Astronomy 8824: Problem Set 1

Due Tuesday, September 3, 2019

The goal of this problem set is to write a program to do numerical integrals of user specified functions, and to compare the performance of some simple algorithms for doing so. You should read sections 4.0-4.4 of Numerical Recipes as background. However, while their description of algorithms and their properties is good, you should write your own code rather than borrow the NR subroutines.

Specifically, you will write a Python program that computes

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

where a and b are finite limits of integration and we will use various choices for f(x) below. Start with the code available on David's course web page, which implements the "Euler method"

$$I = \sum_{i=1}^{N} f(x_i)h_N$$

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

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

1. Look through the code so that you understand its structure. Note in particular that it automatically doubles the number of steps (starting at N = 4) until it converges to a specified fractional tolerance. The program calculates the integral with N = 4, then doubles the number of steps and compares the answers. If the fractional difference is larger than the tolerance, it doubles the number of steps again, continuing until the fractional difference between two successive evaluations is less than the tolerance. It also has a safeguard with a maximum number of steps, to prevent the program from running forever if it doesn't converge.

Using this code, compute the integral

$$\int_1^5 \frac{1}{x^{3/2}}dx$$

How many steps (approximately) are required to get an answer with a fractional error $|(I - I_{exact})/I_{exact}| < 10^{-6}$?

[There are four ways to run the program: (1) Make it executable, then `integrate.py 1 5`. (2) `python integrate.py 1 5`. (3) Start iPython in the directory where you have the program. From the iPython prompt, type `%run integrate.py 1 5`. In these three cases, the integration limits are read from the command line. (4) Copy and execute the code in a jupyter notebook.]

For a specified number of steps, the integral is evaluated using either the function `euler_loop` or `euler`. The former uses a loop structure typical of fortran or C, while the latter uses array operations available in NumPy.

Compare the speed of these two implementations using the `%timeit` function of iPython.

Start up iPython (by typing ipython at the command line). Then

```
%timeit -n 4 %run integrate.py 1 5
```

will run the code twelve times (3 loops, n = 4 times in each loop) and report the average execution time in the best of the 3 loops.

Do this once using the loop implementation of the integral and once using the array implementation.

Which is faster and by how much?

2. Modify the code so that it implements the "Trapezoidal Rule":

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

(This seems at first sight to require twice as many function evaluations as the Euler method, but there is an obvious way to avoid this, which you should implement in your program.)

You can modify either the `euler` or `euler_loop` function to accomplish this, or you can start your own code from scratch if you don't like David's. If you write your own code, you should maintain the automatic step-doubling-to-convergence feature.

Numerically compute the integral from Part 1 with both methods. How many steps (approximately) are required to get an answer with a fractional error $|(I - I_{exact})/I_{exact}| < 10^{-4}$ for the Euler method and for the Trapezoidal Rule? What about 10^{-6}?

3. With step doubling, you can implement a neat trick, described in Numerical Recipes. Given estimates IT_N and $IT_{N/2}$ from the Trapezoidal Rule using N and N/2 steps, make the new estimate $IS = (4 \times IT_N - IT_{N/2})/3$. This approximation, Simpson's Rule, should converge faster than the Trapezoidal Rule. Write a routine to compute an integral via Simpson's Rule using this trick. This involves changing the integration driver routine; for any given N you are still using the Trapezoidal Rule.

Make a plot of the error $|(I - I_{exact})/I_{exact}|$ vs. the step size *h* for the Euler, Trapezoidal, and Simpson's Rule evaluations of the above numerical integral. Include this plot (log-log is recommended) as part of your solution set. Is the behavior what you expect?

Note: If Simpson's doesn't converge noticeably faster than the Trapezoidal Rule, there is a bug in your program.

4. For each of the following integrals, give the value of the integral *I* for each of the three numerical integration methods and the number of steps needed to get convergence to a fractional tolerance of 10^{-6}.

$$\int_1^2 \frac{1}{x^{3/2}(1 + x^{3/2})} dx$$

$$\int_1^{100} \frac{\sin x}{x} dx$$

$$\int_1^{1000} \frac{\sin^2 x}{x^2} dx$$

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

$$\int_0^{\ln 1000} (1 + e^{-2x})^{-1} dx$$

Comment on the comparison of the last two integrals and the number of steps required to compute them.

Note: If you are using an array-based implementation, you will need to use `np.sin` in your integrand, e.g., `return (np.sin(x)/x)`. If you are using a loop-based implementation, you can use either `np.sin` or `math.sin`. Do you know why?

5. Write a routine that implements the Midpoint Rule:

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

where $x_{i+1/2} = x_i + h_N/2$.

For the test integral of Part 1, compare the convergence of the Midpoint Method to that of the Euler, Trapezoid, and Simpson's Rule methods. Add the Midpoint results to your convergence plot.

Use your routine to compute the integrals

$$\int_0^4 \frac{1}{x^{1/2}} dx$$

$$\int_0^{\pi} \frac{\sin x}{x} dx$$

Why is the Midpoint Rule useful even though it is less accurate than Simpson's Rule for the same number of steps? Why does the second integral converge much faster than the first integral?

6. Numerically compute the integrals

$$\int_1^{\infty} \frac{dx}{x^2 + x^3}$$

$$\int_1^{\infty} \frac{\sin^2 x}{x^2} dx$$

Explain how you did it. (Hint: See NR §4.4, or the lecture notes. You do not need to approximate ∞by a large finite number.)

Getting an accurate result for the second integral is much harder than for the first integral. Why?

7. *Optional*: There are numerical integration routines available in scipy.integrate. You can see what they are in iPython by `import scipy.integrate as si` and then `si?` and get

more information about individual routines via, e.g., `si.quad?`

For the 3rd and 4th integrals from Part 4 (the ones with ulim=1000), use the scipy routines `si.quad` and `si.romberg` to evaluate the integrals. Do you get the same answers that you got from Simpson's Rule? Use `%timeit` in iPython to compare their speed to that of your Simpson's Rule program for these two integrals. What do you find?

You can use the program `si.py` provided on David's web page for this experiment if you wish.

8. So that you end this exercise with something of practical use (and we'll use it later in the semester), adapt your Simpson's Rule integrator to a program that specifically computes the co-moving distance to an object at redshift z in a flat universe with a cosmological constant. The formula for the comoving distance is

$$D_C(z) = \frac{c}{H_0}\int_0^z \frac{H_o}{H(z')}dz'$$

with

$$\frac{H(z)}{H_0} = [\Omega_m(1+z)^3 + \Omega_\Lambda]^{1/2}$$

Your program should take as arguments Ω_m, H_0 (in km s^{-1} Mpc^{-1}), and z. Because we are assuming a flat universe, $\Omega_\Lambda = 1 - \Omega_m$.

For $\Omega_m = 0.3$ and $H_0 = 67$ km s^{-1} Mpc^{-1}, what is the comoving distance to redshifts 0.5 and 2, in Mpc?

For more on cosmological distances see:

Hogg 1999, arXiv:astro-ph/9905116

Weinberg et al. 2013, Phys Rep 530, 87, §2

Aubourg et al. 2015, Phys Rev D, 92, 123516 (arXiv:1411.1074)

Note: This problem set was originally developed by David Weinberg.