NUR - Hand-in Exercise 2

Martina Cacciola (s4170814)

March 26, 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:

```python
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
```

```python
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:

```python
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
```

2

```
58        # Takes a weighted average of the current approximation and the previous one
59        # The weights are chosen to cancel out as much of the error as possible
60        # This fills the rest of the Romberg table
61        for j in range(1, n):
62            for k in range(j, n):
63                R[k, j] = R[k, j - 1] + (R[k, j - 1] - R[k - 1, j - 1]) / (4 ** j - 1)
64        return R[-1, -1] # Return the last element of the last row of the Romberg table
65
66
67 # Apply the method above to the integral
68 n = 10
69 m = 6
70 R  = romberg(integrand, 0, xmax, m)
71 result = R * 4 * pi
72 A = Nsat / result
73 with open("normalization.txt", "w") as file:
74     file.write("The result of the numerical integration is: " + str(A))
```

satellite.py

The normalization factor A we obtain is the following:

```
1 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 do this with the following code:

```
1 ## 1b)
2
3 # Define the number density of galaxies
4 # This represents the number of galaxies per unit volume at a given radius x
5 def n(x):
6     return A * Nsat * (x/b)**(a-3) * exp(-(x/b)**c)
7
8 # Define the number of galaxies N(x) in a shell of radius x and thickness dx
9 # We compute N(x) by multiplying the number density n(x) by the volume element 4*pi*x^2
10 def N(x):
11     return 4 * np.pi * x**2 * n(x)
12
13 # Define the probability distribution function p(x) given that p(x)dx = N(x)dx / Nsat
14 # It gives the probability of finding a galaxy at a given radius x
15 def p_x(x):
16     return N(x) / Nsat
17
18
19 # Implement inverse transform sampling to sample from the distribution
20 def inverse_transform_sampling(pdf, n_samples, x_min, x_max):
21     x_values = np.linspace(x_min, x_max, 10000)
22     cdf_values = np.zeros_like(x_values)
23     # Calculate the cumulative distribution function using numerical integration (
        trapezoidal rule)
24     # The cdf at a point x is the integral of the pdf from x_min to x
25     for i in range(1, len(x_values)):
26         cdf_values[i] = cdf_values[i-1] + trapezoidal_rule(pdf, x_values[i-1], x_values[
        i], 1)
27     cdf_values /= cdf_values[-1]  # Normalize to ensure CDF ranges from 0 to 1
28
29     # Generate random numbers uniformly distributed between 0 and 1
30     random_numbers = rng.uniform(0, 1, num_samples=n_samples)
31
32     # Apply the inverse CDF to the random numbers
33     # For each random number, find the corresponding value of x
34     # such that the CDF of x is equal to the random number
35     # This is done by finding the first x value for which the CDF is greater than the
        random number
36     # The corresponding x value is then stored as a sampled point
37     sampled_points = np.zeros(n_samples)
```

```
38      for i in range(n_samples):
39          for j in range(len(cdf_values)):
40              if random_numbers[i] < cdf_values[j]:
41                  sampled_points[i] = x_values[j]
42                  break
43
44      return sampled_points
45
46 # Generate 10,000 sampled points
47 n_samples = 10000
48 x_min, x_max = 10**-4, 5
49 h = (x_max - x_min) / n_samples
50 sampled_points = inverse_transform_sampling(p_x, n_samples, x_min, x_max)
51
52 # Calculate the bin edges
53 bin_edges = np.logspace(np.log10(x_min), np.log10(x_max), 21)
54
55 # Calculate histogram and divide each bin by its width
56 hist, _ = np.histogram(sampled_points, bins=bin_edges, density=False)
57 bin_widths = np.diff(bin_edges)
58 hist_density = hist / (bin_widths * n_samples / Nsat)  # Correcting for the
       normalization offset
59
60 # Plot the analytical function N(x) and the histogram of the sampled points on a log-log
         scale
61 x_values = np.logspace(-4, np.log10(x_max), 20)
62 N_x_values = [N(x) for x in x_values]
63
64
65 plt.figure(figsize=(10, 6))
66 plt.loglog(x_values, N_x_values, label='Analytical N(x)', color='r')
67 plt.bar(bin_edges[:-1], hist_density, width=bin_widths, label='Sampled Points Histogram'
       )
68 plt.xlabel('Relative Radius x')
69 plt.ylabel('Number of Galaxies N(x)')
70 plt.ylim(10**-3, 10**5)
71 plt.legend()
72 plt.title('Analytical N(x) vs Sampled Points Histogram')
73 plt.grid(True)
74 plt.savefig('my_solution_1b.png', dpi=600)
75
76 ## 1c)
77
78 # Define a selection method following these rules:
79 # Select each galaxy with same probability
80 # Not draw the same galaxy twice
81 # Not reject any drawn galaxy
82 def reservoir_selection(sampled_points, n_galaxies):
83     selected_galaxies = []
84     num_samples = len(sampled_points)
85     for i in range(num_samples):
86         # If we haven't selected n_galaxies yet, add the current galaxy
87         if i < n_galaxies:
88             selected_galaxies.append(sampled_points[i])
89         else:
90             # If we have already selected n_galaxies, decide whether to replace one of
       them
91             # Generate a random index j between 0 and i
92             j = rng.combined_rng() % (i + 1)
93             # If j is less than n_galaxies, replace the galaxy at index j with the
       current galaxy
94             if j < n_galaxies:
95                 selected_galaxies[j] = sampled_points[i]
96     return selected_galaxies
97
98 # Define a sorting method (quicksort)
99 # Takes an array and two indices: low (starting index of the array) and high (last index
       )
100 def quick_sort(arr, low, high):
101     # If the low index is less than the high index, there are elements in the array to
```

```
             be sorted
102         if low < high:
103             # partition function called to partition the array around a pivot.
104             # The pivot's final position in the sorted array is returned as pi
105             pi = partition(arr, low, high)
106
107             # quick_sort function recursively called for the parts of the array before pi
        and after pi
108             quick_sort(arr, low, pi - 1)
109             quick_sort(arr, pi + 1, high)
110         return arr
111
112 # This function partitions the array around a pivot
113 def partition(arr, low, high):
114     # Choose the pivot as the median of the first, middle, and last elements
115     mid = low + (high - low) // 2
116     pivot = arr[mid]
117
118     # If the first element is greater than the middle element, swap them
119     if arr[low] > arr[mid]:
120         arr[low], arr[mid] = arr[mid], arr[low]
121
122     # If the middle element is greater than the last element, swap them
123     if arr[mid] > arr[high]:
124         arr[mid], arr[high] = arr[high], arr[mid]
125
126     # If the first element is greater than the middle element, swap them
127     if arr[low] > arr[mid]:
128         arr[low], arr[mid] = arr[mid], arr[low]
129
130     # Now the middle element is the median of the first, middle, and last elements
131     pivot = arr[mid]
132
133     # Swap the pivot with the high element
134     arr[mid], arr[high] = arr[high], arr[mid]
135
136     # i is set to one index less than the low index
137     i = (low - 1)
138     # The array is iterated from the low to the high index
139     for j in range(low, high):
140         # If the element is less than or equal to the pivot, the element is swapped with
        the one at index i + 1
141         if arr[j] <= pivot:
142             i = i + 1
143             arr[i], arr[j] = arr[j], arr[i]
144
145     # After all elements have been checked, the pivot is swapped with the element at
        index i + 1
146     # placing the pivot in its correct position in the sorted array
147     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 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.

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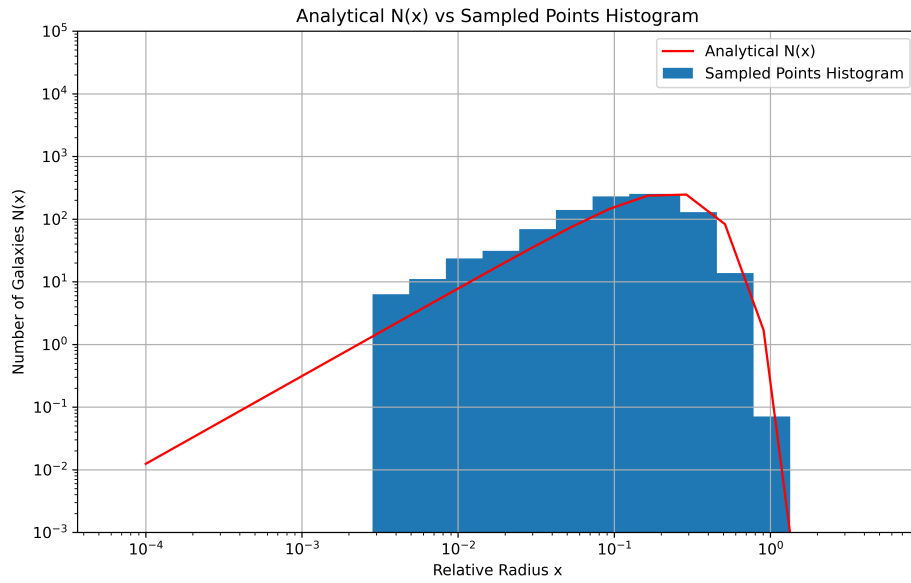
Figure 1: Plot in logarithmic scale showing $N(x)$, function of the number of galaxies at given distance x, and the histogram of the 10000 sampled points. The sampled points match the analytical distribution, but there is a problem of undersampling in the region of smaller radii.

```
1        return (i + 1)
2
3  # Select 100 galaxies
4  n_galaxies = 100
5  selected_galaxies = reservoir_selection(sampled_points, n_galaxies)
6
7  # Sort the galaxies using quicksort from smallest to higher radius
8  sorted_galaxies = quick_sort(selected_galaxies,0,len(selected_galaxies)-1)
9
10 # Plot the cumulative number of the chosen galaxies
11 fig1c, ax = plt.subplots()
12 ax.plot(sorted_galaxies, np.arange(n_galaxies))
13 ax.set(xscale='log', xlabel='Relative radius',
14         ylabel='Cumulative number of galaxies',
15         xlim=(x_min, x_max), ylim=(0, n_galaxies))
16 plt.savefig('my_solution_1c.png', dpi=600)
17
18 ## 1d)
19
20 # Define Ridders' method for numerical differentiation
21 def ridders_method(f, x, h, m, tol=1e-10):
22     # Compute first approx with central difference for high h
23     D = [(f(x + h) - f(x - h)) / (2 * h)]
24
25     # Decrease h by a factor of 2 and calculate a new approximation. Repeat until you
       have m approximations
26     for i in range(1, m):
27         h /= 2
28         D.append((f(x + h) - f(x - h)) / (2 * h))
29         for j in range(i):
30             D[j] = (4**(j+1) * D[j+1] - D[j]) / (4**(j+1) - 1)
31
32     # Terminate when the improvement over previous best approximation is smaller than
       the target error or if error grows
33     best_approximation = None
34     for i in range(1, len(D)):
35         if abs(D[i] - D[i-1]) < tol:
```

```
36              best_approximation = D[i]
37              break
38
39          # Terminate early if the error grows and return the best approximation from
        before that point.
40          if abs(D[i] - D[i-1]) > abs(D[i-1]):
41              best_approximation = D[i-1]
42              break
43
44      if not best_approximation:
45          print("Tolerance not reached or error grew significantly. Consider increasing m.
        ")
```

<div align="center">satellite.py</div>

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.
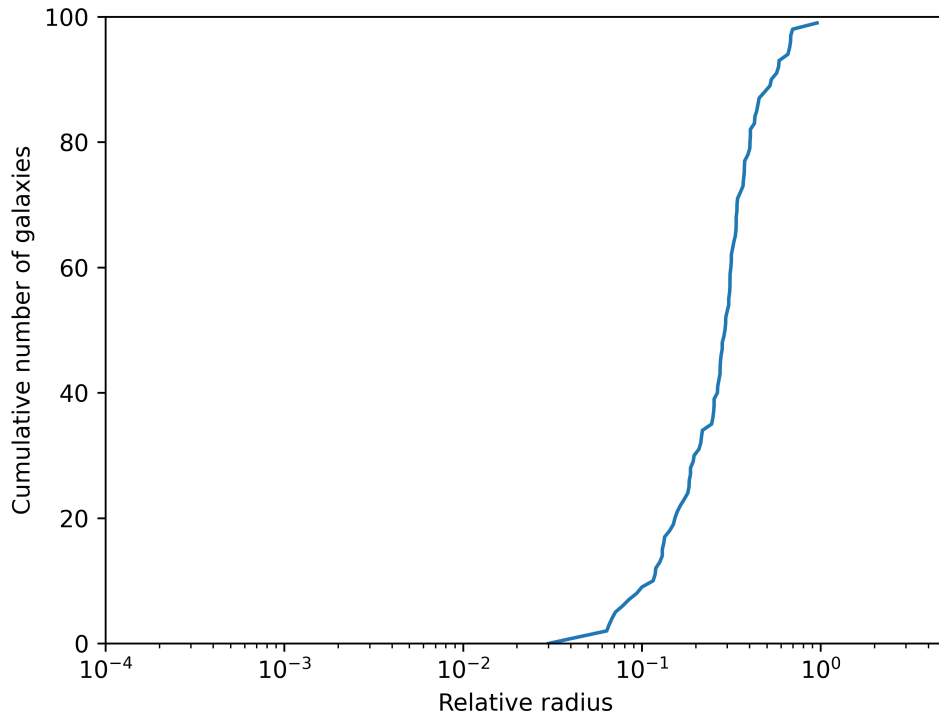

Figure 2: Plot showing the cumulative number of galaxies randomly selected with respect to the relative radius x. The number of galaxies progressively increases with larger distances.

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:

```
1
2      # Now the middle element is the median of the first, middle, and last elements
3      pivot = arr[mid]
4
5      # Swap the pivot with the high element
6      arr[mid], arr[high] = arr[high], arr[mid]
7
8      # i is set to one index less than the low index
9      i = (low - 1)
10     # The array is iterated from the low to the high index
```

<div align="center">7</div>

```python
11        for j in range(low, high):
12            # If the element is less than or equal to the pivot, the element is swapped with
       the one at index i + 1
13            if arr[j] <= pivot:
14                i = i + 1
15                arr[i], arr[j] = arr[j], arr[i]
16
17        # After all elements have been checked, the pivot is swapped with the element at
       index i + 1
18        # placing the pivot in its correct position in the sorted array
19        arr[i + 1], arr[high] = arr[high], arr[i + 1]
20        # Return the final position of the pivot
21        return (i + 1)
22
23 # Select 100 galaxies
24 n_galaxies = 100
25 selected_galaxies = reservoir_selection(sampled_points, n_galaxies)
26
27 # Sort the galaxies using quicksort from smallest to higher radius
28 sorted_galaxies = quick_sort(selected_galaxies, 0, len(selected_galaxies)-1)
29
30 # Plot the cumulative number of the chosen galaxies
31 fig1c, ax = plt.subplots()
32 ax.plot(sorted_galaxies, np.arange(n_galaxies))
33 ax.set(xscale='log', xlabel='Relative radius',
34         ylabel='Cumulative number of galaxies',
35         xlim=(x_min, x_max), ylim=(0, n_galaxies))
36 plt.savefig('my_solution_1c.png', dpi=600)
37
38 ## 1d)
39
40 # Define Ridders' method for numerical differentiation
41 def ridders_method(f, x, h, m, tol=1e-10):
42     # Compute first approx with central difference for high h
43     D = [(f(x + h) - f(x - h)) / (2 * h)]
44
45     # Decrease h by a factor of 2 and calculate a new approximation. Repeat until you
       have m approximations
```

<center>satellite.py</center>

The obtained results can be seen in:

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

<center>derivative.txt</center>

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:

```python
1 ## 2a)
2
3 # Define the equilibrium function
4 def equilibrium_1(T):
5     term1 = psi * Tc * k
6     term2 = (0.684 - 0.0416 * np.log(T / (1e4 * Z * Z))) * T * k
7     return term1 - term2
8
9 # Measure the time taken and find the root of the equilibrium function using the
       bisection method
10 def bisection_method(f, a, b, tol=1e-10, max_iter=100):
11     '''
12     Find the root of the function f(x) = 0 using the bisection method
13     Takes the function f, the endpoints of the initial interval [a, b],
14     and optional arguments tol and max_iter
15     '''
```

<center>8</center>

```
16    if f(a) * f(b) >= 0:
17        print("Bisection method fails.")
18        return None, None
19
20    # Initialize values
21    num_steps = 0
22    start_time = time.time()
23    while (b - a) / 2 > tol and num_steps < max_iter:
24        c = (a + b) / 2
25        if f(c) == 0:
26            end_time = time.time()
27            return c, num_steps + 1, end_time - start_time
28
29        # Update the interval
30        if f(c) * f(a) < 0:
31            b = c
32        else:
33            a = c
34        num_steps += 1
35
36    end_time = time.time()
37    return (a + b) / 2, num_steps, end_time - start_time
38
39 # Define the constants
40 k = 1.38e-16 # erg/K
41 aB = 2e-13 # cm^3 / s
42 Z = 0.015 # metallicity
43 Tc = 1e4 * Z**2 # stellar temperature in K
44 psi = 0.929
45
46 # Measure the time taken and find the root of the equilibrium function using the
        bisection method
47 a = 1
48 b = 1e7
49 T_eq, num_steps, time_taken = bisection_method(equilibrium_1, a, b)
50
51
52 if T_eq is not None:
53     output_text = (f"The equilibrium temperature is {T_eq:.2f} K.\n"
54                     f"The bisection method found the root in {num_steps} steps.\n"
55                     f"The time taken was {time_taken:.10f} seconds.\n")
56 else:
57     output_text = "The bisection method did not converge."
58
59 # Writing output to a file
60 with open("2a.txt", "w") as file:
61     file.write(output_text)
```

<div align="center">heating.py</div>

The output of point (a) is in

```
1 The equilibrium temperature is 3.12 K.
2 The bisection method found the root in 56 steps.
3 The time taken was 0.0001060963 seconds.
```

<div align="center">2a.txt</div>

For point (b), we have adopt 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 to 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 output of (b) is in

```
1 For n_e = 0.0001 cm^-3, the equilibrium temperature is 160561340829973.25 K.
2 The bisection method found the root in 54 steps.
3 The time taken was 0.000035 seconds.
4
5 For n_e = 1 cm^-3, the equilibrium temperature is 34243.13 K.
6 The bisection method found the root in 83 steps.
```

```
 7  The time taken was 0.000058 seconds.
 8
 9  For n_e = 10000.0 cm^-3, the equilibrium temperature is 29.12 K.
10  The bisection method found the root in 83 steps.
11  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 (Λ_{FF}), heating by cosmic rays (Γ_{CR}), and MHD waves (Γ_{MHD}).

- For **low-density** gas ($n_e = 10^{-4} cm^{-3}$), the equilibrium temperature is extremely high ($1.60 \times 10^{14} K$). 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 ($n_e = 1 cm^{-3}$), 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 ($n_e = 10^4 cm^{-3}$), 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.