Homework 1 Computational Physics

Giacomo Recine[*]

22 September 2025

Contents

[*]gr2640@nyu.edu

1 Introduction

In this homework, we aim to compute numerical derivatives and integrals using various methods, and we will also explore the impact of numerical errors and machine precision on these computations. All calculations are performed in single precision, meaning that the computer uses 32 bits to represent floating-point numbers. The main advantage of single precision is improved computational speed compared to double precision. However, this comes at the cost of reduced numerical accuracy. To implement this in Python, we use the np.float32() command, which ensures that the variables are consistently defined with single-precision floating-point format. In Chap. 2, we focus on the computation of numerical derivatives for two specific functions at two given points. In particular, we analyze how different sources of numerical error affect the results. Chap. 3 is dedicated to numerical integration. Here, we examine several integration techniques and again study the influence of truncation and roundoff errors on the accuracy of the results. Finally, in the last chapter Chap. 4, we present a physical application of numerical integration: the calculation of the correlation function in cosmology. This example illustrates how numerical methods are used in a real-world scientific context.

2 Exercise 1: Numerical Derivatives

In this section, we explain how to compute the numerical derivative of the following two functions:

$$f(x) = \cos(x), \qquad f(x) = e^x, \tag{1}$$

We proceed through the following steps:

- we compute the numerical derivatives at two specific points, $x = 0.1$ and $x = 1$, using three different methods;
- we make a log-log plot of the relative error ϵ versus the step size h, to verify whether the observed scaling and number of significant digits match theoretical expectations;
- we discuss the distinct roles of round-off error and truncation error, which become evident in the error plots.

Let us begin with the first step.

2.1 Computation of the numerical derivatives

As mentioned earlier, we now compute the derivative of the functions in Eq. (1) at the points $x = 0.1$ and $x = 1$, using three different methods: the forward method, the central method, and the extrapolated method. Let us briefly review how each of these methods works:

- the forward method is based on the definition of the derivative:

  $$f'(x) \simeq \frac{f(x+h) - f(x)}{h}, \tag{2}$$

  where h is a small real number. In the mathematical definition of the derivative, the limit $h \to 0$ is taken. However, in numerical computations, as we will see, choosing a smaller h does not always guarantee more accurate results due to the influence of truncation error;
- the central method is very similar to the forward method; the only difference is that the function is evaluated at both the previous and the next point with respect to x. This leads to the following expression

  $$f'(x) \simeq \frac{f(x+h) - f(x-h)}{2h}, \tag{3}$$

  where, again, h is a small real parameter. The same issues present in the forward method, such as the trade-off between truncation and roundoff errors also apply here.

- the extrapolated method is a bit more sophisticated, as it effectively takes into account higher-order terms in the Taylor expansion. Here, we consider the following two approximations

$$f_1'(x) \simeq \frac{f(x+h)-f(x-h)}{2h},$$
$$f_2'(x) \simeq \frac{f(x+2h)-f(x-2h)}{4h}, \tag{4}$$

  and then compute the derivative using the extrapolation formula

$$f'(x) \simeq \frac{4f_1' - f_2'}{3}. \tag{5}$$

The algorithms are implemented in the following Python functions:

```
#we define the method to use to integrate

def forward_method(h, f, x):
  h = np.float32(h)
  x = np.float32(x)
  der = np.float32((f(x+h)-f(x))/h)
  return np.float32(der)

def central_method(h, f, x):
  h = np.float32(h)
  x = np.float32(x)
  der = np.float32((f(x+h)-f(x-h))/(2*h))
  return np.float32(der)

def extrapolated_method(h, f, x):
    h = np.float32(h)
    x = np.float32(x)
    term1 = np.float32((f(x + h) - f(x - h)) / (2 * h))
    term2 = np.float32((f(x + h/2) - f(x - h/2)) / h)
    return np.float32((4 * term2 - term1) / 3)
print(x)
```

Implementing these three methods to compute the numerical derivative at $x = 0.1$ and $x = 10$, always in single precision, we obtain the following results:

Function	x	Forward	Central	Extrapolated
$\cos(x)$	0.1	-0.1493758	-0.0996670	-0.0998335
$\cos(x)$	10.0	0.5850357	0.5431169	0.5440229
$\exp(x)$	0.1	1.1623180	1.1070138	1.1051708
$\exp(x)$	10.0	23165.5469	22063.2813	22026.5371

Table 1: Numerical derivative approximations using forward, central, and extrapolated methods for $\cos(x)$ and e^x at $x = 0.1$ and $x = 10$ in single precision.

2.2 Errors in Numerical Derivatives

As already mentioned, in numerical computations the step size h should be neither too small nor too large. We now investigate this statement in more detail by computing the relative error of each method as a function of different values of h. The relative error is defined as

$$\epsilon = \left| \frac{f'_{\text{estimated}} - f'_{\text{true}}}{f'_{\text{true}}} \right|, \tag{6}$$

where $f'_{\text{estimated}}$ is the numerical derivative obtained using Eqs. (2), (3), and (5), and f'_{true} is the exact analytical derivative of the function. In our case, we consider the functions $f(x) = \cos(x)$ and $f(x) = e^x$, whose exact derivatives are $f'_{\text{true}} = -\sin(x)$ and $f'_{\text{true}} = e^x$, respectively. As before, by plotting the relative error ϵ as a function of h for $x = 0.1$ and $x = 10$, we obtain the following results:

- for $f(x) = \cos(x)$ at $x = 0.1$ and at $x = 10$, we have:

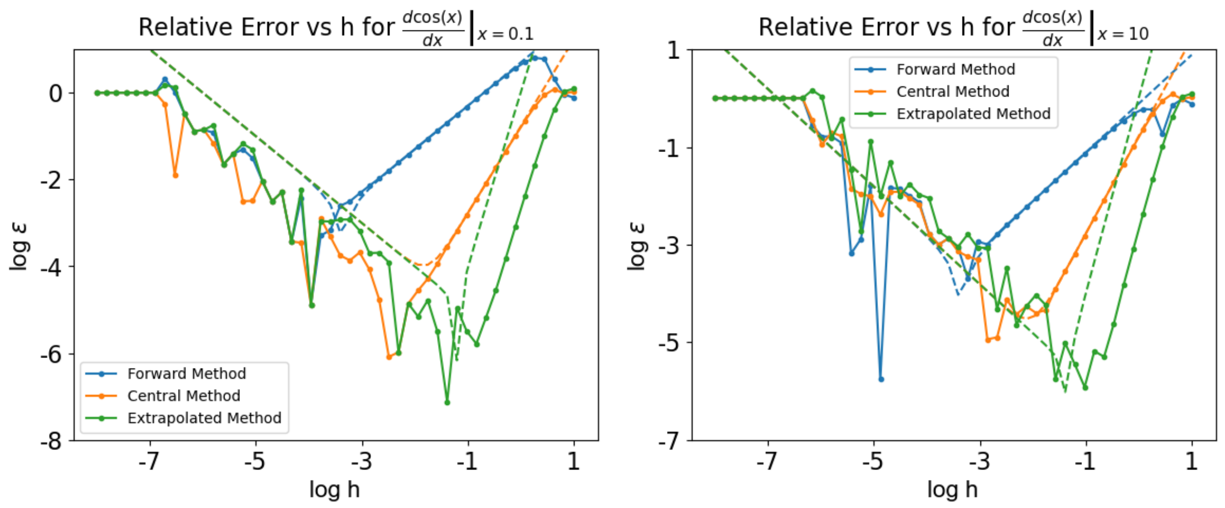

Figure 1: Relative error for $f(x) = \cos(x)$ at $x = 0.1$. The dashed lines represent the theoretical estimates of the relative error, as described below these graphs.

- for $f(x) = e^x$ at $x = 0.1$ and at $x = 10$, we have:

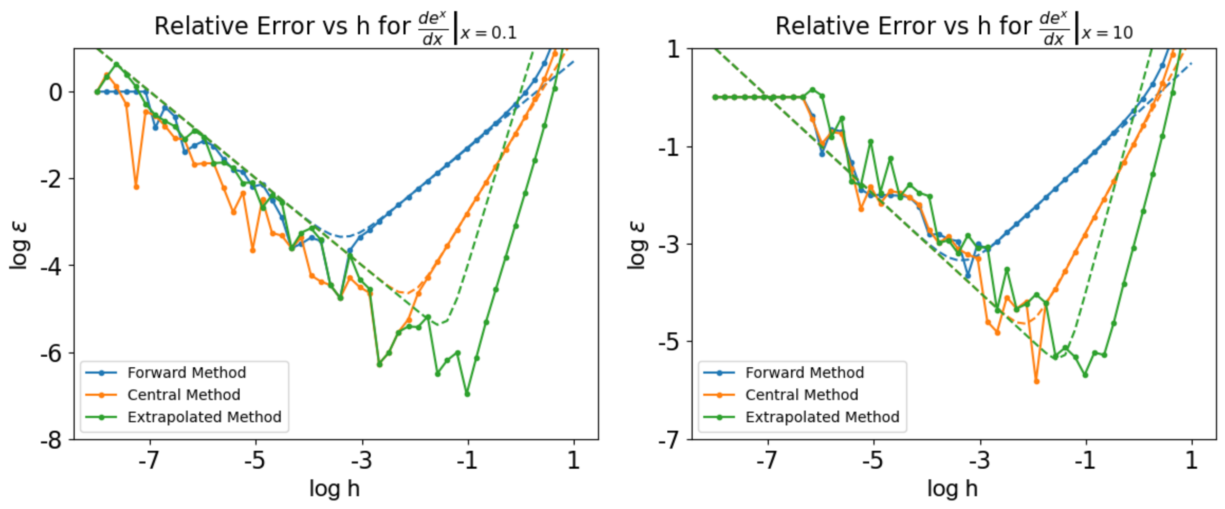

Figure 2: Relative error for $f(x) = e^x$ at $x = 0.1$ and at $x = 10$. The dashed lines represent the theoretical estimates of the relative error, as described below these graphs.

We observe that, in all plotted cases, two distinct regimes of behavior emerge. The first regime occurs for values of h up to approximately 10^{-1}, while the second regime, starting from this value, appears to exhibit a linear trend. This is because, for small values of h, the error is dominated by truncation effects. In all methods, the truncation error scales as

$$e_{\rm t} \sim \epsilon_m \frac{|f'(x)|}{h}, \tag{7}$$

where ϵ_m denotes the machine precision. Since we are working in single precision, we take $\epsilon_m \sim 10^{-7}$. This expression implies that as h decreases, the error increases, a trend clearly visible in the plots.

The second regime, beginning around $h \sim 10^{-1}$, is dominated by roundoff error. Theoretically, this error scales as follows:

- for the forward method:
$$e_{\rm r}^{\rm fwd} \sim h\, f''(x), \tag{8}$$
- for the central method:
$$e_{\rm r}^{\rm cnt} \sim h^2\, f^{(3)}(x), \tag{9}$$
where $f^{(3)}(x)$ is the third derivative;
- for the extrapolated method:
$$e_{\rm r}^{\rm ext} \sim h^4\, f^{(5)}(x), \tag{10}$$
where $f^{(5)}(x)$ is the fifth derivative.

Using these expressions, the theoretical estimate for the relative error is given by

$$e = \frac{e_{\rm r} + e_{\rm t}}{|f'(x)|} \tag{11}$$

and we find that these theoretical trends are in good agreement with the observed values of the relative error. This confirms the correctness of our implementation and allows us to clearly identify the two error regimes, as discussed above. It is also worth noting that none of the plots drop below 10^{-7}, which, as previously stated, represents the machine precision due to the use of single-precision arithmetic.

3 Exercise 2: Numerical Integration

In this section, we discuss numerical integration performed using three different methods. As before, we work in single precision. In particular, we are interested in analyzing the behavior of the relative error associated with each method.

We begin by computing the value of the following integral:

$$\int_0^1 e^{-t}\, dt, \tag{12}$$

using the midpoint method, the trapezoidal method, and Simpson's method. Let us briefly review each of these methods:

- midpoint method: the integral is approximated as
$$I_{ext} = \int_a^b f(x)\, dx \simeq \sum_{i=1}^{N} f(x_i)\, h, \quad \text{with} \quad h = \frac{b-a}{N}, \quad \text{and} \quad x_i = \left(i - \frac{1}{2}\right) h, \tag{13}$$
where N is the number of subintervals into which the interval $[a, b]$ is divided, and h is the width of each subinterval.

- trapezoidal method: to improve upon the midpoint method, we approximate the function over each subinterval using straight line segments rather than constant values. The integral is then approximated by

$$I_{ext} = \int_a^b f(x)\,dx \simeq h\left[\frac{1}{2}f(a) + \frac{1}{2}f(b) + \sum_{i=1}^{N-1} f(x_i)\right], \tag{14}$$

  where h and N are defined as above, and $x_i = a + ih$.

- simpson's method: instead of approximating $f(x)$ with straight lines, this method fits a second-order polynomial (a parabola) to the function over pairs of subintervals. The integral is then approximated as

$$I_{ext} = \int_a^b f(x)\,dx \simeq \frac{h}{3}\left[f(a) + f(b) + 4\sum_{\substack{i=1\\ i\,\mathrm{odd}}}^{N-1} f(a+ih) + 2\sum_{\substack{i=2\\ i\,\mathrm{even}}}^{N-2} f(a+ih)\right], \tag{15}$$

  with the only requirement that N must be even.

These algorithms are implemented in the following three Python functions

```
#Midpoitn Method
def midpoint_method(N, xi, xf, f):
    h = np.float32((xf - xi) / N)
    total = 0.0
    for i in range(N):
        x_mid = np.float32(xi + (i + 0.5) * h)
        total += np.float32(f(x_mid))
    return np.float32(h * total)

#Trapezoid Method
def trapezoid_method(N, xi, xf, f):
    h = np.float32((xf - xi) / N)
    total = np.float32(0.5 * (f(xi) + f(xf)))
    for i in range(1, N):
        x = np.float32(xi + i * h)
        total += np.float32(f(x))
    return np.float32(h * total)

#Simpson Method

def simpson_method(N, xi, xf, f):
    if N % 2 == 1:
        N += 1  # Simpson N even
    h = np.float32((xf - xi) / N)
    total = np.float32(f(xi) + f(xf))
    for i in range(1, N):
        x = np.float32(xi + i * h)
        if i % 2 == 0:
            total += np.float32(2 * f(x))
        else:
            total += np.float32(4 * f(x))
    return np.float32(h / 3 * total)
```

Using this three method, we compute the integral in Eq. (12), we obtain the following the results

Method	Results
Midpoint	0.6321205
Trapezoidal	0.6321207
Simpson	0.6321209

Table 2: Results of numerical integration of $\int_0^1 e^{-t}\,dt$ using different methods.

The true value of the integral is

$$I_{\text{true}} = \int_0^1 e^{-t}\,dt = 1 - e^{-1} \approx 0.6321206. \tag{16}$$

We observe that the results obtained from the three numerical methods are all consistent with the true value of the integral.

3.1 Errors in Numerical Integration

We now aim to understand the role of numerical errors in the integration procedure. In particular, we estimate the relative error, defined analogously to the case of the numerical derivative:

$$\epsilon = \left| \frac{I_{\text{ext}} - I_{\text{true}}}{I_{\text{true}}} \right|. \tag{17}$$

By plotting this quantity as a function of N on a log-log scale, we obtain the following figure:

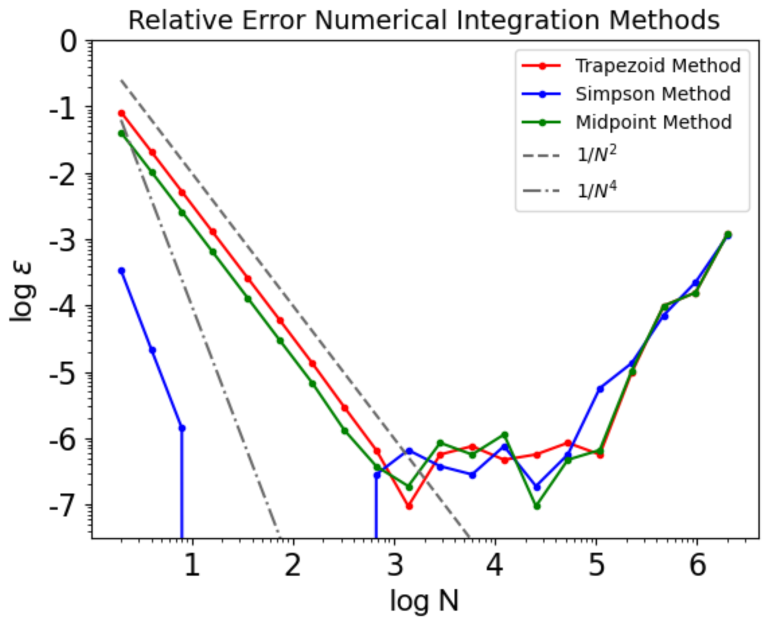

Figure 3: Relative error for the three numerical integration methods applied to the integral in Eq. (12).

Once again, we observe two distinct regimes in the error plot. The first regime, for $\log N \lesssim 3$, shows a clear linear trend in the log-log scale, which corresponds to the region dominated by truncation error. The second regime, for $\log N \gtrsim 3$, deviates from this behavior and is instead dominated by roundoff error. In the truncation-dominated region, we observe the expected convergence behavior: the error decreases as $\mathcal{O}(1/N^2)$ for both the midpoint and trapezoidal methods, and as $\mathcal{O}(1/N^4)$ for Simpson's method. These power-law decays appear as straight lines in the log-log plot, with slopes corresponding to the theoretical order of accuracy of each method. Finally, the "infinity" spike observed in the Simpson curve is not a true divergence. Instead, it indicates that the relative error has dropped below the machine precision threshold (approximately

10^{-7} for single precision), making it no longer representable with sufficient accuracy. This leads to numerical artifacts that appear as a vertical jump in the plot.

4 Exercise 3: Power spectrum and Correlation Function

In this section, we apply the concepts learned about numerical integration to a more physical problem: the calculation of the cosmological correlation function to estimate the Baryon Acoustic Oscillation (BAO) peak.

In cosmology, density fluctuations in the matter distribution are characterized by the power spectrum $P(k)$, which represents the root-mean-square (rms) amplitude of density waves as a function of the wavenumber k (in units of h/Mpc). In configuration space, these density fluctuations are described by the correlation function $\xi(r)$, at a given scale r, typically expressed in Mpc/h. These two quantities are related by the following integral relation

$$\xi(r) = \frac{1}{2\pi^2} \int dk\, k^2 P(k) \frac{\sin(kr)}{kr}. \tag{18}$$

Using the tabulated data, the power spectrum $P(k)$ behaves as shown below:

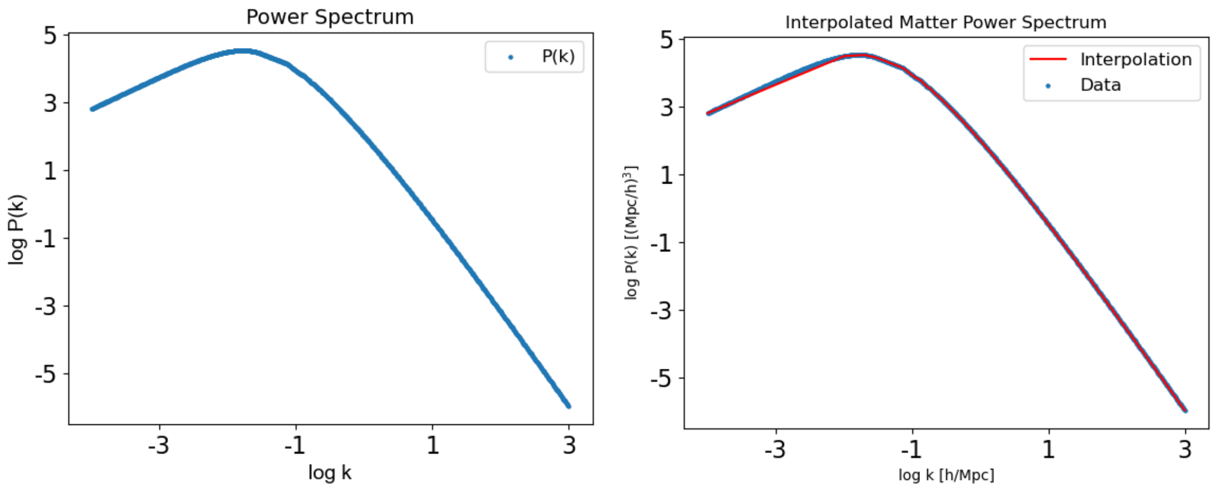

Figure 4: Power spectrum obtained from the tabulated data on the left, and cubic spline interpolation of the power spectrum over the full range of k on the right.

To compute the correlation function, we must numerically evaluate the integral in Eq. (18). However, since $P(k)$ is provided as a set of discrete data points, we first interpolate the data to obtain a continuous function suitable for integration. In particular, we use cubic spline interpolation, implemented in Python using the following command

```
Interpolated_Pk = CubicSpline(k, Pk)
```

we obtain the result shown in the right panel of Fig. 4.

Now, using Simpson's method, described in the previous section, we numerically solve Eq. (18) to compute the correlation function $\xi(r)$ in the range $r \in [50, 120]\,\mathrm{Mpc}$, integrating in $k \in [10^{-4}, 10^3]$ and choosing $N = 140000$. Then we get

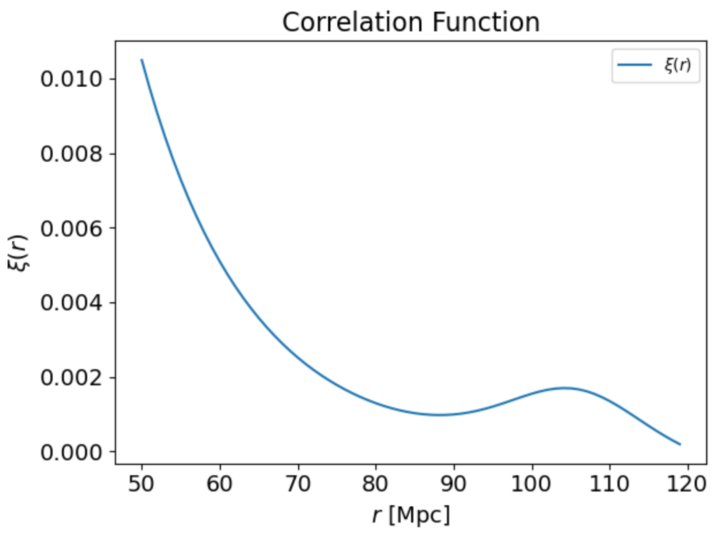

Figure 5: Correlation function $\xi(r)$ as a function of r.

From the plot, we observe a bump appearing in the region $r > 90\,\mathrm{Mpc}$. A more effective way to visualize the position of this peak is by plotting $r^2\xi(r)$, which enhances the visibility of the bump. The resulting plot, restricted to the same r-range, is shown below:

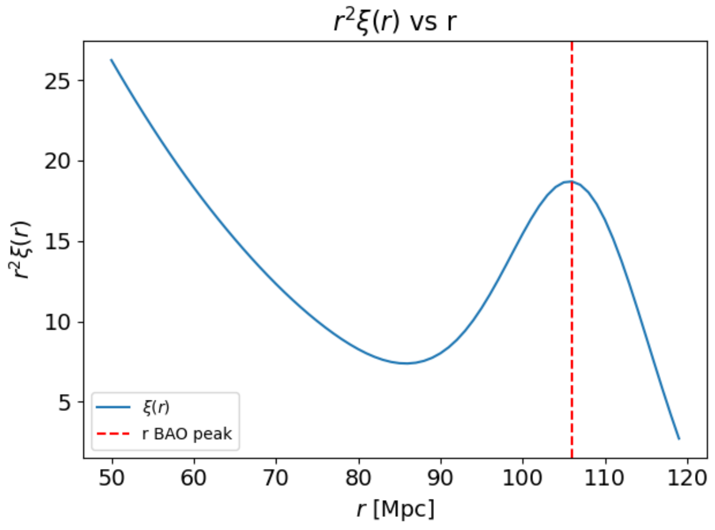

Figure 6: Plot of $r^2\xi(r)$, highlighting the position of the bump.

From this plot, the peak is more clearly defined and is located at:

$$r_{\mathrm{peak}} \simeq 106\,\mathrm{Mpc}. \tag{19}$$

An important point to highlight is the following. In order to compute the correlation function plotted in the previous figures, we evaluated the integral in Eq. (18) over the entire k-axis, specifically $k \in [10^{-4}, 10^3]$. If instead we restrict the integration to a smaller range, such as $k \in [10^{-4}, 10^1]$, the resulting correlation function exhibits an irregular and less smooth behavior. This is illustrated in the following figure:

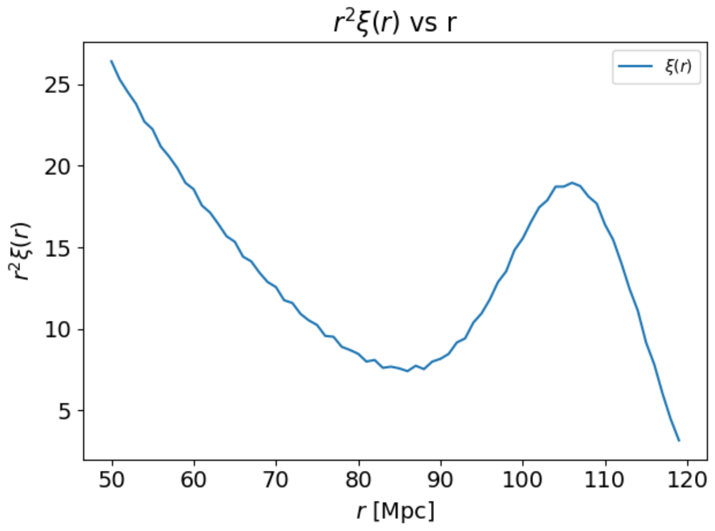

Figure 7: Plot of $r^2\xi(r)$, obtained by integrating over the limited range $k \in [10^{-4}, 10^1]$.

This behavior arises because the integration omits contributions from higher k, which significantly affect the shape of the correlation function at smaller scales. By extending the integration range to $k \in [10^{-4}, 10^3]$, we properly include these contributions, resulting in a smooth and physically accurate correlation function, as shown in Fig. 6.

5 Conclusions

In this homework, we explored how to apply the fundamental concepts of numerical differentiation and integration. In particular, we studied the role of truncation and round-off errors, which dominate in different regimes depending on the size of the parameter h and the number of integration steps N. Additionally, in the final part of the project, we applied these numerical techniques to a physical problem: the computation of the cosmological correlation function. Since the power spectrum data was provided in tabulated form, we used Python's interpolation tools to construct a continuous function. This allowed us to evaluate the integral using Simpson's method and successfully obtain the correlation function in configuration space in order to estimate the BAO peak.

6 Code

The code containing all the results can be found at the following Repository GitHub.