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.