Unit 3: Numerical Integration

Computer Modelling

1 Aims

- Add code to your Cosmnology class to compute the distance integral.
- Explore the convergence of the different integration methods.
- Make a results document to record your results and plots.

2 Preparation

- Make a copy of your unit 1 file `unit1.py` and save it as `unit3.py`.

- Look at the feedback you got for unit 1 and make any changes you think are necessary. Feel free to consult the TAs for advice.

- Start a results document, for example in Google Docs or Word, where you will record various results and plots, below. You will be submitting this with your code. Your results document must be organized so that we can easily tell which part of the assignment each section corresponds to.

3 Background

3.1 Numerical integration methods

In numerical integration we try to compute the value of a definite integral of a function $f(x)$ over some interval $[a, b]$, in the usual case where we cannot do the integral analytically. These methods typically involve evaluating the function at a number of points in the interval, and summing them up in some way. The choice of points and the weights we give them in the sum defines the algorithm.

As with almost all numerical methods, there is an inherent inaccuracy in numerical integration. The amount of inaccuracy (the error) depends both on the specific algorithm we choose, and on the number of points we use in the sum. Using more points gives a lower error, but takes longer to compute as we have to evaluate the function at more points. Sometimes this doesn't matter because our function may be so fast, but in many other cases, for example if we are evaluating it a large number of times, we have to carefully consider the number of points we need in a trade-off.

We have to take care when testing numerical methods with simple functions like polynomials - some methods are exact for low order polynomials but much worse higher up. For example, the trapezoid rule is exact for linear functions, but not for higher order functions.

3.1.1 Rectangle rule

The simplest numerical integration method is the rectangle rule, which is (assuming equal width rectangles):

$$\int_a^b f(x)dx \approx \Delta x \sum_{i=0}^{n-1} f(x_i)$$

where $x_i = a + i\Delta x$ and $\Delta x = (b-a)/n$.

The rectangle rule corresponds to approximating the area under the blue curve as the area of the rectangles in the figure below:

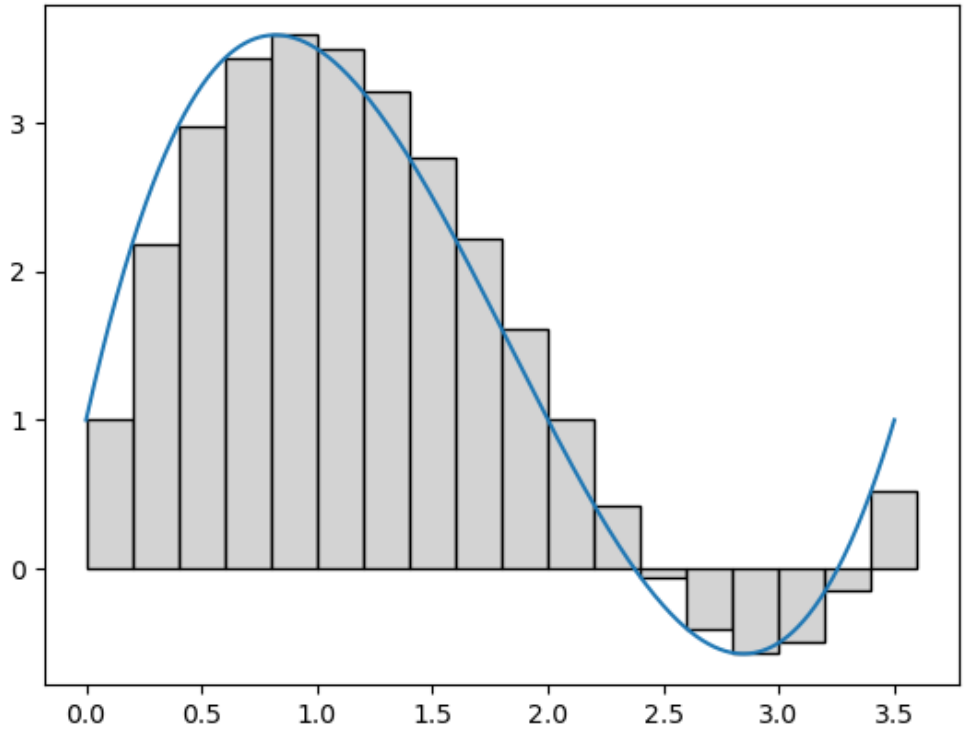

Figure 1: Rectangle Rule illustration

3.1.2 Trapezoid rule

In the trapezoid rule, we approximate the area under the curve as the sum of the areas of trapezoids instead of rectangles:

$$\int_a^b f(x)dx \approx \frac{\Delta x}{2}\left[f(x_0) + 2\sum_{i=1}^{n-2} f(x_i) + f(x_{n-1})\right]$$

where this time $\Delta x = (b-a)/(n-1)$ and we still have $x_i = a + i\Delta x$.

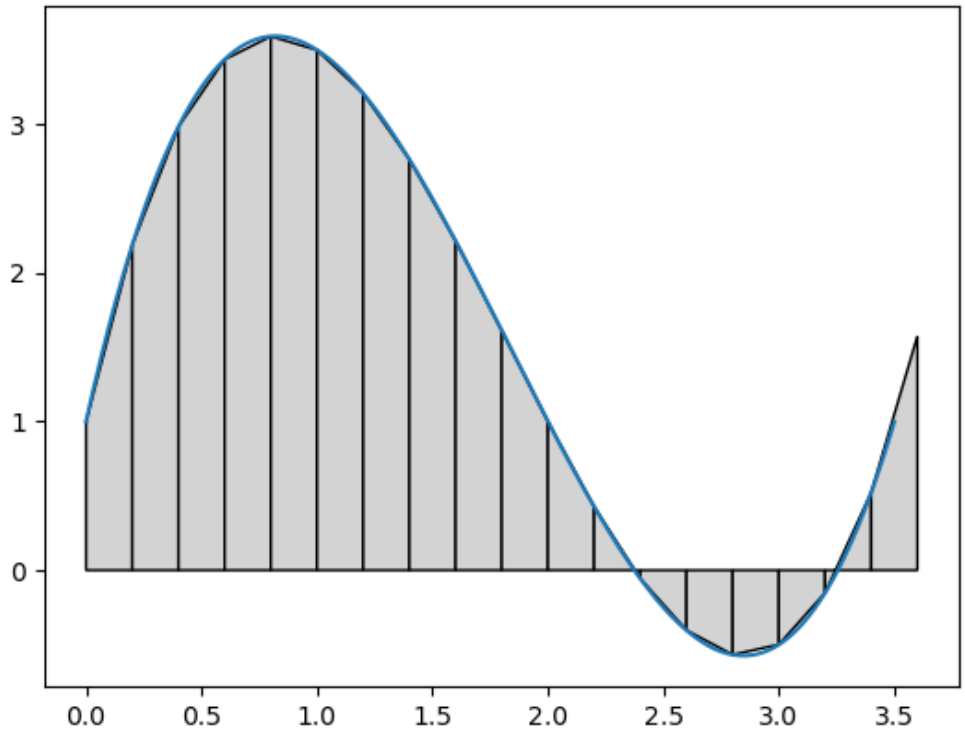

Figure 2: Trapezoid Rule illustration

3.1.2.1 Cumulative trapezoid rule We often want to compute a cumulative version of an integral - that is, an array Y_i where

$$Y_i = \int_a^{X_i} f(x)dx$$

where X_i is an increasing series of x values. This is useful because we can then interpolate into this array to get the value of the integral at any point.

We can do this by setting $Y_0 = 0$ and then iteratively computing the remaining elements as:

$$Y_i = Y_{i-1} + \frac{\Delta x}{2}\left[f(x_{i-1}) + f(x_{i-1})\right]$$

In this case it's wise to store the values of $f(x_i)$, to avoid recomputing them.

3.1.3 Simpson's rule

In Simpson's rule we effectively fit a quadratic curve to the top of each of the small integration regions, instead of a flat top (rectangle rule) or a straight line (trapezoid rule). It turns out this is equivalent to:

$$\int_a^b f(x)dx \approx \frac{\Delta x}{3}\left[f(x_0) + 4\sum_{i=0}^{n-1} f(x_{2i+1}) + 2\sum_{i=1}^{n-1} f(x_{2i}) + f(x_{2n})\right]$$

This time $\Delta x = (b-a)/2n$ and we still have $x_i = a + i\Delta x$.

3.2 Cosmological distances

In unit 1 we defined a distance $D(z)$, but because the universe is expanding there are several ways we can measure distances. In this course we will use a variant called the *luminosity distance*, $D_L(z)$. We can compute this from our original distance $D(z)$ as:

$$D_L = (1+z)\,\frac{c}{\mathrm{H}_0}\,\frac{1}{\sqrt{|\Omega_k|}}\,S\left(\sqrt{|\Omega_k|}\frac{\mathrm{H}_0 D(z)}{c}\right)$$

where c is the speed of light and $S(x)$ is the function:

$$S(x) = \begin{cases} \sinh(x) & \text{if } \Omega_k > 0 \\ x & \text{if } \Omega_k = 0 \\ \sin(x) & \text{if } \Omega_k < 0 \end{cases}$$

Our measurements of brightness are often in the form of another quantity, the *distance modulus* $\mu(z)$, which is defined as:

$$\mu(z) = 5\log_{10}(D_L(z)/\text{megaParsecs}) + 25$$

3.3 Interpolation

Interpolation is the process of estimating the value of a function at a point between the values we have calculated without re-evaluating the function. In cases where the function is slow, or where we have to evaluate it many times, this can be a significant time saver.

There are a great many different interpolation methods, from simple linear interpolation to more complex methods like splines. Some methods are designed to operate on noisy data, so that they don't have to go exactly through all the sampled points, and others are designed for theoretical data where we expect the function to be smooth.

In our case, if we have a large number of galaxies in the same distance range, we don't want to integrate to that distance for each one, but instead we can integrate once and then interpolate the result for each galaxy. This is much faster, but we have to be careful that our interpolation method is accurate enough for our purposes.

We won't explore the mathematics of interpolation methods too deeply in this unit. Instead we will use the scipy class `scipy.interpolate.interp1d`. This class creates an interpolator object which you can then use like a function to get interpolated values. For example:

```
import numpy as np
from scipy.interpolate import interp1d
x = np.linspace(0, 10, 100)

# This is our function to interpolate
y = np.sin(x)

# Create the interpolator object
f = interp1d(x, y)

# Now we can use it like a function
print(f(5.5))
```

4 Tasks

4.1 Adding numerical integration methods

Add three methods to your Cosmology class to compute the distance $D(z)$ to a single given redshift within that model. Use the original distance from unit 1, NOT the luminosity distance yet. In each case you should do the integral over redshift using a different numerical integration method:

1. The rectangle rule
2. The trapezoid rule
3. Simpson's rule

The integrand that you wrote for unit 1 should be used as the function to integrate, but note that there is a factor of (c/H0) outside the integral in the instructions for unit 1.

The number of points n in the rule should be a second input to your function.

You should not use the scipy library or any other package to implement these, though you may use them to check your results.

Ensure that you document in your comments or docstrings what units the distance is computed in, and use the corresponding correct units for other terms in the equation.

If you have done everything right the distance to a value $z = 1$ should be around 3200 Mpc with the default parameters we used previously ($H_0 = 72$km/s/Mpc, $\Omega_m = 0.3$, $\Omega_\lambda = 0.7$, $\Omega_K = 0$).

4.2 Convergence testing

Write a program that uses your cosmology class and does the following:

- Use a very large number of steps, for example 10^4, with your methods to get a highly precise estimate of the distance to redshift z=1.0 for a fixed cosmology. Record this in your results document.

- Explore how the accuracy of the three integration methods changes as you vary the number of steps. Plot the absolute fractional error in the distance as a function of the number of steps for each method. Consider how you can make this plot more clearly show the variation in the data, and fairly reflect how many evaluations of the function each algorithm requires.

- In your results document, choose a good target accuracy for your calculation based on physical reasoning. Show this target on your plot, and include the plot and a discussion of your reasoning for the target in your results document.

- Given your results, write in your results document which method you would recommend for this calculation, and what value of the number of steps to use.

4.3 Cumulative version

- Write a new method that uses the cumulative form of the Trapezoid rule to compute an array of distances to a range of redshifts. It should take as inputs the maximum redshift z and the number of steps n, and return an array of distances to redshifts from 0 to z.

- Write a test program that computes the distances to redshifts from 0 to 1.0 using this method, and plots the result. Save the plot in your results document.

- Use the scipy interpolator to write a function or method that accepts an array of z values, which might be in any order, and returns an array of corresponding distances $D(z)$.

- Use this method to write a function or method that does the same but computes the distance moduli $\mu(z)$ of all the redshift values.

4.4 Exploration

- For each of the three main parameters, explore and demonstrate the effect of changing them on the distance modulus as a function of redshift. Make plots illustrating your findings in your results document.

5 Submitting

Submit two files through learn, `unit3.py`, and a PDF of your results document.

5.1 Marking

You will be marked out of 20 for this unit, and the submission is worth 10% of the final course grade. The marks are broken down as follows:

- 5 marks for the results document
- 10 marks for code functionality
- 5 marks for code quality and style