NUR - Hand-in Exercise 2

Martina Cacciola (s4170814)

March 25, 2024

Abstract

In this document, I present the results for the Hand-in Exercise 2 for the course Numerical Recipes for Astrophysics.

1 Satellite galaxies around a massive central

The exercise is done in the script

```
import time
import numpy as np
from math import *
import matplotlib.pyplot as plt

'''
This script containes the code for Exercise 1.
'''

class CombinedRNG:
    '''
    Pseudo-random number generator combining the XOR-shift and Multiply-With-Carry (MWC)
     methods.
    We set the seed to the current time in milliseconds, as a source of variability
    '''
    def __init__(self, seed = None):
        # If no seed is provided, use the current time in milliseconds
        if seed is None:
            seed = int(time.time() * 1000)
        # Initialize the states for XOR-shift and MWC methods
        self.xor_state = seed
        self.mwc_state = seed

    def xor_shift(self):
        # XOR-shift method
        # The seed is converted to a 64-bit unsigned integer
        #By performing a shift of 21 bits to the left and XORing the result with the
    original state
        self.xor_state ^= (self.xor_state << 21) & 0xFFFFFFFFFFFFFFFF
        # Then, a shift of 35 bits to the right and again XOR with the previous result
        self.xor_state ^= (self.xor_state >> 35)
        # Finally, a shift of 4 bits to the left and XOR with the previous result
        self.xor_state ^= (self.xor_state << 4) & 0xFFFFFFFFFFFFFFFF
        return self.xor_state

    def mwc(self):
        # Multiply-With-Carry (MWC) method
        # Set the multiplier
        a = 4294957665
        # Extract the upper and lower 32 bits of the state
        mwc_upper = (self.mwc_state & 0xFFFFFFFF00000000) >> 32
        mwc_lower = self.mwc_state & 0x00000000FFFFFFFF
        # Calculate the next state by multiplying the lower 32 bits with the multiplier
        # and adding the upper 32 bits
        x_next = a * mwc_lower + mwc_upper
        # Update the state using AND operation with a 64-bit mask
```

```python
        # Only the lower 64 bits are kept, the rest are set to zero
        self.mwc_state = x_next & 0xFFFFFFFFFFFFFFFF
        return x_next >> 32

    def combined_rng(self):
        # Combine the XOR-shift and MWC methods by XORing their outputs
        return (self.xor_shift() ^ self.mwc()) & 0xFFFFFFFFFFFFFFFF

    def uniform(self, low, high, num_samples):
        # Generate a list of uniformly distributed random numbers
        # By scaling the output of the combined RNG to the desired range
        return [low + (high - low) * self.combined_rng() / 0xFFFFFFFFFFFFFFFF for _ in
    range(int(num_samples))]

# Create an instance of the combined RNG
rng = CombinedRNG()


## 1a)

# Define the parameters
a = 2.4
b = 0.25
c = 1.6
xmax = 5
Nsat = 100

## 1a)

def integrand(x):
    # Take into account singularity at x=0
    # Consider the volume element is 4*pi*x^2 (4*pi is taken outside)
    if x == 0:
        return 0
    return Nsat * (x/b)**(a-3) * exp(-(x/b)**c) * x**2

# Implemement a numerical integration
# Using the trapezoidal rule
def trapezoidal_rule(f, a, b, n):
    '''
    Inputs:
    f: Function to integrate
    a, b: Integration limits
    n: Number of intervals
    It works by approximating the integral of f(x) between a and b
    by the sum of the areas of trapezoids formed by the function and the x-axis
    '''
    # Width of each trapezoid
    h = (b - a) / n
    # Initialize the result with the average of the function at the limits
    result = 0.5 * (f(a) + f(b))
    for i in range(1, n):
        # Add the value of the function at each interior point
        #Each of the function values corresponds to the height of a trapezoid
        result += f(a + i * h)
    # Multiply the sum by the width of the trapezoids
    # This gives the total area under the curve
    result *= h
    return result


# Romberg integration
def romberg(f, a, b, n):
    '''
    Inputs:
    f: Function to integrate
    a, b: Integration limits
    n: Number of rows in the Romberg table
    '''
    R = np.zeros((n, n))
    # At each iteration, calculates an approximation of the integral
    # using the trapezoidal rule with 2 ** i intervals
    # This forms the first column of the Romberg table
    for i in range(n):
        R[i, 0] = trapezoidal_rule(f, a, b, 2 ** i)
    # At each iteration, calculates an improved approximation of the integral
    # using Richardson extrapolation
    # Takes a weighted average of the current approximation and the previous one
    # The weights are chosen to cancel out as much of the error as possible
    # This fills the rest of the Romberg table
    for j in range(1, n):
        for k in range(j, n):
            R[k, j] = R[k, j - 1] + (R[k, j - 1] - R[k - 1, j - 1]) / (4 ** j - 1)
    return R[-1, -1] # Return the last element of the last row of the Romberg table


# Apply the method above to the integral
n = 10
m = 6
R  = romberg(integrand, 0, xmax, m)
result = R * 4 * pi
A = Nsat / result
with open("normalization.txt", "w") as file:
    file.write("The result of the numerical integration is: " + str(A))


## 1b)

# Define the number density of galaxies
# This represents the number of galaxies per unit volume at a given radius x
def n(x):
    return A * Nsat * (x/b)**(a-3) * exp(-(x/b)**c)

# Define the number of galaxies N(x) in a shell of radius x and thickness dx
# We compute N(x) by multiplying the number density n(x) by the volume element 4*pi*x^2
def N(x):
    return 4 * np.pi * x**2 * n(x)

# Define the probability distribution function p(x) given that p(x)dx = N(x)dx / Nsat
# It gives the probability of finding a galaxy at a given radius x
def p_x(x):
    return N(x) / Nsat


# Implement inverse transform sampling to sample from the distribution
def inverse_transform_sampling(pdf, n_samples, x_min, x_max):
    x_values = np.linspace(x_min, x_max, 10000)
    cdf_values = np.zeros_like(x_values)
    # Calculate the cumulative distribution function using numerical integration (
    trapezoidal rule)
    # The cdf at a point x is the integral of the pdf from x_min to x
    for i in range(1, len(x_values)):
        cdf_values[i] = cdf_values[i-1] + trapezoidal_rule(pdf, x_values[i-1], x_values[
    i], 1)
    cdf_values /= cdf_values[-1]  # Normalize to ensure CDF ranges from 0 to 1

    # Generate random numbers uniformly distributed between 0 and 1
    random_numbers = rng.uniform(0, 1, num_samples=n_samples)

    # Apply the inverse CDF to the random numbers
    # For each random number, find the corresponding value of x
    # such that the CDF of x is equal to the random number
    # This is done by finding the first x value for which the CDF is greater than the
    random number
    # The corresponding x value is then stored as a sampled point
    sampled_points = np.zeros(n_samples)
    for i in range(n_samples):
        for j in range(len(cdf_values)):
            if random_numbers[i] < cdf_values[j]:
                sampled_points[i] = x_values[j]
                break

    return sampled_points

# Generate 10,000 sampled points
n_samples = 10000
x_min, x_max = 10**-4, 5
h = (x_max - x_min) / n_samples
sampled_points = inverse_transform_sampling(p_x, n_samples, x_min, x_max)

# Calculate the bin edges
bin_edges = np.logspace(np.log10(x_min), np.log10(x_max), 21)

# Calculate histogram and divide each bin by its width
hist, _ = np.histogram(sampled_points, bins=bin_edges, density=False)
bin_widths = np.diff(bin_edges)
hist_density = hist / (bin_widths * n_samples / Nsat)  # Correcting for the
    normalization offset

# Plot the analytical function N(x) and the histogram of the sampled points on a log-log
     scale
x_values = np.logspace(-4, np.log10(x_max), 20)
N_x_values = [N(x) for x in x_values]


plt.figure(figsize=(10, 6))
plt.loglog(x_values, N_x_values, label='Analytical N(x)', color='r')
plt.bar(bin_edges[:-1], hist_density, width=bin_widths, label='Sampled Points Histogram'
    )
plt.xlabel('Relative Radius x')
plt.ylabel('Number of Galaxies N(x)')
plt.ylim(10**-3, 10**5)
plt.legend()
plt.title('Analytical N(x) vs Sampled Points Histogram')
plt.grid(True)
plt.savefig('my_solution_1b.png', dpi=600)

## 1c)

# Define a selection method following these rules:
# Select each galaxy with same probability
# Not draw the same galaxy twice
# Not reject any drawn galaxy
def reservoir_selection(sampled_points, n_galaxies):
    selected_galaxies = []
    num_samples = len(sampled_points)
    for i in range(num_samples):
        # If we haven't selected n_galaxies yet, add the current galaxy
        if i < n_galaxies:
            selected_galaxies.append(sampled_points[i])
        else:
            # If we have already selected n_galaxies, decide whether to replace one of
    them
            # Generate a random index j between 0 and i
            j = rng.combined_rng() % (i + 1)
            # If j is less than n_galaxies, replace the galaxy at index j with the
    current galaxy
            if j < n_galaxies:
                selected_galaxies[j] = sampled_points[i]
    return selected_galaxies

# Define a sorting method (quicksort)
def quick_sort(arr):
    # Base case: if the array is empty or has only one element, it is already sorted
    if len(arr) <= 1:
        return arr
    # Choose the pivot element as the middle element of the array
    pivot = arr[len(arr) // 2]
    # Partition the array into three parts:
    # elements less, equal and greater than the pivot
    left = [x for x in arr if x < pivot]
    middle = [x for x in arr if x == pivot]
    right = [x for x in arr if x > pivot]
    # Recursively sort the left and right parts
    # then concatenate the three parts to produce the final sorted array
    return quick_sort(left) + middle + quick_sort(right)

# Select 100 galaxies
n_galaxies = 100
selected_galaxies = reservoir_selection(sampled_points, n_galaxies)

# Sort the galaxies using quicksort from smallest to higher radius
sorted_galaxies = quick_sort(selected_galaxies)

# Plot the cumulative number of the chosen galaxies
fig1c, ax = plt.subplots()
ax.plot(sorted_galaxies, np.arange(n_galaxies))
ax.set(xscale='log', xlabel='Relative radius',
       ylabel='Cumulative number of galaxies',
       xlim=(x_min, x_max), ylim=(0, n_galaxies))
plt.savefig('my_solution_1c.png', dpi=600)

## 1d)

# Define Ridders' method for numerical differentiation
def ridders_method(f, x, h, m, tol=1e-10):
    # Compute first approx with central difference for high h
    D = [(f(x + h) - f(x - h)) / (2 * h)]

    # Decrease h by a factor of 2 and calculate a new approximation. Repeat until you
    have m approximations
    for i in range(1, m):
        h /= 2
        D.append((f(x + h) - f(x - h)) / (2 * h))
        for j in range(i):
            D[j] = (4**(j+1) * D[j+1] - D[j]) / (4**(j+1) - 1)

    # Terminate when the improvement over previous best approximation is smaller than
    the target error or if error grows
    best_approximation = None
    for i in range(1, len(D)):
        if abs(D[i] - D[i-1]) < tol:
            best_approximation = D[i]
            break

        # Terminate early if the error grows and return the best approximation from
    before that point.
        if abs(D[i] - D[i-1]) > abs(D[i-1]):
            best_approximation = D[i-1]
            break

    if not best_approximation:
        print("Tolerance not reached or error grew significantly. Consider increasing m.
    ")

    return best_approximation


# Calculate the analytical result using derivative function
#Following the formal definition of the derivative
def derivative_n(f, x, h= 1e-10):
    return (f(x + h) - f(x)) / h

# Calculate the numerical result using central difference method
numerical_result = ridders_method(n, 1, 0.1, 15, tol=1e-10)
analytical_result = derivative_n(n, 1, h=1e-10)

# Output the results
with open('derivative.txt', 'w') as f:
    f.write("Analytical result: " + format(analytical_result, '.12f') + "\n")
    f.write("Numerical result: " + format(numerical_result, '.12f') + "\n")
```

sat.py

. The necessary explanations of the methods used are in the comments of the code. For question (a), the normalization factor A we obtain is the following:

```
The result of the numerical integration is: 9.05216643171278
```

normalization.txt

. For question (b), we generate 3D satellite positions that statistically follow the satellite profile $n(x)$. We sample them from the probability density function $p(x)$, using the inverse transform sampling. We compare the distribution of the sampled points with $N(x)$, the analytical function describing the number of galaxies in a shell of thickness dx at a given distance x. From Fig.1, we can see that the sampled points match the expected distribution. Nevertheless, we can see how the samples are not present in the first interval of x values. Inverse sampling works by transforming uniform random numbers into samples from the desired distribution, using the cumulative distribution function (CDF) of the PDF. The CDF maps values from the sample space to the interval [0, 1], and its inverse can be used to map values from [0, 1] back to the sample space. In the case of smaller values, a small range of uniform random numbers can map to a large range of small values in the sample space, leading to undersampling in these regions. This might explain why there is a discrepancy at smaller values in our histogram.

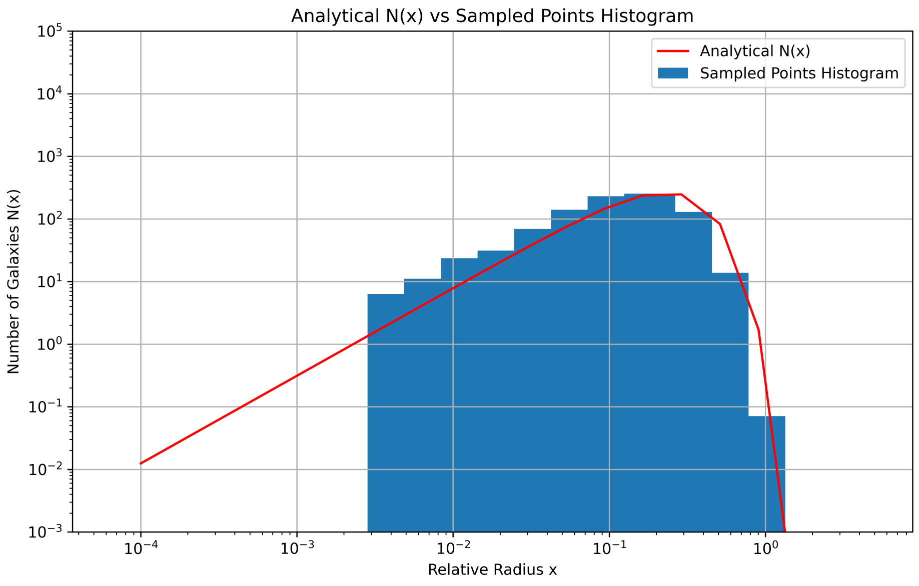

Figure 1: Plot in logarithmic scale showing $N(x)$, function of the number of galaxies at given distance x, and the histogram of the 10000 sampled points. The sampled points match the analytical distribution, but there is a problem of undersampling in the region of smaller radii.

For question (c), we select 100 random galaxies from the ones sampled in point (b), using Reservoir method. It is a sampling algorithm that chooses a random sample, without replacement, of k items from a population of unknown size n in a single pass over the items. The 100 drawn galaxies are ordered from smallest to largest radius. The number of galaxies within a radius x are shown in Fig. 2. The radii selected are starting from higher values because of the previous undersampling. The increasing trend makes sense as the number of galaxies within a given radius should increase as the radius increases.

For question (d), we compute the derivative of the function $n(x)$ using both the Ridders' method and the analytical procedure. The obtained results can be seen in:

```
Analytical result: −0.615624912159
```

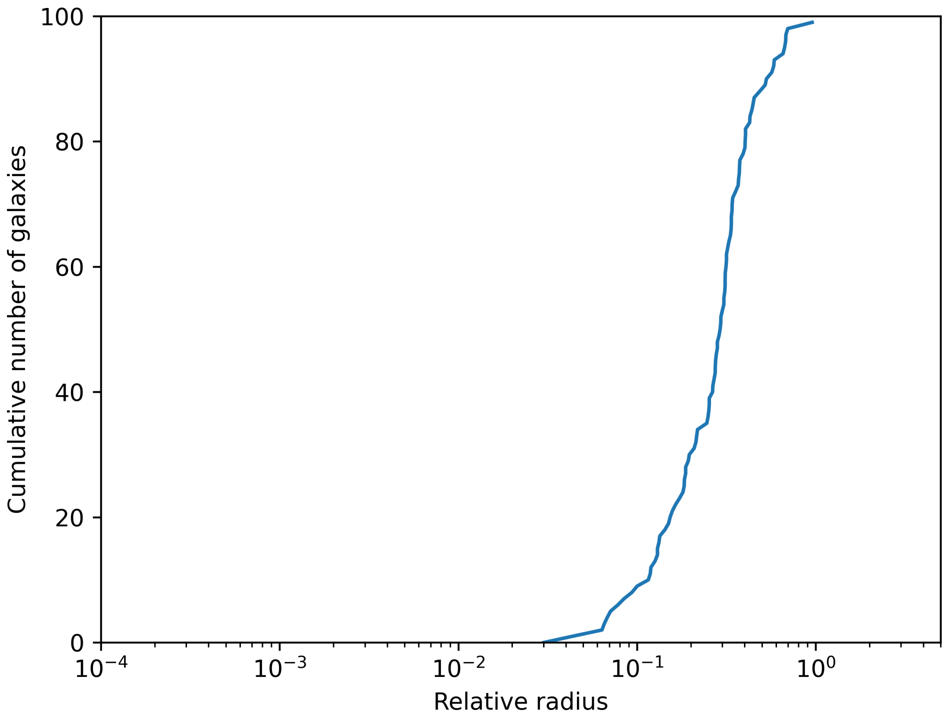

Figure 2: Plot showing the cumulative number of galaxies randomly selected with respect to the relative radius x. The number of galaxies progressively increases with larger distances.

```
Numerical result: −0.615624525721
```

derivative.txt

2 Heating and cooling in HII regions