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.