NUR - Hand-in Exercise 2

Martina Cacciola (s4170814)

March 27, 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 satellite.py. The necessary explanations of the methods used are in the comments of the code. For the purpose of this task, we implement a Random Number Generator (RNG) as follows:

```
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
        # 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
```

```
43
44      def uniform(self, low, high, num_samples):
45          # Generate a list of uniformly distributed random numbers
46          # By scaling the output of the combined RNG to the desired range
47          return [low + (high - low) * self.combined_rng() / 0xFFFFFFFFFFFFFFFF for _ in
    range(int(num_samples))]
48
49  # Create an instance of the combined RNG
50  rng = CombinedRNG()
```

satellite.py

For question (a), we do the following:

```
1   ## 1a)
2
3   # Define the parameters
4   a = 2.4
5   b = 0.25
6   c = 1.6
7   xmax = 5
8   Nsat = 100
9
10  def integrand(x):
11      # Take into account singularity at x=0
12      # Consider the volume element is 4*pi*x^2 (4*pi is taken outside)
13      if x == 0:
14          return 0
15      return Nsat * (x/b)**(a-3) * exp(-(x/b)**c) * x**2
16
17  # Implemement a numerical integration
18  # Using the trapezoidal rule
19  def trapezoidal_rule(f, a, b, n):
20      '''
21      Inputs:
22      f: Function to integrate
23      a, b: Integration limits
24      n: Number of intervals
25      It works by approximating the integral of f(x) between a and b
26      by the sum of the areas of trapezoids formed by the function and the x-axis
27      '''
28      # Width of each trapezoid
29      h = (b - a) / n
30      # Initialize the result with the average of the function at the limits
31      result = 0.5 * (f(a) + f(b))
32      for i in range(1, n):
33          # Add the value of the function at each interior point
34          #Each of the function values corresponds to the height of a trapezoid
35          result += f(a + i * h)
36      # Multiply the sum by the width of the trapezoids
37      # This gives the total area under the curve
38      result *= h
39      return result
40
41
42  # Romberg integration
43  def romberg(f, a, b, n):
44      '''
45      Inputs:
46      f: Function to integrate
47      a, b: Integration limits
48      n: Number of rows in the Romberg table
49      '''
50      R = np.zeros((n, n))
51      # At each iteration, calculates an approximation of the integral
52      # using the trapezoidal rule with 2 ** i intervals
53      # This forms the first column of the Romberg table
54      for i in range(n):
55          R[i, 0] = trapezoidal_rule(f, a, b, 2 ** i)
56      # At each iteration, calculates an improved approximation of the integral
57      # 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))
```

satellite.py

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 distribution $p(x)$, using the inverse transform sampling. We do this with the following code:

```
## 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)


    # 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('./plots/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)
# Takes an array and two indices: low (starting index of the array) and high (last index
    )
def quick_sort(arr, low, high):
    # If the low index is less than the high index, there are elements in the array to
```

```
    be sorted
    if low < high:
        # partition function called to partition the array around a pivot.
        # The pivot's final position in the sorted array is returned as pi
        pi = partition(arr, low, high)

        # quick_sort function recursively called for the parts of the array before pi and after pi
        quick_sort(arr, low, pi - 1)
        quick_sort(arr, pi + 1, high)
    return arr

# This function partitions the array around a pivot
def partition(arr, low, high):
    # Choose the pivot as the median of the first, middle, and last elements
    mid = low + (high - low) // 2
    pivot = arr[mid]

    # If the first element is greater than the middle element, swap them
    if arr[low] > arr[mid]:
        arr[low], arr[mid] = arr[mid], arr[low]

    # If the middle element is greater than the last element, swap them
    if arr[mid] > arr[high]:
        arr[mid], arr[high] = arr[high], arr[mid]

    # If the first element is greater than the middle element, swap them
    if arr[low] > arr[mid]:
        arr[low], arr[mid] = arr[mid], arr[low]

    # Now the middle element is the median of the first, middle, and last elements
    pivot = arr[mid]

    # Swap the pivot with the high element
    arr[mid], arr[high] = arr[high], arr[mid]

    # i is set to one index less than the low index
    i = (low - 1)
    # The array is iterated from the low to the high index
    for j in range(low , high):
        # If the element is less than or equal to the pivot, the element is swapped with the one at index i + 1
        if arr[j] <= pivot:
            i = i + 1
            arr[i], arr[j] = arr[j], arr[i]

    # After all elements have been checked, the pivot is swapped with the element at index i + 1
    # placing the pivot in its correct position in the sorted array
    arr[i + 1], arr[high] = arr[high], arr[i + 1]
```

satellite.py

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 our case, we are sampling from $p(x)$, so it is expected for the sampled values to have higher density in regions of higher probability. It is likely to get more samples from the region corresponding to the peak of the distribution, whereas lower values far from the peak have a lower probability of being drawn. This might explain why there is a discrepancy at smaller values in our histogram.

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

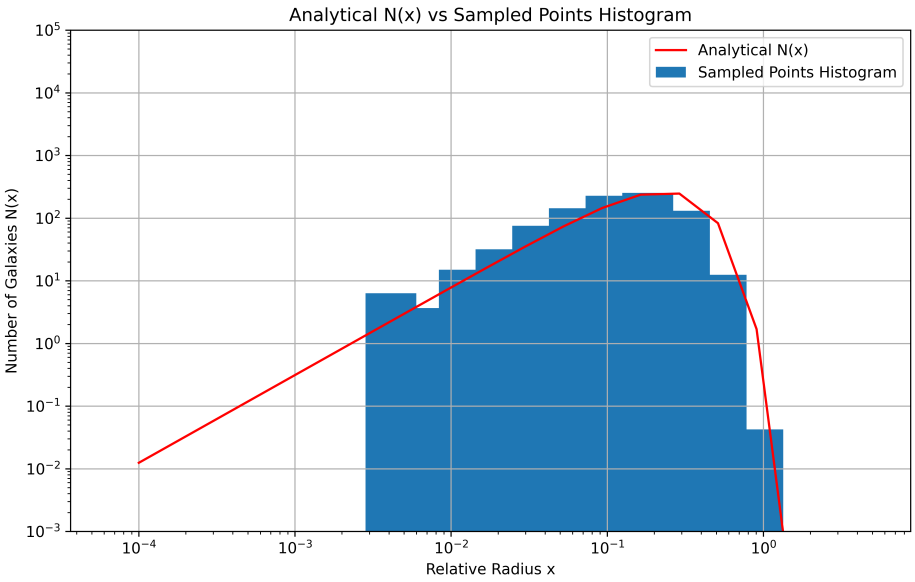

Figure 1: Plot in logarithmic scale showing $N(x)$, function of the number of galaxies at given distance x, and the histogram of the 10000 sampled points. The sampled points match the analytical distribution.

smallest to largest radius. We do this as follow:

```
    return (i + 1)

# 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,0,len(selected_galaxies)-1)

# 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('./plots/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.
    ")
```

satellite.py

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 at lower values. The increasing trend makes sense as the number of galaxies within a given radius should increase as the radius increases.

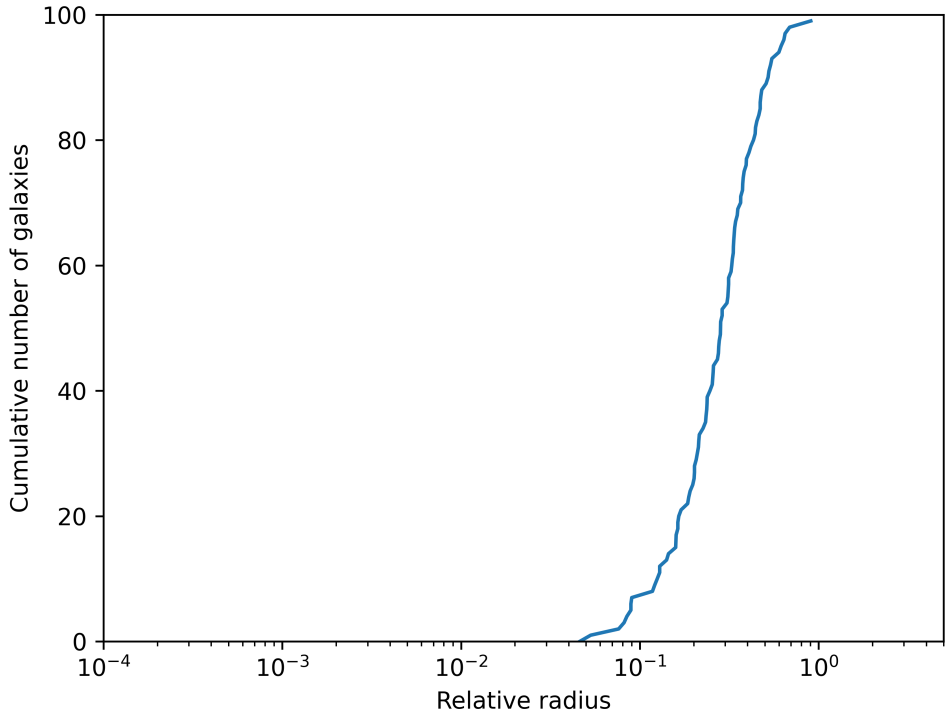

Figure 2: Plot showing the cumulative number of galaxies randomly selected with respect to the relative radius x. The number of galaxies progressively increases with larger distances.

For question (d), we compute the derivative of the function $n(x)$ using both the Ridders' method and the analytical procedure. The code is below:

```

    # Now the middle element is the median of the first, middle, and last elements
    pivot = arr[mid]

    # Swap the pivot with the high element
    arr[mid], arr[high] = arr[high], arr[mid]

    # i is set to one index less than the low index
    i = (low - 1)
    # The array is iterated from the low to the high index
```

```
    for j in range(low , high):
        # If the element is less than or equal to the pivot, the element is swapped with the one at index i + 1
        if arr[j] <= pivot:
            i = i + 1
            arr[i], arr[j] = arr[j], arr[i]

    # After all elements have been checked, the pivot is swapped with the element at index i + 1
    # placing the pivot in its correct position in the sorted array
    arr[i + 1], arr[high] = arr[high], arr[i + 1]
    # Return the final position of the pivot
    return (i + 1)

# 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,0,len(selected_galaxies)-1)

# 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('./plots/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
```

satellite.py

The obtained results can be seen in:

```
Analytical result: -0.615624912159
Numerical result: -0.615624525721
```

derivative.txt

2 Heating and cooling in HII regions

The code relative to this exercise is in the script heating.py. The code relative to point (a) is the following:

```
## 2a)

# Define the equilibrium function
def equilibrium_1(T):
    term1 = psi * Tc * k
    term2 = (0.684 - 0.0416 * np.log(T / (1e4 * Z * Z))) * T * k
    return term1 - term2

# Measure the time taken and find the root of the equilibrium function using the bisection method
def bisection_method(f, a, b, tol=1e-10, max_iter=100):
    '''
    Find the root of the function f(x) = 0 using the bisection method
    Takes the function f, the endpoints of the initial interval [a, b],
    and optional arguments tol and max_iter
    '''
```

```
    if f(a) * f(b) >= 0:
        print("Bisection method fails.")
        return None, None

    # Initialize values
    num_steps = 0
    start_time = time.time()
    while (b - a) / 2 > tol and num_steps < max_iter:
        c = (a + b) / 2
        if f(c) == 0:
            end_time = time.time()
            return c, num_steps + 1, end_time - start_time

        # Update the interval
        if f(c) * f(a) < 0:
            b = c
        else:
            a = c
        num_steps += 1

    end_time = time.time()
    return (a + b) / 2, num_steps, end_time - start_time

# Define the constants
k = 1.38e-16 # erg/K
aB = 2e-13 # cm^3 / s
Z = 0.015 # metallicity
Tc = 1e4 * Z**2 # stellar temperature in K
psi = 0.929

# Measure the time taken and find the root of the equilibrium function using the
    bisection method
a = 1
b = 1e7
T_eq, num_steps, time_taken = bisection_method(equilibrium_1, a, b)


if T_eq is not None:
    output_text = (f"The equilibrium temperature is {T_eq:.2f} K.\n"
                   f"The bisection method found the root in {num_steps} steps.\n"
                   f"The time taken was {time_taken:.10f} seconds.\n")
else:
    output_text = "The bisection method did not converge."

# Writing output to a file
with open("2a.txt", "w") as file:
    file.write(output_text)
```

heating.py

The output of point (a) is in

```
The equilibrium temperature is 3.12 K.
The bisection method found the root in 56 steps.
The time taken was 0.0001101494 seconds.
```

2a.txt

For point (b), we have adopted again the bisection method, which guarantees convergence to a root of the function within a given interval, provided that the function is continuous and changes sign within that interval. It has a linear rate of convergence, which makes it slower with respect to other approaches, e.g. the secant method. We tried also this approach but the time taken for this specific problem was approximately the same and it experienced divergence from the expected root. The code follows the same structure as the one implemented in point (a), now taking into account the density variable n_e. The code is reported below:

```
## 2b)

# Define the constants
```

```python
k = 1.38e-16 # erg/K
aB = 2e-13 # cm^3 / s
Z = 0.015 # metallicity
Tc = 1e4 * Z**2 # stellar temperature in K
psi = 0.929
A = 5e-10 # erg
epsilon_CR = 1e-15 # s^-1

# Define the equilibrium function
def equilibrium_2(T, n_e):
    T4 = T / 1e4
    term1 = 0.54 * T4**0.37 * aB * n_e * n_e * k * T
    term2 = A * n_e * epsilon_CR
    term3 = 8.9e-26 * n_e * T4
    return term1 - term2 - term3

# Bisection method
def bisection_method_ne(f, a, b, n_e, tol=1e-10, max_iter=100):
    # Check if the function values at the interval endpoints have the same sign
    if f(a, n_e) * f(b, n_e) >= 0:
        print("Bisection method fails.")
        return None, None, None

    num_steps = 0
    start_time = time.time()
    # Iterate until convergence or maximum iterations
    while (b - a) / 2 > tol and num_steps < max_iter:
        # Midpoint of the interval
        c = (a + b) / 2
         # Check if the midpoint is already the root
        if f(c, n_e) == 0:
            end_time = time.time()
            return c, num_steps + 1, end_time - start_time

        # Oheck if the root lies between a and c
        if f(c, n_e) * f(a, n_e) < 0:
             # Update the upper bound
            b = c
        else:
            # Update the lower bound
            a = c
        num_steps += 1

    end_time = time.time()
    return (a + b) / 2, num_steps, end_time - start_time

# Densities to consider
densities = [1e-4, 1, 1e4]

with open('2b.txt', 'w') as f:
    for n_e in densities:
        # Initial interval for bisection method
        a_bisection = 1
        b_bisection = 1e15
        T_eq_bisection, num_steps_bisection, time_taken_bisection = bisection_method_ne(equilibrium_2, a_bisection, b_bisection, n_e)

        # Write the result to the file
        if T_eq_bisection is not None:
            f.write(f"For n_e = {n_e} cm^-3, the equilibrium temperature is {T_eq_bisection:.2f} K.\n")
            f.write(f"The bisection method found the root in {num_steps_bisection} steps.\n")
            f.write(f"The time taken was {time_taken_bisection:.6f} seconds.\n\n")
        else:
            f.write(f"For n_e = {n_e} cm^-3, the bisection method failed to converge.\n\n")
```

heating.py

The output of (b) is in

```
For n_e = 0.0001 cm^-3, the equilibrium temperature is 160561340829973.25 K.
The bisection method found the root in 54 steps.
The time taken was 0.000037 seconds.

For n_e = 1 cm^-3, the equilibrium temperature is 34243.13 K.
The bisection method found the root in 83 steps.
The time taken was 0.000057 seconds.

For n_e = 10000.0 cm^-3, the equilibrium temperature is 29.12 K.
The bisection method found the root in 83 steps.
The time taken was 0.000052 seconds.
```

2b.txt

These results represent the equilibrium temperature of the ionized gas in HII regions. This quantity is determined by the balance between heating and cooling processes, which involve various physical phenomena such as free-free emission ($\Lambda_{\rm FF}$), heating by cosmic rays ($\Gamma_{\rm CR}$), and MHD waves (Γ_{MHD}).

- For **low-density** gas ($n_{\rm e} = 10^{-4}\,{\rm cm}^{-3}$), the equilibrium temperature is extremely high (1.60×10^{14} K). This is because, at low densities, the cooling processes (like free-free emission which depends on the square of the density) are inefficient, and the gas remains hot.
- For **intermediate-density** gas ($n_{\rm e} = 1\,{\rm cm}^{-3}$), the equilibrium temperature drops to 34243.13 K. At these densities, the cooling processes become more efficient, and the gas can cool down to lower temperatures.
- For **high-density** gas ($n_{\rm e} = 10^{4}\,{\rm cm}^{-3}$), the equilibrium temperature is quite low (29.12 K). At high densities, the cooling processes are very efficient, and the gas can cool down to very low temperatures.