Astronomy 8824: Numerical Methods Notes 1
Numerical Integration

Reference: Numerical Recipes, Chapter 4

A Preamble on Data Sizes, Memory, and Disk

(See, e.g., NR §1.2, ECP Table 9-2, or google terms like "floating point precision")

Numbers on a computer are stored as a sequence of bits, each of which is 1 or 0 (i.e., binary values).

A "single precision" floating point number is represented by 4 bytes (32 bits). This can generally represent a number with 6 significant decimal places and an exponent, i.e., a fractional error of about 10^{-6}.

A "double precision" floating point number is represented by 8 bytes. This can generally represent about 14 significant decimal places.

Integers are usually represented by 4-byte or 8-byte sequences, depending on the desired dynamic range and whether the integers are signed or unsigned (i.e., strictly ≥ 0).

If computation speed and memory are not an issue, then you might as well use double precision, and this is the default assignment in Python.

However, there are two reasons you might decide to use 32-bit instead of 64-bit precision.

The computation speed may be faster for 32-bit, depending on the computer, code, compiler, etc.

If you have a lot of numbers, then the factor-of-two memory savings for 32-bit may be important. This is often issue with 3-dimensional calculations.

For example, if you are doing a simulation with a billion particles (1000^3), then you need 4 Gbytes for each phase space coordinate in single precision, 24 Gbytes in total, and that's before you include other things you might care about like gravitational potential, temperature, etc. If you're using double precision, you need twice as much memory.

This should be compared to the RAM available on your computer. My MacBook Pro laptop and my linux workstation both have 8 Gbytes of RAM, so I can't do a billion particle simulation on either one, even in single precision.

The amount of *disk space* I have is many hundreds of Gbytes, so I can easily store the output of many billion particle simulations, but I can't read all of this information at once.

If your calculations are likely to be memory limited, then you should think carefully about the precision you need for them. If they are not, go ahead with double precision.

Suppose you have a long list of floating point numbers that you want to write to disk. If you write this as a binary file (e.g., with numpy.save in Python) then the size on disk will be the same as the size in memory, and reading and writing is relatively fast.

However, you can't easily view what's in a binary file; you have to have a program read it, and you have to know what the format is ahead of time.

If you write your file in ascii (e.g., with numpy.savetxt in Python), then it is easy to see what's in the file using a standard text editor. However, the storage required is typically larger (1 byte per character that you write), and i/o is *much* slower because each character has to be read and translated.

If you have short lists (e.g., less than a million numbers), then the convenience and transportability of ascii probably wins, unless you are going to be reading and writing so many times that speed is an issue.

If you have long lists, then you probably want to read and write binary files.

Computing a Numerical Integral

We want to evaluate

$$I = \int_a^b f(x)dx.$$

This is equivalent to solving the ODE

$$\frac{dy}{dx} = f(x)$$

with boundary condition $y(a) = 0$ for $y(b)$.

Thus, ODE solvers (which we'll discuss later) can be used for numerical integrals in tricky cases — mainly ones where adaptive step sizes are required. Conversely, some ODEs can be solved by a simple numerical integration.

Euler Method

$$S_N^{\rm Eul} = \sum_{i=1}^{N} f(x_i)h_N,$$

where N is the number of (equal-sized) integration steps and

$$h_N = \frac{b-a}{N}, \qquad x_i = a + (i-1)h_N.$$

Geometric interpretation.

Bad method. Error per step is $O(h^2)$, error of integral is $O(h)$. Can drift systematically from correct result.

Trapezoidal Rule

$$S_N^{\rm Trap} = \sum_{i=1}^{N} \left[\frac{1}{2}f(x_i) + \frac{1}{2}f(x_{i+1})\right] h_N.$$

Linear fit to function between x_i, x_{i+1}.

Much better method. Error $O(h^3)$ per step, $O(h^2)$ total.

Geometric interpretation.

Can be written

$$S_N^{\rm Trap} = \sum_{i=1}^{N} \left[\tfrac{1}{2}f(x_i) + \tfrac{1}{2}f(x_{i+1})\right] h_N = \tfrac{1}{2}\left[f(x_1) + f(x_{N+1})\right] h_N + \sum_{i=2}^{N} f(x_i) h_N,$$

so requires no more function calls than Euler.

Automatic tolerance control: Keep doubling N, check if $|S_N/S_{N/2} - 1| < \texttt{TOL}$.

Simpson Rule

$$\int_{x_1}^{x_3} f(x)dx = h\left[\frac{1}{3}f_1 + \frac{4}{3}f_2 + \frac{1}{3}f_3\right] + O(h^5 f^{(4)}).$$

Parabolic fit to curve in $x_1 - x_3$.

Stringing together steps yields

$$\int_{x_1}^{x_N} f(x)dx = h_N\left[\frac{1}{3}f_1 + \frac{4}{3}f_2 + \frac{2}{3}f_3 + ...\frac{4}{3}f_{N-1} + \frac{1}{3}f_N\right] + O(h_N^4 f^{(4)}).$$

Neat and useful trick:

$$S_N^{\rm Simp} = \frac{4}{3}S_N^{\rm Trap} - \frac{1}{3}S_{N/2}^{\rm Trap}.$$

So with interval doubling trapezoidal routine, get (usually) much higher accuracy of Simpson's rule for free.

Simpson routine.

With each iteration, can take advantage of fact that 0.5 times the previous iteration is all of the odd terms in the trapezoid sum, and only the even (intermediate) terms need to be calculated.

Convergence plot.

Roundoff Error

In single-precision, roundoff error is $\sim 10^{-6}$.

Error in $y = 1.0 + 10^{-6}$ is 100%.

If adding up 10^6 numbers of similar order, result can be way off if roundoff error is biased.

Even if unbiased, roundoff error in sum is $\gg 10^{-6}$.

Worse if numbers are of different order.

Implications:

Reducing number of steps is important for accuracy, not just speed.

Always be cautious if adding $> 10^4$ numbers in single precision.

When necessary, use double precision, roundoff $\sim 10^{-14}$.

32-bit integer arithmetic is like single precision.

Midpoint Method

$$S_N^{\rm Mid} = \sum_{i=1}^{N} f(x_{i+1/2}) h_N$$

(Sum to N is correct because of definition of x_i; $x_{N+1/2} = b - h_N/2$.)

Geometric interpretation.

Like trapezoid, also second order.

Doesn't require evaluation at end points. Good for integrals where $f(x)$ is undefined at endpoint but has integrable singularity, e.g.,

$$\int_2^{10} \frac{dx}{(x-2)^{1/2}} \ .$$

But can't use doubling trick to get Simpson's rule.

Transformation of Variables

Improper integrals (with limits of $-\infty$ or ∞) can be treated by transformation of variables.

Practically useful example, with substitution $t = 1/x$,

$$\int_a^b f(x)dx = \int_{1/b}^{1/a} \frac{1}{t^2} f\left(\frac{1}{t}\right) dt.$$

Variable substitutions can be useful in other situations, so that variation of integrand is similar in each dx interval.

Otherwise, required h is set by most rapidly varying region of integrand, but tiny steps are still being taken when integrand is varying slowly.

Example: The substitution $y = \ln x$ shows that

$$\int_1^{1000} \left(x + \frac{1}{x}\right)^{-1} dx$$

is equivalent to

$$\int_0^{\ln 1000} \left(1 + e^{-2y}\right)^{-1} dy.$$

The second integral can be evaluated with many fewer steps because the integrand changes significantly when x changes by a constant *factor* rather than a constant *interval* Δx.

It is sometimes necessary/useful to break an integral into two or three parts, with different variable substitutions, to deal with improper integration boundaries or with different regimes of integrand behavior.

Other Methods

Trapezoid/Midpoint/Simpson are robust methods that work well for general cases, when they are fast enough.

If you need higher accuracy for given amount of CPU time (or less CPU for given accuracy), more sophisticated methods can be significantly faster.

Richardson extrapolation (a.k.a. Romberg integration, NR routines `qromb` or `qromo`):

Perform integration to various values of h.

Fit polynomial to results, extrapolate to $h = 0$.

For smooth integrands, big speedup possible.

But significantly less robust, requires some care.

Probably method of choice for general purpose smooth function if Trapezoid/Simpson/Midpoint aren't fast enough.

Gaussian quadrature:

Choose both abscissas and weights:

$$I \approx w_1 f(x_1) + w_2 f(x_2) + w_3 f(x_3) + ...$$

More degrees of freedom allows higher order with same number of function evaluations.

Can choose weights to make approximation exact for

$$\int_a^b W(x)f(x)dx,$$

where $f(x)$ is a polynomial of a given degree, $W(x)$ is a weighting function.

Requires work to figure out weights, abscissa locations. Many standard cases worked out and tabulated.

For smooth integrands, can be much more efficient.

Method of choice for smooth, well known functions that need to be evaluated many times at high accuracy.

See Numerical Recipes discussion.

Multi-dimensional Integrals

If N is number of steps needed per dimension, total number of function evaluations scales as N^D.

Multi-dimensional integrals can become hard very quickly.

Reduce dimensionality of integration whenever possible, using symmetries of problem.

Simple boundary, smooth integrand: Recursively use 1-d integration.

$$\int\int\int dx\,dy\,dz\,f(x,y,z) = \int_{x_1}^{x_2} dx \int_{y_1(x)}^{y_2(x)} dy \int_{z_1(x,y)}^{z_2(x,y)} dz\,f(x,y,z).$$

If you cannot analytically solve for, e.g., $z_1(x,y)$, you can choose a larger volume and set $f(x,y,z) = 0$ outside the integration region (if you have some test on (x,y,z) that tells you when you are in that region).

If so, be careful of losing accuracy at the boundaries, where the value of f changes discontinuously.

You may want to use numerical root finding (discussed later in the course) to find the inner limits, $z_1(x,y)$, $z_2(x,y)$, etc.

Complicated boundary, smooth integrand, high accuracy not required: Monte Carlo integration (NR §7.6).

Choose a volume V that encloses integration region, e.g., a cube.

Choose N random points that evenly sample this volume.

Sum the values of the function $f(\mathbf{x})$ at points $\mathbf{x}$ that lie within the integration region, setting $f(\mathbf{x}) = 0$ outside that region.

The volume integral

$$\int f(\mathbf{x}) dV \approx \sum f(\mathbf{x}_i) \times V/N \ ,$$

where V is your enclosing volume.

Each point is representing a volume $dV = V/N$.

An estimate of the error is

$$\pm V \left(\frac{\langle f^2 \rangle - \langle f \rangle^2}{N} \right)^{1/2}$$

where

$$\langle f \rangle = \frac{1}{N} \sum_{i=1}^{N} f(\mathbf{x}_i), \qquad \langle f^2 \rangle = \frac{1}{N} \sum_{i=1}^{N} f^2(\mathbf{x}_i) \ .$$