this enough times, then we will eventually build up an estimate of the integral that will become better and better with more draws.

It turns out that this Monte Carlo integration technique converges extremely slowly – it scales as $\sqrt{N}$, where N is the number of random numbers used. We have to calculate the function once per random number, of course. Simpson's rule has an error that scales as N^{-4}. So, why would we ever use Monte Carlo integration?

There are a few reasons. One is that for very badly behaved functions, we have to be very careful with the step sizes with Simpson's rule – i.e. if the fourth derivative is often extremely large, then it can be hard to ever get convergence with Simpson's rule. The more common reason is if the number of dimensions is large. If we are integrating to find the volume of an object instead of an area, we must use N steps in two different dimensions, or $N \times N$ steps. We can see that if we need to integrate over a large number of different variables, we eventually run into a point where the accuracy of the Monte Carlo simulation in a given number of steps will start to be better than the accuracy of the Runge-Kutta integration or the integration by some other standard method.

Random numbers on a computer

There are no ways to generate *truly* random numbers purely mathematically.[1] What most computers' random number generators do is to start with a

[1] There are ways to generate random numbers by measuring quantum mechanical processes to high precision, since the quantum mechanical processes do have a truly random component to them. You can buy a hardware random number generator that implements one of these solutions, but they're expensive, and usually, if you're careful, for most purposes in physics, the *pseudorandom* numbers that come from a computer program are good enough. For cryptography, it may be another story.

number called a "seed" and then apply some mathematical operations to that number to get the next number in a sequence.[2] The operations are set up in a way that the sequence should not repeat until one gets all the way through all the possible numbers. Beyond that, there shouldn't be any correlations between adjacent values, nor any preference for going up or down in any particular number of steps. Different algorithms do better or worse jobs of this, and in some cases, you will need to check carefully to make sure that the random numbers you are drawing are "random" enough for your purposes. This is another one of these things that we don't need to concern ourselves with for this course, but of which you should be aware in case you do a research level project in the future involving the use of computer-generated random numbers.

We can do the following: `#include <time.h>`
`#include <math.h>`

At the top of the program. These are the libraries that include the time function and the random number function. Then, we will also include the following routine, `r2`:

```
double r2()
{
return (double)rand() / (double)RAND_MAX ;
/* Returns a random number from 0 to 1, since rand gives a number up
to RAND_MAX*/
}
```

What this does is it allows us to take the output from the function `rand` which is the standard random number generator, and convert it into a random number between 0 and 1. If we wanted something between -1 and $+1$, we could then make a multiplication and a subtraction. What we don't usually want is the actual output of that function which is an integer (expressed in double precision floating point form) from 1 to `RAND_MAX`, because `RAND_MAX` varies from computer to computer and anyway, it's not as useful as a number from 0 to 1.

We also need to initialize the random number seed. We only need to do this once, near the start of the program. There are two approaches to doing this. One is to choose a number yourself and use it. The other is to read a number off the computer's clock. To read a number off the clock, we can do:
`srand(time(NULL));`
which will read the clock value and send it to the random number generator as its seed.

There are some specific reasons why you might wish to specify the number yourself: you may be worried that you are getting results that are not specific to how you change the physics of what you are doing in the simulation, but rather due to where you start in the random number generator. You can get some feel

[2]Sometimes, the seed will declared explicitly, and sometimes the seed is taken e.g. from the clock on the computer.

for whether that's true by running through different physics situations with the same set of random numbers. The input to `srand` should be an integer.

Note that you can also enter `srand(1);`
to return to the first random number in the sequence that you chose, so you could just run both sets of physics in the same program. This isn't always convenient, since often it is only after the fact that you realize that you want to try something like this. You could also print out the random numbers to a file, but remember that writing to disk is one of the slowest things that a computer can do – and in some cases, with millions of random numbers with tens of digits each, you will use a lot of disk space. Forcing the random number seed to a particular value really will be the most effective way to test something like this – this actually means that for some specific purposes, using a pseudorandom number generator is better than a true random number generator from a hardware device, in addition to being much cheaper.

Modelling data

Much of the course so far has emphasized how we might go about making simulations of things – i.e. things that are useful for a theoretical physicist who wishes to figure out what might be the expected outcome of some theoretical model. Some of you who plan to do experimental physics or observational astronomy may have been left wondering why computers are really useful for you.

The main point of our lesson this week will be to discuss how to test theoretical models against real data, and how to make estimates of the values of parameters based on theoretical models, when the actual key parameter cannot be measured directly – these are key goals for any experimentalist. First, I'm going to give a bit of a discussion of philosophy about how to work with data. Then, I'm going to give you a recipe for one of the most popular and useful ways to fit models to data, but only after telling you all the reasons not to trust it blindly. At the end of this week's notes, I'll also discuss briefly a few other places where computers are used in experimental physics, but in ways that don't really fit in with this course.

Einstein is quite famous for having said, "No amount of experimentation can ever prove me right; a single experiment can prove me wrong." In the experimental sciences, we can never prove a theory is correct. What we can do, however, is to show that it describes nature very well over a broad range of circumstances – once we do that, it rises from the level of a hypothesis or a model to the level of a theory or a law. We can also prove that one theory provides a better description of nature than another theory does.

Sometimes it can be generations between when an idea is established as a theory which explains a wide variety of phenomena very well until the theory can be established not to describe some experimental data. In these cases, the old theory is often retained as useful and is often still used in cases where it was already established to provide a good description of what Nature does. An example, to which we will come back a bit later, is that of Newtonian gravity versus general relativity as the "correct" theory of gravity. Newton's theory provides calculations which are more precise than any deviation we could hope to measure for a wide range of conditions, and it involves doing calculations simple enough for high school students. Einstein's theory involves using mathematics that very few people learn even in an undergraduate physics degree, and in most cases, results in corrections that don't matter. For certain purposes, almost exclusively in astrophysics, we use general relativity as our theory of gravity. Otherwise, we use Newton's theory.

A general philosophy of how to analyze data

We should never think in science about whether hypotheses are right or wrong. Instead we should think about whether the hypotheses provide a good description of what happens in experiments or a poor description. Hypotheses which provide poor descriptions of nature can be rejected. Hypotheses which provide good descriptions of nature can be retained. This doesn't make them "right" in the sense that something that can be proved mathematically is right. It makes them useful.

A large fraction of the realm of statistical analysis is concerned with hypothesis testing – determining whether we should reject an idea. This is more of an art form than a science – although it's a rather unusual art form that often involves doing calculus.

What we want to do is to find some mathematical technique that helps us to determine whether our model is acceptable or rejected on the basis of the data. There are three important things that we must keep in mind when we are doing this:

1. Sometimes the data are not precise enough to allow us to find the flaws in our model, or the deviations between the model and reality aren't important within the range of parameter space in which we have tested the model.

2. Sometimes the statistical technique we are applying is not quite correct, or the data are not correct.[1]

3. Sometimes the model which is, in essence, correct, deviates from the data more than a model which is totally different from the data. This can happen if there are a lot of fudge factors in one model, and the other model is not fully developed yet.

Let's consider some examples for each of these cases.
Data not precise enough, or tests not made in the right regime

For quite a long time, it appeared that Newton's theory of gravity and Galileo's theory of relativity were just flat out correct descriptions of Nature. We now know that Einstein's theory's of general relativity, and special relativity, are better descriptions of nature. We still use Newtonian gravity and Galilean relativity for a lot of purposes, of course, because the differences between them and Einstein's theories, in "weak" gravitational fields, and at "slow" speeds are very small, and because the calculations involved in using Einstein's theories are much more difficult.

Even before Einstein produced general relativity, there were a few measurements that were better explained by it than by Newtonian gravity. The most famous involves the orbit of Mercury – the precession of the periastron of its elliptical orbit. After correcting for the effects on Mercury's orbit from all the other large, known bodies in the solar system, there were still some effects that remained. We know now that general relativity can fix the problem. Before other evidence also showed general relativity was a better theory than Newtonian gravity, the general assumption had been that there was probably an

[1] The data, of course, are always correct, unless we have done something like make a copying error in collecting the data. But often, we don't understand the calibration of our experiment well enough, and the data don't mean what we think they do, and when we convert from the "raw" data, which are things like "a number of photons we have detected with a photon counting device" or "the time series of voltages measured by a voltmeter" to the kinds of things we compare directly with the predictions of a physical theory, we sometimes make mistakes.

unknown planet in the outer solar system or something like that that was causing these effects. Some other tests, notably the detection, during a solar eclipse, of a star that should have been behind the Sun, really helped cement the idea that general relativity worked, because there was no way to explain such a result in Newtonian gravity, and general relativity explained it to high precision.

In any event, if we had had a cleaner solar system, so that there was not a possibility that Mercury's orbit was affected by something we didn't know about, or something like the binary pulsar which came later on, we could have proved that general relativity outperformed Newtonian gravity much earlier on. Similarly, if we had had the opportunity to make measurements in a stronger gravitational field than that of the Sun, we also could have made measurements earlier on, since the deviations from Newtonian gravity would have been smaller. Instead, until the solar eclipse measurements, we didn't have good enough data to tell the difference between Newtonian and Einsteinian gravity for quite a while.

"Systematic errors" or "wrong data"

Many of you may remember a few years ago there was a claim that neutrinos were seem to be travelling faster than the speed of light. Most physicists, at the time, assumed that there was probably something wrong with the experimental data. This is almost always the reaction of most physicists when such an important (or in this case potentially important) discovery is made. It was later found out that there were two pieces of equipment failure in the experiment. The major one was simply a loose cable. As a result, the measurements did not mean what the physicists thought they meant.

Cases where a model is basically right, but not sophisticated enough

We know realize that the Earth goes around the Sun. For a long time, though, there was a philosophical notion that the Sun and the planets orbited the Earth. The astronomers of the time were able to make mathematical formulae that could predict the locations of all the planets to high precision by using "epicycles" – basically making circles within circles. The small circles were orbits within the big circles – sort of the way the moon orbits the Sun, making smaller orbits around the Earth.

When Kepler made the first real quantitative predictions about the orbits of the planets on the basis of a heliocentric solar system, his predictions were not as accurate as the predictions of the geocentric model with epicycles. Those epicycles were basically fixing both the global problem with the geocentric model, and with problems related to the gravitational attractions of the planets on one another and the moon on the Earth.

This is the diciest question for scientists to deal with. One can invoke Occam's razor, which states that the simplest theory that explains all the data is correct. But what happens when one theory is much much simpler and the other explains the theory slightly better? Or when one theory is simpler, but rests on questionable assumptions? In some cases, there are mathematical techniques for dealing with this – *Bayesian analysis*, and the use of something called the f-test. We won't get into these, but Bayesian analysis really just forces us to state a lot of our assumptions very clearly ahead of time, and the f test is really

only useful under cases where one model is exactly the same as another, except for an added level of complication.

Usually the way we should deal with these cases, instead, is by getting some outside information for which the two models make very different predictions. In the case of the solar system, the key idea was put forth by Galileo (and actually before Kepler came along) – that Venus shows phases in the same way that the moon does. Galileo showed that there was no way to have this happen if both Venus and the Sun were orbiting the Earth, but that it would happen naturally in the Copernican/Keplerian view of things. The moral to this story is that we should not become overly reliant on fitting one type of experimental data against models, particularly when the data and the models agree reasonably well. Instead, we should look for the kinds of tests of theories where the competing theories differ as much as possible.[2]

χ^2 fitting: a workhorse for experimental sciences

Most of the time in the experimental sciences, we will be working with data where the measurement errors are well described by a Gaussian function. Another way of saying this is that if one were to make a particular measurement an infinite number of times, then the probability density function that would result would be a Gaussian with mean. If we go back to the formula from a few weeks ago:

$$f(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-} \frac{(x-\mu)^2}{2\sigma^2}, \tag{1}$$

we can now take a look at the meanings of the parameters for the Gaussian probability density function.

The mean value of the function will be μ and the standard deviation of the function will be σ. Usually, if a value is reported as $x \pm y$, y is the value of σ for the variable x. This is not always the case – in some fields of research slightly different numbers are used. But you can see that it doesn't mean that the full possible range of values of x is from $x - y$ to $x + y$. This is an important point to take home, because this is a point that confuses students a lot of the time.

So what is σ? It is the square root of the variance of the data. The variance is what is called the second moment of the data. The nth moment about a particular value c is defined by:

$$\mu_n = (x-c)^n f(x) dx. \tag{2}$$

If we set $c = 0$, and $n = 1$, we get the first moment, which is the mean. If we set $c = \mu$ (so that we are taking the moment about the mean), and set $n = 2$, then we get a quantity called the variance. The square root of the variance tells

[2]This is especially true when the experimental data are known about before the theoretical work is done, which is most of the time. Then, the theoretical work is aimed at explaining the data. This is a perfectly reasonable approach, but it is also something that we need to bear in mind – theorists will often adjust the assumptions they make a little bit to be sure they match the existing data. Coming up with new tests which make radically different predictions pushes the theorists into a corner. Good theorists try to push themselves into a corner by proposing ways that their theories could be proved false.

Number of sigmas	Percentage of distribution
0.67	50
1	68.27
2	95.45
1.645	90
3	99.7
4	99.9936
5	99.999943

Table 1: The table gives the percentage of a Gaussian distribution enclosed within $\pm$ the number of multiple σ away from the mean.

us something about the "typical" uncertainty in the parameter value. If $f(x)$ is a Gaussian, then μ will be the mean and σ^2 will be its variance.

Since you already know how to integrate Gaussians, you can check this, but there are regions called confidence intervals for Gaussians:

For large multiples of σ away from the mean, it can be more convenient to give the probability that an event takes place outside the distribution, rather than within it.

If we have a single variable with a normal distribution, then we can test a measurement by looking at the difference between our data value and the predicted value based on our hypothesis in terms of the difference in units of σ. What we try to do when we compare a model with some data is to estimate something called the "null hypothesis probability". This is the probability that just random chance, based on the measurement errors, would give us as bad a match we get when we do the experiment, even if the model were an exactly correct description of what is happening in the experiment. A large null hypothesis probability means that the hypothesis cannot be rejected *by this experiment alone*. That is, the hypothesis has survived this test. This is, at some level, evidence in favor of the hypothesis, but it is not proof of the theory – as we have stated above[3], no theory can ever be proved correct, but rather theories can merely be demonstrated to provide excellent descriptions of nature over a broad range of conditions.

Suppose we had reason to believe that the Earth's gravitational constant is 9.8 km/sec/sec exactly at sea level, and we were trying to test whether we measured a different value at an elevation of 3000 m. Suppose we measured a value of 9.705 ± 0.019. We then subtract the two values and get: 0.095 ± 0.019. This is 5σ and we can see that 99.999943% of the measurements should be within 5σ of the expected value. Therefore, there is only a 0.000057% chance that such a large deviation would occur due to measurement errors, provided that we have correctly estimated our measurement errors. Under these circumstances, we could very safely say that g has changed between our two measurement locations. This would also provide strong support for the idea that g changes as we increase altitude above sea level, but we would want to take more measurements at

[3]It bears repeating.

different altitudes to be more confident in drawing that conclusion.

This approach is thus great, if we just want to compare one number with another number. That's not what we usually want to do in physics. Usually, we want to compare a set of measurements with a model that attempts to describe what is happening in Nature. Most often, what we will have is data in the form of x and y, with uncertainties on x and on y. Our procedure is greatly simplified if (1) y is a function of x and (2) the errors on x are sufficiently small that we can ignore them. For the purposes of this class, we will proceed as though those conditions are met.[4] We will then have three quantities associated with each measurement we make – the values of x, y and σ_y.

We won't go into a full mathematical derivation, but what we can do is to definte a new quantity called χ^2:

$$\chi^2 = \sum_{i=0}^{n} \left(\frac{f(x_i) - y_i}{\sigma_{y,i}} \right)^2 , \tag{3}$$

where n is the number of data points we have, the subscript i is telling us that we are using the ith value in the sequence, and $f(x)$ is the function we are testing. Essentially, we are summing the squares of the differences between the data and the model. We can perhaps make a leap of intuition without doing all the rigorous math here, and say that this makes sense, since σ is the mean value of the square of the difference between the actual value of the data points and the values you would get by sampling the distribution randomly a large number of times.

Next, we have to figure out how to interpret that. We need to know how many "degrees of freedom" we have for the model. A degree of freedom is essentially a "test" of the model The number of degrees of freedom is typically represented by ν. If we have a model which is completely constrained ahead of time, then the number of degrees of freedom is the number of data points. However, in most cases, our model will have parameters that we are trying to estimate at the same time that we are trying to test whether the model is correct. Every time we have a parameter that we allow to vary in order to determine whether the data can be explained by the model, we use up one of the degrees of freedom provided by the data. There are mathematically rigorous ways to think about this, which are a bit too complicated for this course, but a quick way to think about this is that you can always fit an n element data set with a polynomial of order $n-1$ if you can vary all n coefficients (including the constant) – but that doesn't mean anything, really. On the other hand, if you can fit $n+50$ data points with a polynomial of order n, that really probably does mean that the data are well described by a polynomial.

OK – so now we know how to compute χ^2 and we know how to compute the number of degrees of freedom. As you have probably guessed, if χ^2/ν equals 1, we have a good fit. The model is a statistically acceptable description of the data. But how do we figure out what the situation is when χ^2 is *not* 1? When

[4] It is often not the case, and then more complicated procedures must be used for testing hypothesis.

is a fit rejected? We can compute the probability of getting a value of χ^2 at least as large as the measured one by chance. The formula:

$$F(\chi^2, \nu) = \frac{\gamma\left(\frac{k}{2}, \frac{\chi^2}{2}\right)}{\Gamma\left(\frac{k}{2}\right)} \tag{4}$$

gives the total probability that one would obtain a value of χ^2 less than or equal to the measured value, so the probability of getting a worse answer by chance is $1-F$. The function γ is something called the lower incomplete gamma function and the function Γ is the actual gamma function. If you ever want to write a computer program to compute the these things both of these functions are in the GNU Scientific Library and most commerical mathematical packages.

The one last thing we need to worry about is how to get the best values of the parameters for the model we have in mind. What you want to do is to find the minimum value of χ^2 for a given model by changing around the parameters. A variety of algorithms do this. The Levenberg-Marquardt algorithm is probably still the most commonly used one, but there are more computationally intensive algorithms that can sometimes do a better job. The chief worry with any fitting algorithm is that it may find what is called a "local" minimum of χ^2 – that is a set of parameters which does not give the actual minimum for χ^2, but for which *small* changes in the parameter values result in a worse fit.

Fitting functions with GNUPLOT: some recipes and some examples

Now, with the philosophy of hypothesis testing, and the theoretical foundations for using χ^2 fitting laid out, we can move on to how we go about actually fitting functions to data within `gnuplot`.

I have generated a file, "output_values_linear" with fake data with errors. The data are produced from a straight line with $y = mx + b$, with $m = 1$, and $b = 2$. I then randomly assigned 30 values of x, and 30 values of σ_y to the data points. I then computed what y should be, and then changed y by taking a random deviate from a Gaussian distribution with $\sigma = \sigma_y$. Basically, this data set should be consistent with a straight line with slope 1 and intercept 2. We probably will get values very close to those values, but not exactly equal to them. Let's give it a try.

First, here are the numbers:

Now, let's first plot the data in `gnuplot`:

```
gnuplot> plot "./output_values_linear" u 1:2:3 with yerrorbars
```

and we see that it looks pretty much like a straight line, so it is reasonable to try a straight line model.

Now, we need to define the model we want to use:

```
gnuplot> f(x) = m*x+b
gnuplot> m=1
gnuplot> b=2
gnuplot> fit f(x) "./output_values_linear" u 1:2:3 via m,b
```

We then get some output: $m = 0.997465, b = 2.03362$ with 28 degrees of freedom and χ^2/ν of 0.779. We have a good fit, and we recovered what we thought we were going to recover.

Now, let's see what happens if we try to fit an exponential to the data.
`gnuplot> f(x) = m*exp(-x/b)`
`gnuplot> fit f(x) "./output_values_linear" u 1:2:3 via m,b`
Here, we get a value of χ^2/ν of more than 75 – basically the exponential is such a horrible fit that b runs up to a very high value, so that effectively the exponential becomes a constant that is the mean value of y.

Finally, you should always take a look at the residuals any time you fit a function to some data. `gnuplot> plot "./output_values_linear" u 1:(2-f(1)):3 with yerrorbars` will give you a plot of the data minus the model. Sometimes when you get a reasonably good fit, you will still see a trend in the residuals that tells you that there is something interesting going on that you might have missed otherwise. Also, when you get a bad fit, you can often isolate what is going wrong and figure out how to make a slightly more complicated model that will then become a good fit. I saw a nice example of this from a carefully-done experiment made by my lab instructor when I was a freshman in college. He had measured current versus resistance with a variable resistor and an old fashioned ammeter that had a needle on it. He found that a straight line was a very good fit to the data, but then there was a small additional smoothly varying residual. This was due to the weight of the needle on the ammeter.

So, this is just a start to how to analyze data, but if you master the ideas in this set of notes, and you can do the exercises, and you remember when you are violating the key assumptions that go into χ^2 fitting and then either take the results with a grain of salt, or use a more sophisticated approach, you should be able to do a lot of useful work.

Other uses of computers in experimental/observational sciences

1. Computers are often used in the data collection process in the experimental sciences. For example, computers can be used to control experiments, or to collect readout data from an instrument (e.g. an oscilloscope or a voltmeter or a charge coupled device). This type of work often involves specialized techniques for programming, since it gets in to the details of the hardware, and often involves specialized software for using, since the user just wants the answer, and doesn't want to have to understand the details of the computer hardware, unless the computer hardware can lead to distortions of the measurements. It is thus a very important area, but not really appropriate for a first course on programming.

2. Computers are often used to do simple processing of large data sets. We discussed the case of the Fourier transform already, which is often used for processing time series data (and sometimes, two dimensional Fourier transforms are used to process images). A variety of other types of time series and image processing are done in physics, astronomy, geology, engineering, and increasingly in certain areas of biology and chemistry. The data from the Large Hadron Collider, and other particle accelerators, are compared with computer simulations of what different kinds of particle interactions should look like, and this is how it is inferred what types of particle interactions took place in the collider.

3. Computers, and the internet can be used to distribute data to colleagues, and to the general public. This is rather straightforward in the case of simply sending data to your colleagues. One of the more interesting applications in the past couple of decades has been that of "citizen science" projects.

   These projects have taken a couple of forms. One is the "volunteer computing" approach. Certain types of tasks require a large number of computations, which can be split up easily. At the same time, large numbers of computer owners use their computers only a small fraction of the time. Some enterprising scientists have created software which people can download that allows their computers to do work on projects for the outside scientists. A few physics-related projects which have taken advantage of this include LHC@home, which does simulations needed for making the most of the Large Hadron Collider, Einstein@home which searches radio telescope data for evidence of gravitational radiation, and SETI@home, which involves searching through data from radio telescopes for signals of extraterrestrial intelligence. Interestingly, there are also many biology projects using volunteer computing, but they are almost all for theoretical calculations of protein structures, rather than for analysis of experimental data. If you are interested in contributing computer CPU cycles, check out boinc.berkeley.edu.

   The other way that citizen science can work is when there are types of data that need to have a large number of humans look at them. There are still types of pattern recognition that humans do much better than machines. Many of these are on the borderline between psychology and artifical intelligence – e.g. the Royal Society's Laughter project in which you listen to some laughs and try to determine if they are real or fake. Others involve physics, like the Galaxy Zoo project in which you look at images of galaxies from large sky surveys and tell which type of galaxy it is, which direction it's rotating, and also flag up if anything looks unusual. Large numbers of non-experts can be trusted to get the right answers, as a group, an awful lot of the time, and a small fraction of the data can be examined by experts in order to determine how often things go wrong by trusting non-experts. This is a type of project that requires the internet. If people needed to make printouts and pay postage in order to send the data back and forth, the cost of such projects would be prohibitive. The Galaxy Zoo project has resulted in a few discoveries of things that were made only because a human looked at the images, but which didn't require real experts to be the first ones to notice something unusual. I am not enough of an expert on the marine biology experiments using the same techniques to identify types of fish in videos from underwater experiments, but I would guess that they are just as effective.