Numerical Recipes: Hand-In 2

Rens Kievit (s1948415)

March 29, 2023

Abstract

In this report we present the problems, solutions and scripts for the exercises from the second handout for the course Numerical Recipes.

Plotting styles in this report are set using the following code

```
def set_styles():
    """For consistent plotting scheme"""
    plt.style.use('default')
    mpl.rcParams['axes.grid'] = True
    plt.style.use('seaborn-darkgrid')
    mpl.rcParams['font.family'] = 'serif'
    mpl.rcParams['lines.linewidth'] = 1.5
    mpl.rcParams['font.size'] = 14
```

Listing 1: Matplotlib Plotting Styles

1 Satellite Galaxies Around a Massive Central

```
# INTEGRATION
def romberg_integration(func, a, b, order, open_formula=False):
    """Integrate a function, func, using Romberg Integration over the interval [a,b]
    This function usually sets h_start = b-a to sample from the widest possible interval.
    If open_formula is set to True, it assumes the function is undefined at either a or b
    and h_start is set to (b-a)/2.
    This function returns the best estimate for the integrand
    """
    # initiate all parameters
    r_array = np.zeros(order)
    h = b - a
    N_p = 1

    # fill in first estimate, don't do this if we cant evaluate at the edges
    if open_formula:
        # First estimate will be with h = (b-a)/2
        start_point = 0
    else:
        r_array[0] = 0.5*h*(func(b) - func(a))
        start_point = 1

    # First iterations to fill out estimates of order m
    for i in range(start_point, order):
        delta = h
        h *= 0.5
        x = a + h

        # Evaluate function at Np points
        for j in range(N_p):
            r_array[i] += func(x)
            x += delta
        # Combine new function evaluations with previous
        r_array[i] = 0.5*(r_array[i-1] + delta*r_array[i])
        N_p *= 2

    # Combine all of our estimations to cancel our error terms
    N_p = 1
    for i in range(1,order):
        N_p *= 4
        for j in range(order-i):
            r_array[j] = (N_p*r_array[j+1] - r_array[j])/(N_p-1)

    return r_array[0]

# TODO: Improve this into a better method
# DISTRIBUTION SAMPLING
def rejection_sampling(func, rng, N,
                       shift_x=lambda x: x,
                       shift_y=lambda x: x,
                       x0 = 4891653):
    """Sample a 1D distribution using rejection sampling. This function generates
    two random numbers using the provided rng. It then assigns the first value as
    'x' and shifts it using the shift_x function, and assigns the second value as
```

```
    'y' and shifts it using the shift_y function. If y<func(x) the point is accepted
    Repeat this until we have N points, and return these
    x0 is used as a starting seed for the rng
    """

    sampled_points = np.zeros(N)
    num_tries = 0 # For performance testing
    for i in range(N):
        not_sampled = True

        # Keep sampling until we find a x,y pair that fits
        while not_sampled:
            numbers, x0 = rng(2, x0=x0, return_laststate=True) # This is now U(0,1)

            x = shift_x(numbers[0])
            y = shift_y(numbers[1])
            num_tries += 1
            print(x, y, func(x))
            if y < func(x):
                sampled_points[i] = x
                not_sampled = False

    print(f'Average No. tries: {num_tries/N:.1f}')
    return sampled_points


# DIFFERENTIATION
def central_difference(func, x, h):
    """Comptue the derivative of a function evaluated at x, with step size h"""
    return (func(x+h) - func(x-h))/(2.*h)

def ridders_equation(D1, D2, j, dec_factor):
    """Ridders Equation used to combine two estimates at different h"""
    j_factor = dec_factor**(2.*(j+1.))
    return (j_factor * D2 - D1)/(j_factor - 1)


def ridders_method(func, x_ar, h_start, dec_factor, target_acc, approx_array_length=15):
    """Compute the derivative of a function at a point, or points x using Ridder's Method
    The function iteratively adds in more estimates at a lower h until it achieves the provided
    target accuracy. It then returns the best estimate, and the uncertainty on this, which is
    defined as the difference between the current and previous best estimates
    """
    derivative_array = np.zeros_like(x_ar, dtype=np.float64)
    unc_array = np.zeros_like(x_ar, dtype=np.float64)

    for ar_idx in range(len(x_ar)):
        x = x_ar[ar_idx]
        # Make this larger if we have not reached our target accuracy yet
        approximations = np.zeros(approx_array_length, dtype=np.float64)
```

```
104         uncertainties = np.zeros(approx_array_length, dtype=np.
    float64)
105         uncertainties[0] = np.inf # set uncertainty arbitrarily
    large for the error improvement comparison
106 
107         h_i = h_start
108         approximations[0] = central_difference(func, x, h_i)
109         best_guess = approximations[0]
110 
111         for i in range(1, approx_array_length):
112             # Add in a new estimation with smaller step size
113             h_i /= dec_factor
114             approximations[i] = central_difference(func, x, h_i)
115             for j in range(i):
116                 # Add the new approximation into the 'tree of
    estimations'
117                 approximations[i-j-1] = ridders_equation(
    approximations[i-j-1], approximations[i-j], j, dec_factor)
118             uncertainties[i] = np.abs(approximations[0] -
    best_guess)
119             # Test if we are below our target accuracy
120             if (uncertainties[i] < target_acc) or (uncertainties[i]
     > uncertainties[i-1]):
121                 derivative_array[ar_idx] = approximations[0]
122                 unc_array[ar_idx] = uncertainties[i]
123                 break
124             else:
125                 best_guess = approximations[0]
126 
127     return derivative_array, unc_array
128 
129 
130 # SORTING
131 def sort_subarrays(a1, a2, k1, k2):
132     """Takes two subarrays 1,2 with indices a1, a2 and values k1,
    k2 and combines these
133     into array with indices a such that the k are sorted in
    ascending order"""
134     N1 = len(a1)
135     N2 = len(a2)
136 
137     # We built up a new instance of our sorting array. This is not
    memory efficient
138     # TODO: Study the algorithm below to try and find a better
    method
139     a_sorted = np.zeros(N1+N2)
140 
141     if N1 == 0:
142         return a2
143     if N2 == 0:
144         return a1
145 
146     # Walk through the left- and right- sub arrays separately
147     idx1 = 0
148     idx2 = 0
149     while True:
150         if k1[idx1] > k2[idx2]:
151             # Then place the second element to the left of the
    first element and take
152             # one step to the right in the right sub-array
153             a_sorted[idx1+idx2] = a2[idx2]
154             idx2 += 1
```

```
155
156             # Then, if there are elements remaining in the right
    array, keep
157             # placing them to the left as long as they're smaller
158             if idx2 < N2: # need this if statement to save us from
    indexing errors in the while
159                 while (k1[idx1] > k2[idx2]):
160                     a_sorted[idx1+idx2] = a2[idx2]
161                     idx2 += 1
162
163                     if idx2 >= (N2):
164                         # No more elements left in the right array,
     we can fill out with the left array
165                         for j in range(idx1, N1):
166                             a_sorted[j+idx2] = a1[j]
167                         return a_sorted
168
169                 # Now the element from the left array is smaller
    than the first remaining element
170                 # from the right array, we can safely place it
171                 a_sorted[idx1+idx2] = a1[idx1]
172                 idx1 += 1
173             else:
174                 # No more elements left in the right sub-array
175                 for j in range(idx1, N1):
176                     a_sorted[j+idx2] = a1[j]
177                 return a_sorted
178
179         else:
180             a_sorted[idx1+idx2] = a1[idx1]
181             idx1 += 1
182
183         # Check if we have reached the end of the left sub-array
184         # If we have, fill out the rest of the array with the right
     sub-array
185         if idx1 == N1:
186             for j in range(idx2, N2):
187                 a_sorted[idx1+j] = a2[j]
188             return a_sorted
189
190
191
192 def merge_sort(a=None, key=None):
193     """Sorts the array or list using merge sort. This function
    iteratively
194     builds up the array from single elements which are sorted by
    sort_subarrays
195     Note, in principle one should only provide either 'a' or 'key':
196
197     - If 'a' is provided, that array is sorted in ascending order,
    and returned
198       by this function
199     - If 'key' is provided, this function returns the indices
    corresponding to the
200       order in which the array would be sorted
201     - If both 'a' and 'key' are provided, this function assumes '
    key' has already
202       been previously shuffled, and just swaps indices of the
    preexisting 'a'
203
204     RETURNS: 'a': numpy array
205     """
```

```
206     if key is not None:
207         key = np.array(key)
208 
209     if a is None and key is not None:
210         a = np.arange(len(key))
211 
212     a = np.array(a)
213     subsize = 1
214     N = len(a)
215     is_sorted = False
216     # Build up the array sorting arrays of increasing subsize
217     while not is_sorted:
218         subsize *= 2
219         if subsize > N:
220             is_sorted = True # After this iteration, the array is
    sorted
221 
222         for i in range(int(np.ceil(N/subsize))):
223             # We need the min(.. , N) to ensure that we do not
    exceed the length of the
224             # array with our indexing
225             subarray1 = a[i*subsize: i*subsize+int(0.5*subsize)] #
    First half of the interval
226             subarray2 = a[i*subsize+int(0.5*subsize): np.min(((i+1)
    *subsize, N))]
227             if key is not None:
228                 key1, key2 = key[subarray1] , key[subarray2]
229                 sorted_sub = sort_subarrays(subarray1, subarray2,
    key1, key2)
230             else:
231                 # we feed in 'subarrayx' twice because if we only
    sort a, a is its own key
232                 sorted_sub = sort_subarrays(subarray1, subarray2,
    subarray1, subarray2)
233 
234             a[i*subsize:subsize*(i+1)] = sorted_sub
235 
236     return a
237 
238 
239 # ROOT FINDING
240 def false_position(func, bracket, target_x_acc=1e-10, target_y_acc
    =1e-10, max_iter=int(1e5)):
241     """Given a function f(x) and a bracket [a,b], this function
    returns
242     a value c within that interval for which f(c) ~= 0 using the
    false
243     position method. Guaranteed to converge, slow but faster than
    bisection.
244     """
245     a, b = bracket
246     fa, fb = func(a), func(b)
247 
248     # Test if the bracket is good
249     if not (fa*fb) < 0:
250         raise ValueError("The provided bracket does not contain a
    root")
251 
252     for i in range(max_iter):
253         print(a, b, fa, fb)
254 
255         c = a - fa*((a-b)/(fa-fb))
```

```
        fc = func(c)

        # Check if we made our precisions
        # x-axis
        if np.abs(b-c) < target_x_acc or np.abs(a-c) < target_x_acc:
            return c, i+1
        # relative y-axis
        if np.abs((fc-fb)/fc) < target_y_acc or np.abs((fc-fa)/fc) < target_y_acc:
            return c, i+1

        if (fa*fc) < 0:
            # Then the new interval becomes a, c
            # keep the number of function calls as low as possible
            b, fb = c, fc
        elif (fc*fb) < 0:
            # Then the new interval becomes c, b
            a, fa = c, fc
        else:
            print("Warning! Bracket might have diverged")
            return c, i+1

    print("Maximum number of iterations reached")
    return c, i
```

Listing 2: Code for all the algorithms used in this exercise

```
def to_int32(x):
    """Takes any integer x, and returns the last 32 bits"""
    binx = bin(np.uint64(x))
    # First two chars are '0b' can ignore these
    if len(binx) > 33:
        bin32 = binx[-32:]
    else:
        bin32 = (32 - len(binx)) * '0' + binx[2:]
    return int(bin32, 2) # The  '2' indicates the val is given in base2

def mwc_base32(x, a):
    """"""
    # Set the first 32 bits to zero
    x = np.uint64(x)
    x = a*(x & np.uint64((2**32 - 1))) + (x >> np.uint64(32))
    return x

def rng_from_mwc(N, x0=1898567, a=4294957665, return_laststate=False):
    """Sample N values using mwc_base32 with starting value x0
    The given value for a is an optimal seed. We use all 64-bits
    to generate the random number, but only return the last 32
    """
    x = np.zeros(N)
    for i in range(N):
        x0 = mwc_base32(x0, a)
        x[i] = to_int32(x0)
    x /= (2**32 - 1) # this ensures we return U(0,1)
    if return_laststate:
        return x, x0
    else:
        return x
```

Listing 3: Code for the random number generator

All dynamically written results from the code corresponding to this section can be found in `results/satellite_galaxies_results.txt`.

In this section we will investigate the spherical distribution of satellite galaxies around a massive central galaxies. Their density distribution n can be described

$$n(x) = A \langle N_{sat} \rangle \left(\frac{x}{b}\right)^{a-3} \exp\left[-\left(\frac{x}{b}\right)^{c}\right] \tag{1}$$

Where we take $a = 2.4$, $b = 0.25$, $c = 1.6$ and $\langle N_{sat} \rangle = 100$. x is the radius relative to the virial radius, i.e. $x \equiv r/r_{vir}$. A is a normalization constant that we do not a priori know, but is set such that the three dimensional spherical from $x = 0$ to $x_{max} = 5$ is equal to the average number of satellites, $\langle N_{sat} \rangle$:

$$\int\int\int_V n(x) dV = \langle N_{sat} \rangle \,. \tag{2}$$

```
    """Density profile of the spherical distribution of
    satellite galaxies around a central as a function of
    x = r/r_vir. The values given come from hand-in 2"""
    return A*Nsat*((x/b)**(a-3))*np.exp(-(x/b)**c)
def full_run():
    # Parameters given in the hand-in
    a = 2.4
    b = 0.25
    c = 1.6
    A = 1 # Need to compute this later
    Nsat = 100
    x_min = 0
    x_max = 5
    N_sample = int(1e4)
    sample_min_x = 1e-4
    nbins = 20
    num_random_sample = 100

    results_txt = "This file contains all results from question 1\n
    "
    set_styles()
    ########
```

Listing 4: Setup code for the first assignment

1.1 Normalization Constant

Looking at equation 1 we can see that it is independent of angle, and only depends on the radial distance. This allows us to transform the integral in equation 2 to a spherical integral over θ, ϕ, x as

$$\begin{aligned}\int\int\int_V n(x)dV &= \int_0^{x_{max}}\int_0^{\pi}\int_0^{2\pi} n(x)x^2\sin(\theta)d\phi d\theta dx \\ &= 4\pi\int_0^{x_{max}} n(x)x^2 dx \\ &= 4\pi\int_0^{x_{max}} x^2 A\left\langle N_{sat}\right\rangle\left(\frac{x}{b}\right)^{a-3}\exp\left[-\left(\frac{x}{b}\right)^c\right]dx\end{aligned}$$

Here we begin by setting $A = 1$, and using the result of this integral to calibrate A such that equation 2 holds. We can evaluate this integral using a simple one dimensional numerical integrator. We apply a Romberg integrator with order 10 to ensure sufficient accuracy. We have to account for the fact that $n(x)$ is not defined at $x = 0$ because $\left(\frac{x}{b}\right)^{a-3}$ causes a division by zero if $a < 3$, which is the case here. To combat this we set $h_{start} = \frac{b-a}{2}$ instead of $h_{start} = b - a$ to avoid evaluating $n(x = 0)$.

We find $\int_V n(x)dV = 10.88$, which means we have to set the normalization constant $A = \frac{\langle N_{sat}\rangle}{10.88} = \frac{100}{10.88} = 9.19$. We will use this value for A throughout the rest of this work.

```
# 1a. Integrate the 3D spherical integral to find A
to_integrate = lambda x: 4*np.pi*x*x* n(x, A, Nsat, a, b, c)
volume_integral = romberg_integration(to_integrate, x_min,
x_max, 10, open_formula=True)
A = Nsat/volume_integral
print(f'The volume integral evaluated from 0 to 5 returns: {
volume_integral:.6f}')
print(f'Therefore we need a normalization constant A = {A:.2f}'
)
results_txt += f"{volume_integral:.2f}\n{A:.2f}\n"
```

Listing 5: Code applying the integration algorithm to the algorithm in this assignment

1.2 Distribution Sampling

We want to sample the 3D distribution of satellites such that they statistically follow the distribution in equation 1. This means that the the probability distribution should be $p(x)dx = N(x)dx/\left\langle N_{sat}\right\rangle$. Here $N(x)dx$ is the number of satellites in a spherical shell of size dx at relative radius x from the center of the massive galaxy. This is equal to the result we saw for the spherical integral in the previous section, thus the probability distribution we want to sample is

$$p(x)dx = x^2 A\left(\frac{x}{b}\right)^{a-3}\exp\left[-\left(\frac{x}{b}\right)^c\right]dx \tag{3}$$

In this work we will use rejection sampling to simulate this distribution. The implementation in this work first generates two random numbers generated using `32-bit multiply with carry`, we call the first number x and shift it to a uniform distribution in log-space in the range $\left[-4,^{10}\log(5)\right]$ (corresponding to the range $\left[10^{-4}, 5\right]$ in linear space), and the second we call y and shift it to the range $[0, 2.68]$ where 2.68 is the maximum of the distribution given in equation 3 as shown in figure 1. If the (x_i, y_i) point falls below the curve described

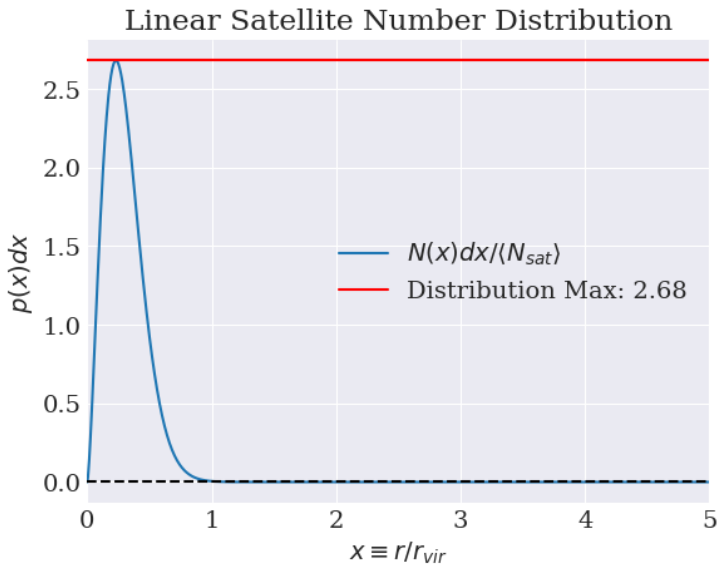

Figure 1: Analytical distribution of the number of satellite galaxies in a shell around a massive galaxy with thickness dr at radius r, described by equation 3.

by equation 3 we accept it, otherwise we reject it[1]. After each step we use the last generated random number as `x0` for the next number. We repeat this entire process, until we have $N = 10^4$ samples in total. We note that rejection sampling is far from the most efficient method to sample a distribution such as this because the majority of possible points in our (x, y) range will be rejected as is evident in figure 1. To slightly combat this we have sampled x in log- instead of linear space which means small values (< 1) are sampled at a higher rate, and the algorithm finds itself in the "interesting" region of the distribution more often. Nevertheless, on average the rejection sampling algorithm still takes $\sim$ 6 attempts (12 random numbers generated) untill it finds an accepted point.

We plot the results from our sampling algorithm as a histogram with 20 equally spaced bins in log-space in figure 2. We divided the height of each bin by 10^3 to account for the difference between $\langle N_{sat} \rangle$ and N. We can see in the figure that the analytical distribution, and our sampled distribution match almost perfectly everywhere except the edges. These edges are missing from this plot because the sampler is often unable to sample points in these regions due to the large value for $p(x)dx$ there.

```
    # 1b. Simulate the distribution
    # Set Nsat = 1 because we divide n(x) by Nsat
    rng = rng_from_mwc
    log_distribution = lambda x: 4*np.pi*10**(2*x)*n(10**x, A, 1, a
    , b, c)

    # Find maximum of this distribution
    x = np.linspace(sample_min_x, x_max, 1000)
    nx = log_distribution(np.log10(x))
```

[1]In the code, we have replaced each x in equation 3 with 10^x to account for the fact we sample in log-space.

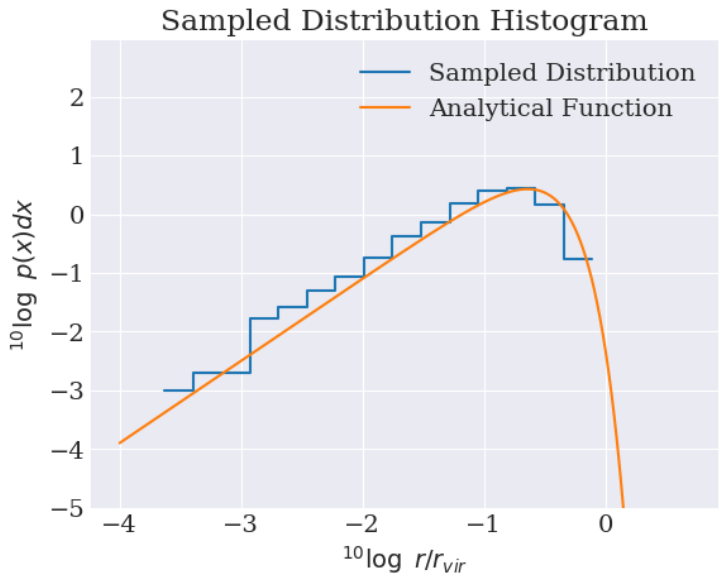

Figure 2: Analytical distribution of satellite galaxies around a massive central overplotted on the same distribution sampled by our rejection algorithm with $N = 10^4$ described in the text. The histgoram bin heights are scaled down by a factor 10^3 to enforce the same normalization.

```
    log_dist_max = np.max(nx)

    # Plot this distribution in linear space for investigation
    purposes
    results_txt += f"{log_dist_max:.2f}\n"
    plt.plot(x, nx, label=r'$N(x)dx/\left<N_{sat}\right>$')
    plt.axhline(y=0, c='black', ls='--')
    plt.axhline(y=log_dist_max, c='red', label=f'Distribution Max:
    {log_dist_max:.2f}')
    plt.xlim(x_min, x_max)
    plt.xlabel(r'$x \equiv r/r_{vir}$')
    plt.ylabel(r'$p(x)dx$')
    plt.title('Linear Satellite Number Distribution')
    plt.legend()
    plt.savefig('results/pxdx.png', bbox_inches='tight')
    plt.clf()

    # Create shift functions to transform U(0,1) to proper
    boundaries
    shift_x = lambda x: x * (np.log10(x_max) - np.log10(
    sample_min_x)) + np.log10(sample_min_x)
    shift_y = lambda y: y * log_dist_max

    print('Sampling Distribution..')
    sampled_points = rejection_sampling(log_distribution, rng,
    N_sample, shift_x=shift_x, shift_y=shift_y)
    sampled_points = 10**sampled_points # Sampled in log space, go
    back to linear

    # Plotting of log distribution + sample
    bin_heights, bin_edges = hist(sampled_points, sample_min_x,
```

```
    x_max, nbins, log=True)
    bin_centers = np.zeros(len(bin_edges)-1)
    for i in range(bin_centers.shape[0]):
        bin_centers[i] = bin_edges[i] + 0.5*(bin_edges[i+1] -
    bin_edges[i])

    plt.step(np.log10(bin_centers), np.log10(bin_heights/1000),
    label='Sampled Distribution')
    plt.plot(np.log10(x), np.log10(nx), label='Analytical Function'
    )
    plt.xlabel(r'$^{10}\log~r/r_{vir}$')
    plt.ylabel(r'$^{10}\log~p(x)dx$')
    plt.title('Sampled Distribution Histogram')
    plt.ylim(bottom=-5)
    plt.legend()
    plt.savefig('results/satellite_galaxies_pdf.png', bbox_inches='
    tight')
    plt.clf()
```

Listing 6: Code to sample the distribution given in this assignment

1.3 Random Selection

We want to look at the cumulative distribution function (CDF) of the sampled probability distribution shown in figure 2. To do this we first select 100 random satellite galaxies following the criteria given on the handout: each galaxy is selected with equal probabilty, no galaxy is selected twice, and we do not reject any draws. The simple (but maybe relatively overboard) method we use to ensure these criteria are upheld is to first shuffle the array of 10^4 galaxies in a random order, and then select the first 100.

We perform the random shuffling by generating 10^4 random numers using our random number generator mentioned earlier, and use these numbers as keys in our `merge_sort` algorithm on which to "sort" an array of indices. We then apply these "sorted" indices on the sampled array to shuffle it, and select the first 100 instances. Finally, we sort these 100 galaxies using the same algorithm on their radii directly. We then use this sorted array to plot a CDF, normalized to 1, and present this in figure 3.

In correspondence with the PDF we saw in figure 2, we can see there are almost no satellites present at either edges of the interval. The most satellite galaxies are found at $\sim x = 0.1$, a result we also saw earlier.

```
def hist(x, binmin, binmax, nbins, log=False):
    """"""
    if log:
        bin_edges = np.logspace(np.log10(binmin), np.log10(binmax),
     nbins+1)
    else:
        bin_edges = np.linspace(binmin, binmax, nbins+1)

    histogram = np.zeros(nbins)
    for i in range(nbins):
        bin_mask = (x>bin_edges[i]) * (x<bin_edges[i+1])
        histogram[i] = len(x[bin_mask])#/(bin_edges[i+1]-bin_edges[
    i])

    return histogram, bin_edges
```

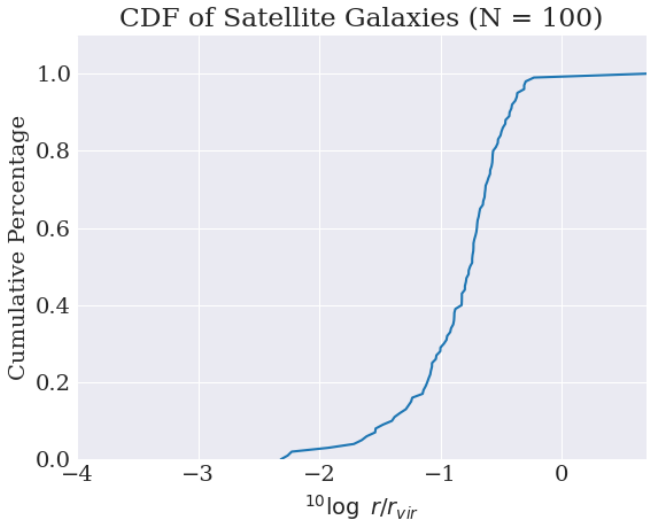

Figure 3: Cumulative distribution function normalized to 1 of $p(x)dx$ sampled using rejection sampling with $N = 10^4$, created using only a random subset of 100 satellites.

Listing 7: Code used to compute the histogram bins

```
    # 1c.
    # Step 1: Order a list of random numbers, select the first 100
    print('Shuffling Sample..')
    random_keys = rng(N=N_sample)
    random_idxs = merge_sort(key=random_keys)

    random_sample_idxs = random_idxs[:num_random_sample]
    random_sample = sampled_points[random_sample_idxs]

    # Step 2: Order these 100 galaxies by radius
    random_sample_ordered_idxs = merge_sort(key=random_sample)
    random_sample_ordered = random_sample[
    random_sample_ordered_idxs]

    # Make the 'Cumulative Distribtuion Function'
    y = np.arange(num_random_sample)/num_random_sample
    y = np.append(y, 1) # Add a final point to complete the CDF to
    (xmax, 1)
    random_sample_ordered = np.append(random_sample_ordered, x_max)

    plt.plot(np.log10(random_sample_ordered), y)
    plt.xlim(np.log10(sample_min_x), np.log10(x_max))
    plt.ylim(0, 1.1)
    plt.xlabel(r'$^{10}\log~r/r_{vir}$')
    plt.ylabel('Cumulative Percentage')
    plt.title(f'CDF of Satellite Galaxies (N = {num_random_sample})
    ')
    plt.savefig('results/satellite_galaxies_cdf')
```

Listing 8: Code to generate the CDF presented in this work

1.4 Derivative

To conclude this section we will look at the derivative of $n(x)$ (equation 1, not $N(x)$!) at $x = 1$. To compute this derivative we have used Ridder's Method with $h_{start} = 0.1$. Similarly to the Romberg integration earlier, we cannot choose $h_{start} = 1$ because then we would have to sample $n(x = 0)$, where the function is not defined. In agreement with the assignment, we set our target accuracy at 10^{-12}. This means we stop if our uncertainty, defined as the absolute difference between the current and previous best estimates, drops below this threshold. We compare our findings to the analytical derivative of $n(x)$ which we find to be

$$\frac{dn}{dx}_{(analytical)} = -A \langle N_{sat} \rangle \exp\left[-\left(\frac{x}{b}\right)^c\right] \frac{b^3 \left(\frac{x}{b}\right)^a \left(c\left(\frac{x}{b}\right)^c - a + 3\right)}{x^4} \tag{4}$$

Now comparing the two results, using Ridder's Method we find

$$\frac{dn}{dx}_{(ridder)} = -0.625328061833 \pm 2.220 \times 10^{-14}$$

While analytically we find

$$\frac{dn}{dx}_{(analytical)} = -0.625328061833$$

These two values match exactly up to 12 digits, indicating the strength of the numerical differentiator implemented here.

```
# 1d.
to_diff = lambda x: n(x, A, Nsat, a, b, c)
x = [1]
dndx, diff_unc = ridders_method(to_diff, x, 0.1, 2, 1e-12)
dndx_analytical = analytical_derivative(x[0], A, Nsat, a, b, c)

print(f'The derivative of n(x) calculated using Ridders Method at x = {x[0]} is {dndx[0]:.12f} +/- {diff_unc[0]:.3E}')
print(f'The analytical derivative of n(x) at x = {x[0]} is {dndx_analytical:.12f}')

results_txt += f'{dndx[0]:.12f}\n{diff_unc[0]:.3E}\n{dndx_analytical:.12f}'
```

Listing 9: Analytical derivative function, and code to use Ridder's Method

2 Heating and Cooling in HII regions

In this section we will investigate the equilibrium temperature of heating and cooling rates in an HII region similar to Orion. We will search for these equilibrium temperatures using a root finding algorithm. In this work we have chosen to use only the False Position Method to maintain the convergence guarantee provided by bisection, but increase the convergence speed somewhat. We note that faster convergence is possibly by combining a faster and less accurate in the beginning, with a slower and more accurate algorithm near the root. However this is not implemented here due to time constraints.

2.1 Simple Case

First we will only consider an HII region with one heating and cooling mechanism. The heating is produced by photoionization given by

$$\Gamma_{pe} = \alpha_B n_H n_e \psi k_B T_c, \tag{5}$$

and the cooling through radiative recombination:

$$\Lambda_{rr} = \alpha_B n_H n_e \left\langle E_{rr} \right\rangle, \tag{6}$$

with

$$\left\langle E_{rr} \right\rangle = \left[0.684 - 0.0416 \ln\left(\frac{T_4}{Z^2} \right) \right] k_B T. \tag{7}$$

In these equations, $\alpha_B = 2 \times 10^{-13}$ cm^3 s^{-1} is the case B recombination coefficient, $k_B = 1.38 \times 10^{-16}$ erg K^{-1} is the Stefan-Boltzmann constant, $\phi = 0.929$ is given, $T_c = 10^4$ K is the stellar temperature, $Z = 0.015$ is the metalicity and $T_4 \equiv \frac{T}{10^4 \mathrm{K}}$ is the temperature. Finally, we assume the proton and electron densities to be equal, such that $n_e = n_p$.

The equilibrium temperature T_eq is given by the temperature at which the heating rate is equal to the cooling rate, $\Gamma_{pe} = \Lambda_{rr}$. We can formulate this in terms of a root finding problem by searching for the point where $\Gamma_{pe} - \Lambda_{rr} = 0$. We can work out what this should be by combining equations 5 and 6 to find that this means the following should hold

$$\psi T_c - \left[0.684 - 0.0416 \ln\left(\frac{T}{10^4 \cdot Z^2} \right) \right] \cdot T = 0. \tag{8}$$

Note that the densities n, α_B and k have all dropped out of this specific equation, even though this does not represent the true difference between heating and cooling rate at other temperatures. This is not important because we are only interested in the temperature at which the above equation is true, where additional scaling factors are important.

We apply our root finding algorithm introduced earlier to this equation with an initial bracket in the range $T = \left[1, 10^7\right]$ K. We aim for a temperature accuracy of 0.1 K. We present the result in figure 4, we can see that we find a root at $T = 3.25 \times 10^4$ K, where the function has a value of -3.45×10^{-3}. This is a relatively large value, but the algorithm has converged because it is taking very small steps along the temperature axis.

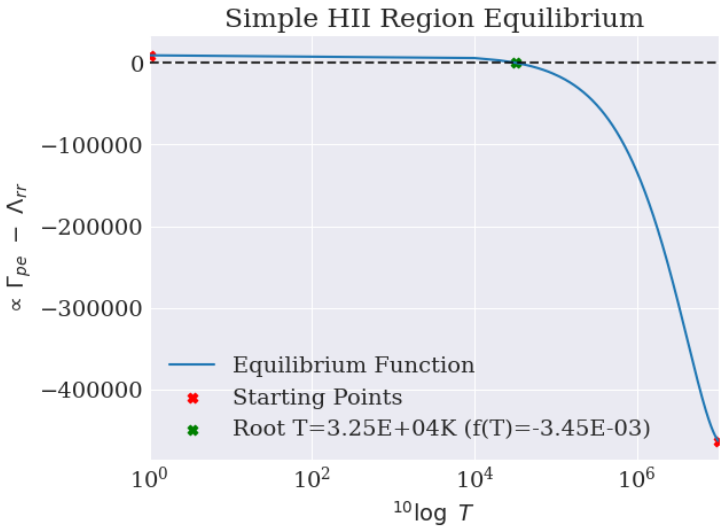

Figure 4:

2.2 More Complex Case

In addition to the singular heating and cooling mechanisms we described in the previous section, we will now setup a more realistic scenario and also include cooling through free-free emission (Λ_{ff}) and heating through cosmic rays (Σ_{cr}) and magnetohydrodynamic waves (Σ_{MHD}). These are all individually,given by the following three equations:

$$\Lambda_{ff} = 0.54 T_4^{0.37} \alpha_B n_e n_h k_B T \tag{9}$$

$$\Gamma_{cr} = A n_e \xi_{CR} \tag{10}$$

$$\Gamma_{MHD} = 8.9 \times 10^{-26} n_h T_4 \tag{11}$$

Synonymous to the previous section we can find the equilibrium temperature T_{eq} by finding the temperature for which $\Gamma - \Lambda = 0$ where Γ and Λ represent the total heating and cooling rate respectively, which are just the sums of each of the individual effects. Working this out, we find that the following equation most hold:

$$\left(\psi T_c - \left[0.684 - 0.0416\left(\frac{T}{10^4 \cdot Z^2}\right)\right] \cdot T - 0.54\left(\frac{T}{10^4}\right) T\right) \cdot k_B n_H \alpha_B$$
$$... + A\xi + 8.9 \times 10^{-26} n_H \frac{T}{10^4} = 0 \tag{12}$$