NUR Hand-in Exercise 3

Angèl Pranger

April 10, 2025

Abstract

This document shows my solutions to hand-in exercise 3 of numerical methods in astrophysics.

1 Satellite galaxies around a massive central

General code used in this exercise is:

```python
import numpy as np
import matplotlib.pyplot as plt
from scipy.special import gammainc
from tqdm import tqdm

def n(x,A,Nsat,a,b,c):
    """Returns the function value of n."""
    return A*Nsat*((x/b)**(a-3))*np.exp(-(x/b)**c)

class Func:
    def __init__(self, A, Nsat, a, b, c):
        """Class to compute the maximum of N."""
        self.A = A
        self.Nsat = Nsat
        self.a = a
        self.b = b
        self.c = c
        pass

    def min_N(self, x):
        """Returns the function value -N(x)dx which must be minimized."""
        return -4*np.pi*x**2*n(x, self.A, self.Nsat, self.a, self.b, self.c)

class Integrand:
    def __init__(self,a,b,c):
        """Class to compute A and Ntilda."""
        self.a = a
        self.b = b
        self.c = c
        pass

    def integrand(self, x):
        """Returns the integrand that is used to compute A and Ntilda."""
        return x**2*((x/self.b)**(self.a-3))*np.exp(-(x/self.b)**self.c)

def readfile(filename):
    """Takes a filename and returns the radius of all the galaxies in the filename and
    the number of halos."""
    f = open(filename, 'r')
    data = f.readlines()[3:] #Skip first 3 lines
    nhalo = int(data[0]) #number of halos
    radius = []

    for line in data[1:]:
        if line[:-1]!='#':
            radius.append(float(line.split()[0]))

```

```
    radius = np.array(radius, dtype=float)
    f.close()
    return radius, nhalo #Return the virial radius for all the satellites in the file, and the number of halos

# Copied from the previous assignment
def extended_midpoint_Romberg(f, a, b, m=5):
    """Computes the integral of f between a and b (with a<b) using the open method. Returns the best estimate value of the integral and an estimate for the error."""
    one_third = 1/3
    # Set initial step size
    h = b-a
    # Initialize array of estimates r of size m to zero
    r = np.zeros(m)
    # Calculate initial estimate using midpoint rule for N=1
    r[0] = h*f(0.5*(a+b))
    # Computing estimates for N=3, N=9, etc
    Np = 2
    for i in range(1, m):
        h *= one_third
        x = a + h*0.5
        # Add midpoint rule for new points
        j = 1 # Tracker of step size between new points (either h or 2h)
        for _ in range(Np):
            r[i] += h*f(x)
            x += (j+1)*h # j+1 alternates between 1 and 2
            j = (j+1)%2 # j alternates between 0 and 1
        # Add values for already calculated points
        r[i] += one_third*r[i-1]
        Np *= 3
    # Do weighted combinations
    Np = 1
    for i in range(1, m):
        Np *= 9
        factor = 1/(Np-1)
        for j in range(m-i):
            r[j] = (Np*r[j+1]-r[j])*factor
    return r[0], np.absolute(r[0]-r[1])

# Copied from the previous assignment, changed that the indexing array is now sorted instead of the original array
def quicksort_recursive(array, indx, low, high):
    """Recursively sort indx from low to high using array as key using quicksort algorithm. Alters the indexing array."""
    # Set pivot to middle element
    pivot = np.int64(np.ceil((low+high)/2))
    x_pivot = array[indx][pivot]
    # Looping to sort elements with respect to the pivot
    i = low
    j = high
    while True:
        while (array[indx][i] < x_pivot):
            i += 1
        while (array[indx][j] > x_pivot):
            j -= 1
        if (j<=i):
            break
        else:
            mem = indx[i]
            indx[i] = indx[j]
            indx[j] = mem
            # If pivot is swapped, change location of pivot to new location
            # Let complementary indexer continue to prevent infinite looping in case array[indx][i]=array[indx][j]=array[indx][pivot]
            if (i == pivot):
                pivot = j
                i += 1
            elif (j == pivot):
                pivot = i
                j -= 1
```

```
    # Apply algorithm recursively to subarrays left and right of the pivot
    if (low < pivot-1):
        indx = quicksort_recursive(array, indx, low, pivot-1)
    if (high > pivot+1):
        indx = quicksort_recursive(array, indx, pivot+1, high)
    return indx

def quicksort(array):
    """Sort array using quicksort algorithm. Returns the sorted indexing array."""
    indx = np.arange(array.shape[0])
    indx = quicksort_recursive(array, indx, 0, array.shape[0]-1)
    return indx
```

Q1.py

1.1 a

We use golden section search to find the maximum $N(x)$ for $x \in [0, 5)$. We implemented the golden section search algorithm as a minimization algorithm, so instead of maximizing $N(x)$ we minimize $-N(x)$. The algorithm requires us to choose an initial bracket. We plot the function N with the given parameters and domain, the result of which is shown in Figure 1, from which we see that (0, 0.5, 1) forms a good bracket (because $N(0) < N(0.5)$ and $N(1) < N(0.5)$). Using this bracket and applying the minimization algorithm, we find the maximum which is also indicated in Figure 1.

The code used is:

```
def golden_section_minimization(f, a, b, c, target_accuracy=1e-10, num_steps=1000):
    """Assumes that a, b, c form a bracket with b in between a and c.
    Looks for a minimum of f between a and c. Note that only one minimum is found,
    which might be a local minimum instead of a global minimum if there are multiple.
    Returns the found minimum of f between a and c."""
    # Identify larger interval
    side_tracker = False # false if larger interval is (b,c), true if (a,b)
    if (np.absolute(c-b) < np.absolute(b-a)):
        side_tracker = True
    # Walk through the steps repeatedly
    for _ in range(num_steps):
        # Choose point d inside the larger interval in self-similar way
        if (side_tracker):
            d = b+(a-b)*0.38197
        else:
            d = b+(c-b)*0.38197
        # Return if target accuracy is reached
        if (np.absolute(c-a) < target_accuracy):
            if (f(d) < f(b)):
                return d
            else:
                return b
        # Tightening bracket
        if (f(d) < f(b)): # Larger interval side stays the same
            if (side_tracker):
                c = b
                b = d
            else:
                a = b
                b = d
        else: # Larger interval side switches
            if (side_tracker):
                a = d
                side_tracker = False
            else:
                c = d
                side_tracker = True
    print("The number of steps was too small to reach the target accuracy.")
    return d

# Initialize class for the function used in minimization
Func1 = Func(256/(5*np.pi**(1.5)), 100, 2.4, 0.25, 1.6)
```

```python

# Finding the values at the maximum
max_x = golden_section_minimization(Func1.min_N, 0, 0.5, 1)
max_N = -Func1.min_N(max_x)
print(f"We find that x={max_x} and N(x)={max_N} at the maximum.")

# Plotting the function with the maximum indicated
fig1a, ax = plt.subplots(1,1,figsize=(5.0,5.0))
x = np.linspace(0, 5, 1000)
ax.plot(x, -Func1.min_N(x), label='N(x)')
ax.vlines(max_x, 0, max_N, label='maximum', color='orange', linestyle='--')
plt.tight_layout()
ax.legend(loc='best')
ax.set_xlabel('x')
ax.set_ylabel('N(x)')
plt.savefig('my_solution_1a.png', dpi=600)
```

Q1.py

The resulting maximum is:

```
We find that x=0.22998263292929105 and N(x)=267.8455331337432 at the maximum.
```

output_Q1.txt

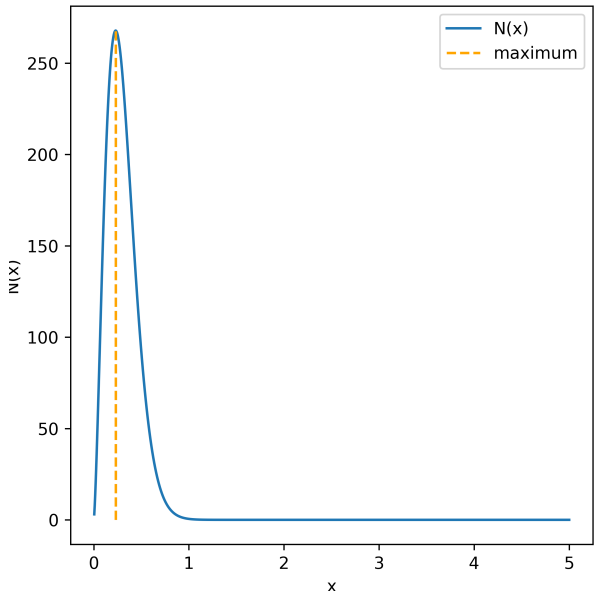

Figure 1: The function $N(x)$ is plotted for $x \in [0, 5)$. The found maximum is indicated with a vertical line. Indeed, this corresponds to the maximum of the function.

1.2 b

We choose to use 50 radial bins in log space, as more bins gives more accurate information on the function so a tighter constraint on the model fit, which means that the minimum will be more easily found. As range we choose to use $xmin = 1e^{-4}$ as in the previous assignment, such that we do not have to deal with division by zero but at the same time do keep x small. We considered using the minimal radius within the dataset as $xmin$, but this would mean we do not take the fact that we find no satellite galaxies below this radius into account in the model, while this is important information that should not be left out. For the maximal radius we choose to use the maximal radius found in the dataset.

We calculate $\langle N_{sat} \rangle$ by dividing the total number of radii by the number of halos within the dataset. The binned data N_i is also divided by the number of halos.

We have that

$$\begin{aligned}\tilde{N}_i &= 4\pi \int_{x_i}^{x_{i+1}} n(x)x^2\mathrm{d}x \\ &= 4\pi \int_{x_i}^{x_{i+1}} A\langle N_{sat}\rangle (\frac{x}{b})^{a-3} \exp -(\frac{x}{b})^c x^2\mathrm{d}x \\ &= \frac{4\pi \int_{x_i}^{x_{i+1}} \langle N_{sat}\rangle (\frac{x}{b})^{a-3} \exp -(\frac{x}{b})^c x^2\mathrm{d}x}{4\pi \int_0^5 (\frac{x}{b})^{a-3} \exp -(\frac{x}{b})^c x^2\mathrm{d}x} \\ &= \frac{\langle N_{sat}\rangle \int_{x_i}^{x_{i+1}} (\frac{x}{b})^{a-3} \exp -(\frac{x}{b})^c x^2\mathrm{d}x}{\int_0^5 (\frac{x}{b})^{a-3} \exp -(\frac{x}{b})^c x^2\mathrm{d}x}.\end{aligned}$$

The integrals are computed using the midpoint extended Romberg method which we implemented in the previous assignment. We use this form to compute

$$\chi^2 = \sum \frac{(N_i - \tilde{N}_i)^2}{\tilde{N}_i},$$

where the sum is over the bins.

To minimize this χ^2 we use the downhill simplex method with a maximum of 500 steps, as we found that in most cases it will converge within a few hundred steps and otherwise take ages (get stuck?) to converge. The sorting within the downhill simplex method is done using quicksort which we implemented in the previous assignment. We have found that the found minimum depends on the choice of initial simplex. Therefore, we use a few different initial simplexes, around the parameters $a = 2.4, b = 0.25, c = 1.6$, and repeat the method for each of these, after which we take the parameters that provide the lowest χ^2 over all. The resulting fits are shown in Figure 2. By visual inspection the fits look good.

The code used is:

```
def downhill_simplex(f, points, target_accuracy=1e-8, limit_num=500):
    """Uses the downhill simplex method to compute the minimum of a multidimensional
    function f.
    The initial simplex is given by points, which contains the N+1 points forming the
    simplex.
    Returns the point x which is the found minimum."""
    N = points.shape[0]-1
    f_points = np.zeros(N+1)
    factor_N = 1/N
    for i in range(N+1):
        f_points[i] = f(points[i])
    best_f = np.zeros(limit_num)
    for j in tqdm(range(limit_num)):
        # Order the points such that f(x0) leq f(x1) leq ... leq f(xN)
        indx = quicksort(f_points)
        f_points = f_points[indx]
        points = points[indx,:]
        best_f[j] = f_points[0]
        # Check accuracy
        if (2*np.absolute(f_points[-1]-f_points[0]) < target_accuracy*np.absolute(
    f_points[-1]+f_points[0])):
            return points[0], best_f[0:j+1]
        # Calculate the centroid of the first N points (excluding the worst one)
        centroid = np.sum(points[:-1], axis=0)*factor_N
        # Propose new point by reflecting worst point xN in centroid
        x_try = 2*centroid-points[-1]
        f_try = f(x_try)
        if (f_points[0] <= f_try):
            if (f_try < f_points[-1]):
                # Replace worst point
                points[-1] = x_try
                continue
            else:
                # Propose new point by contracting instead of reflecting
                x_try = 0.5*(centroid+points[-1])
```

```
33                  f_try = f(x_try)
34                  if (f_try < f_points[-1]):
35                      # Replace worst point
36                      points[-1] = x_try
37                      f_points[-1] = f_try
38                      continue
39                  else:
40                      # All other options are bad so zoom in on the best point by
    contracting all other points
41                      for i in range(1, N+1):
42                          points[i] = 0.5*(points[0]+points[i])
43                          f_points[i] = f(points[i])
44                      continue
45          elif (f_try < f_points[0]):
46              # Propose second point by expanding further in same direction
47              x_exp = 2*x_try-centroid
48              f_exp = f(x_exp)
49              if (f_exp < f_try):
50                  # Replace worst point by expanded one
51                  points[-1] = x_exp
52                  f_points[-1] = f_exp
53                  continue
54              else:
55                  # Replace worst point by initial reflected one
56                  points[-1] = x_try
57                  f_points[-1] = f_try
58                  continue
59      return points[0], best_f
60
61  class Likelihood_chi2:
62      def __init__(self, bin_edges, binned_data, Nsat):
63          """Class to compute the minimum of chi2 for the problem."""
64          self.bin_edges = bin_edges
65          self.binned_data = binned_data
66          self.Nsat = Nsat
67          self.mean_var = np.zeros(bin_edges.shape[0]-1)
68          pass
69
70      def chi2(self, params):
71          """Returns the value of chi2 for the given parameters.
72          Meanwhile updates self.mean_var to contain the corresponding model bin values.
    """
73          Integrand1 = Integrand(params[0],params[1],params[2])
74          normalization_factor = self.Nsat / extended_midpoint_Romberg(Integrand1.
    integrand, 0, xmax)[0]
75          chi2 = 0
76          # Loop over all bins, adding the corresponding contribution to chi2
77          for i in range(len(self.bin_edges)-1):
78              self.mean_var[i] = normalization_factor * extended_midpoint_Romberg(
    Integrand1.integrand, self.bin_edges[i], self.bin_edges[i+1])[0]
79              chi2 += (self.binned_data[i]-self.mean_var[i])**2 / self.mean_var[i]
80          return chi2
81
82      def get_model(self, params):
83          """Returns the chi2 for the given parameters and the corresponding model bin
    means."""
84          chi2 = self.chi2(params)
85          return chi2, self.mean_var
86
87  # Set number of bins
88  n_bins = 50
89  # Set initial simplex vertices
90  initial_params = np.array
        ([[[2.4,0.25,1.6],[2.5,0.25,1.6],[2.4,0.35,1.6],[2.4,0.25,1.7]],
91                              [[2.4,0.25,1.6],[2.9,0.25,1.6],[2.4,0.75,1.6],[2.4,0.25,2.1]],
92                              [[2.4,0.25,1.6],[3.4,0.25,1.6],[2.4,1.25,1.6],[2.4,0.25,2.6]],
93                              [[2.4,0.25,1.6],[1.4,0.25,1.6],[2.4,1.25,1.6],[2.4,0.25,0.6]],
94
        [[3.4,1.25,2.6],[2.9,1.25,2.6],[3.4,0.75,2.6],[3.4,1.25,2.1]]])
95  # Set xmin and xmax for all datasets
```

```
xmin = 1e-4

print("dataset, Nsat, best-fit a b c, minimal chi2, G, Q")

# Minimizing chi2 and plotting the results for all datasets
fig1b, ax = plt.subplots(3,2,figsize=(6.4,8.0))
for i in range(5):
    # Read data
    radii, nhalo = readfile(f'satgals_m1{i+1}.txt')
    xmax = np.max(radii)
    # Set the bin edges
    edges = np.exp(np.linspace(np.log(xmin), np.log(xmax), n_bins+1))
    factor_halo = 1/nhalo
    # Compute mean number of satellites in each halo
    Nsat = radii.shape[0]*factor_halo
    # Mean number of satellites per halo (so divide by number of halos) in each radial bin
    binned_data=np.histogram(radii,bins=edges)[0]*factor_halo

    # Minimize the chi squared for different starting simplex
    Minimal_chi2 = np.inf
    for j in range(len(initial_params)):
        Chi2 = Likelihood_chi2(edges, binned_data, Nsat)
        best_params, best_f = downhill_simplex(Chi2.chi2, initial_params[j])
        # fig, axes = plt.subplots(1,1)
        # axes.plot(best_f)
        # fig.savefig(f'convergence_{i}{j}.png')
        minimal_chi2, Ntilda_option = Chi2.get_model(best_params)
        if (minimal_chi2 < Minimal_chi2):
            Minimal_chi2 = minimal_chi2
            Best_params = best_params
            Ntilda = Ntilda_option

    # Perform G-test, multiplying by nhalo to ensure the observation counts are integers
    G, Q = G_test(binned_data*nhalo, Ntilda*nhalo, n_bins-3)
    print(f"satgals_m1{i+1}", f"{Nsat:.4}", Best_params, f"{Minimal_chi2:.4}", f"{G:.4}", f"{Q:.4}")

    row=i//2
    col=i%2
    ax[row,col].step(edges[:-1], binned_data, where='post', label='binned data')
    ax[row,col].step(edges[:-1], Ntilda, where='post', label='best-fit profile')
    ax[row,col].set(yscale='log', xscale='log', xlabel='x', ylabel='N', title=f"$M_h \\approx 10^{{{11+i}}} M_{{\\odot}}/h$")
ax[2,1].set_visible(False)
fig1b.tight_layout()
handles,labels=ax[2,0].get_legend_handles_labels()
fig1b.legend(handles, labels, loc=(0.65,0.15))
fig1b.savefig('my_solution_1b.png', dpi=600)
```

Q1.py

The corresponding fit values for the Gaussian fit are:

```
dataset, Nsat, best-fit a b c, minimal chi2, G, Q
satgals_m11 0.01368 [1.31597531 1.11463223 3.13341498] 1.809e-06 273.3 0.0
satgals_m12 0.2509 [1.59335936 0.91850015 3.46907134] 3.474e-05 172.3 3.331e-16
satgals_m13 4.374 [1.48561461 0.80223428 2.87434531] 0.00237 89.82 0.0001696
satgals_m14 29.13 [1.9285268  0.60648511 2.47866022] 0.1567 34.26 0.917
satgals_m15 329.5 [1.85587579 0.75587043 2.14070353] 7.793 19.83 0.9998
```

output_Q1.txt

1.3 c

The Poisson log-likelihood for $\tilde{N}_i$ model counts and N_i mean observed counts per bin is

$$-\ln(\mathcal{L}(a,b,c)) = -\sum(N_i \ln(\tilde{N}_i) - \tilde{N}_i)$$

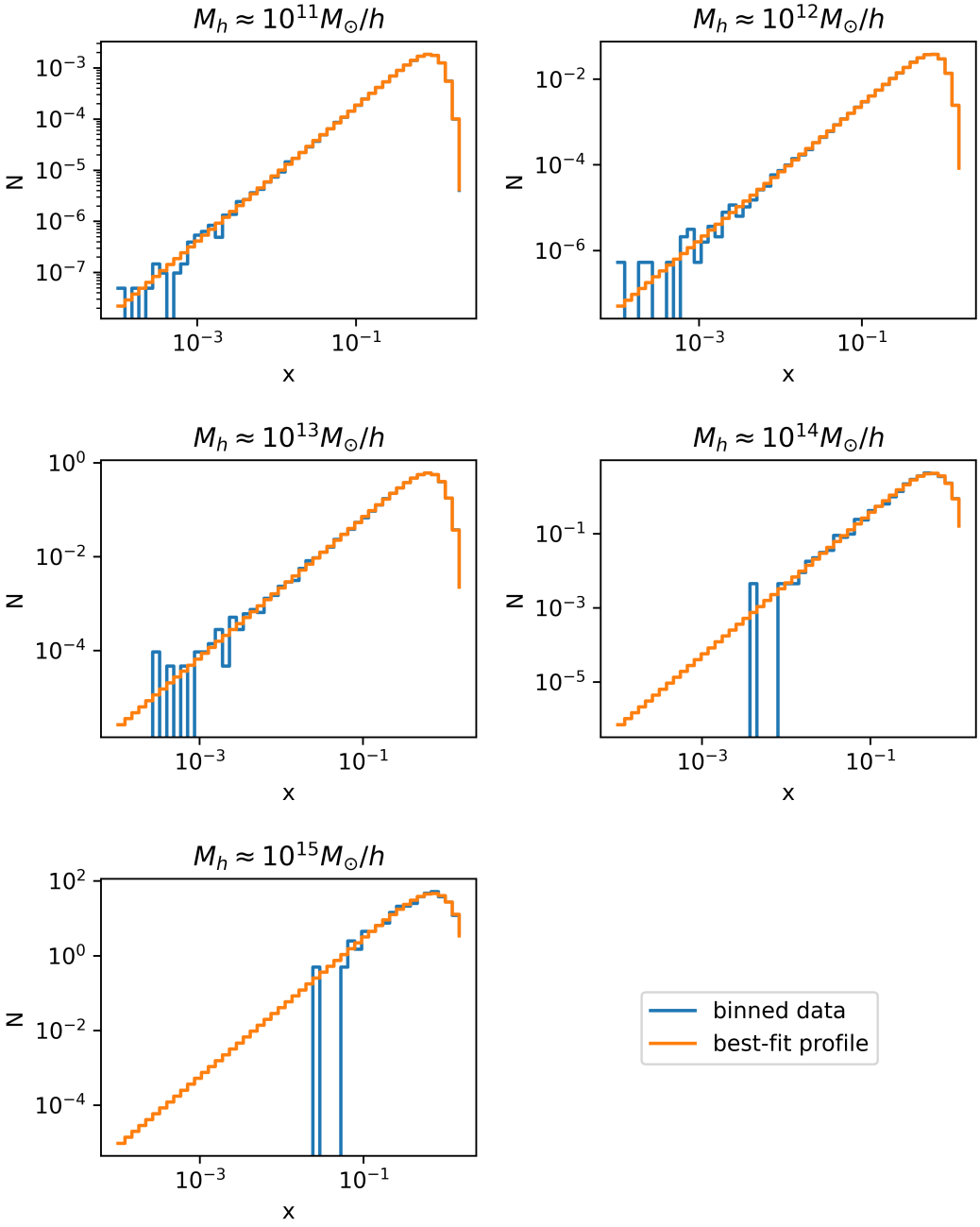

Figure 2: In each of the panels a different binned dataset (blue) and corresponding model fit (orange) is shown. The used model is Gaussian, resulting in a χ^2 fit. Note that the axes are in log space, such that the obvious and seemingly weird gaps at lower radii are caused by for example missing only one or two count in that particular bin. The fits look good by visual inspection.

where the sum is again over the same bins as for b. We now use the same method as for b to minimize the -log likelihood. The resulting fits are shown in Figure 3.

The code used is:

```
class Likelihood_poisson:
    def __init__(self, bin_edges, binned_data, data, Nsat):
        """Class to compute the minimum of -ln(likelihood) corresponding to the poisson
    distribution for the problem."""
```

```
        self.bin_edges = bin_edges
        self.binned_data = binned_data
        self.data = data
        self.Nsat = Nsat
        self.mean_var = np.zeros(bin_edges.shape[0]-1)
        pass

    def poisson_binned_log_likelihood(self, params):
        """Returns the value of -ln(likelihood) for the given parameters.
        Meanwhile updates self.mean_var to contain the corresponding model bin values.
    """
        Integrand1 = Integrand(params[0],params[1],params[2])
        normalization_factor = self.Nsat / extended_midpoint_Romberg(Integrand1.
    integrand, 0, xmax)[0]
        likelihood = 0
        # Loop over all bins, adding the corresponding contribution to -ln(likelihood)
        for i in range(len(self.bin_edges)-1):
            self.mean_var[i] = normalization_factor * extended_midpoint_Romberg(
    Integrand1.integrand, self.bin_edges[i], self.bin_edges[i+1])[0]
            likelihood -= (self.binned_data[i]*np.log(self.mean_var[i])-self.mean_var[i
    ])
        return likelihood

    def get_model(self, params):
        """Returns the -ln(likelihood) for the given parameters and the corresponding
    model bin means."""
        likelihood = self.poisson_binned_log_likelihood(params)
        return likelihood, self.mean_var

print("dataset, Nsat, best-fit a b c, minimal -ln L(a,b,c), G, Q")

# Minimizing the poisson likelihood and plotting the results for all datasets
fig1c, ax = plt.subplots(3,2,figsize=(6.4,8.0))
for i in range(5):
    # Read data
    radii, nhalo = readfile(f'satgals_m1{i+1}.txt')
    xmax = np.max(radii)
    # Set the bin edges
    edges = np.exp(np.linspace(np.log(xmin), np.log(xmax), n_bins+1))
    factor_halo = 1/nhalo
    # Compute mean number of satellites in each halo
    Nsat = radii.shape[0]*factor_halo
    # Mean number of satellites per halo (so divide by number of halos) in each radial
    bin
    binned_data=np.histogram(radii,bins=edges)[0]*factor_halo

    # Minimize the negative poisson log likelihood for different starting simplex
    Minimal_likelihood = np.inf
    for j in range(len(initial_params)):
        Poisson = Likelihood_poisson(edges, binned_data, radii, Nsat)
        best_params, best_f = downhill_simplex(Poisson.poisson_binned_log_likelihood,
    initial_params[j])
        # fig, axes = plt.subplots(1,1)
        # axes.plot(best_f)
        # fig.savefig(f'convergence2_{i}{j}.png')
        minimal_likelihood, Ntilda_option = Poisson.get_model(best_params)
        if (minimal_likelihood < Minimal_likelihood):
            Minimal_likelihood = minimal_likelihood
            Best_params = best_params
            Ntilda = Ntilda_option

    # Perform G-test, multiplying by nhalo to ensure the observation counts are integers
    G, Q = G_test(np.int64(binned_data*nhalo), Ntilda*nhalo, n_bins-3)
    print(f"satgals_m1{i+1}", f"{Nsat:.4}", Best_params, f"{Minimal_likelihood:.4}", f"{
    G:.4}", f"{Q:.4}")

    row=i//2
    col=i%2
    ax[row,col].step(edges[:-1], binned_data, where='post', label='binned data')
    ax[row,col].step(edges[:-1], Ntilda, where='post', label='best-fit profile')
```

```
    ax[row,col].set(yscale='log', xscale='log', xlabel='x', ylabel='N', title=f"$M_h \\
    approx 10^{{{11+i}}} M_{{\\odot}}/h$")
ax[2,1].set_visible(False)
fig1c.tight_layout()
handles,labels=ax[2,0].get_legend_handles_labels()
fig1c.legend(handles, labels, loc=(0.65,0.15))
fig1c.savefig('my_solution_1c.png', dpi=600)
```

Q1.py

The corresponding fit values for the Poisson fit are:

```
dataset, Nsat, best-fit a b c, minimal -ln L(a,b,c), G, Q
satgals_m11 0.01368 [1.31703755 1.1139355  3.12978727] 0.1086 266.6 0.0
satgals_m12 0.2509 [1.59538908 0.91779227 3.46371133] 1.226 117.9 5.155e-08
satgals_m13 4.374 [1.49148062 0.79985991 2.86094259] 9.323 82.38 0.001084
satgals_m14 29.13 [1.96513299 0.59657734 2.43247333] 3.514 31.6 0.9585
satgals_m15 329.5 [2.00527625 0.69291502 1.97492878] -751.9 18.92 0.9999
```

output_Q1.txt

1.4 d

We calculate the G-statistic for both the Gaussian and Poissonian method. The G-statistic is calculated by

$$G = 2\sum O_i \ln(\frac{O_i}{E_i}),$$

where O_i is the observed number of satellites in each bin (as an integer, so we multiply by the number of halos), and E_i is the expected number of satellites in each bin according to the model. E_i must be a positive number, which it is in each bin. Furthermore, if $O_i = 0$ in certain bins, we do not take the contribution of these bins into account as $O_i \ln(O_i) \to 0$. To compute the significance Q of G we have that

$$\begin{aligned} Q &= 1 - P(G,k) \\ &= 1 - \frac{\gamma(\frac{k}{2},\frac{G}{2})}{\Gamma(\frac{k}{2})}, \end{aligned}$$

which is the regularized lower incomplete gamma function. Here k is the number of degrees of freedom, which is equal to the number of bins minus the number of parameters (3) in this case.

The results for each of the datasets are shown in the tables of results for b and c. We find that the values for G for the Gaussian and Poissonian models are comparable for each dataset, with the value for the Poissonian model always being a bit smaller than the value for the Gaussian model, indicating that the Poissonian model is a bit more consistent with the data (or we are a bit more unlucky with the Gaussian model, but as it is the case for each of the five datasets, it is more likely that the Gaussian model is actually less consistent with the data).

I am confused by the values I am getting for Q, as the first three datasets get very low values (smaller than 0.01) for Q for each of the two models, even though the models seem to fit well to the data (be consistent) by eye. Therefore, I think I might have done something wrong in computing Q. However, the values for Q for a certain dataset are each time larger for the Poissonian model than for the Gaussian model, again indicating the the Poissonian model is more consistent with the data when comparing the two.

The code specific for this exercise is the following, but the use of this function is done within the codes for b and c.

```
def G_test(observations, expectations, k_dof):
    """Computes the G statistic and corresponding Q. The observations must be integers
    >=0.
    The expectations must be >0 for each bin. Returns G and Q."""
    # Terms with observations=0 are masked, because these do not contribute to the sum
```

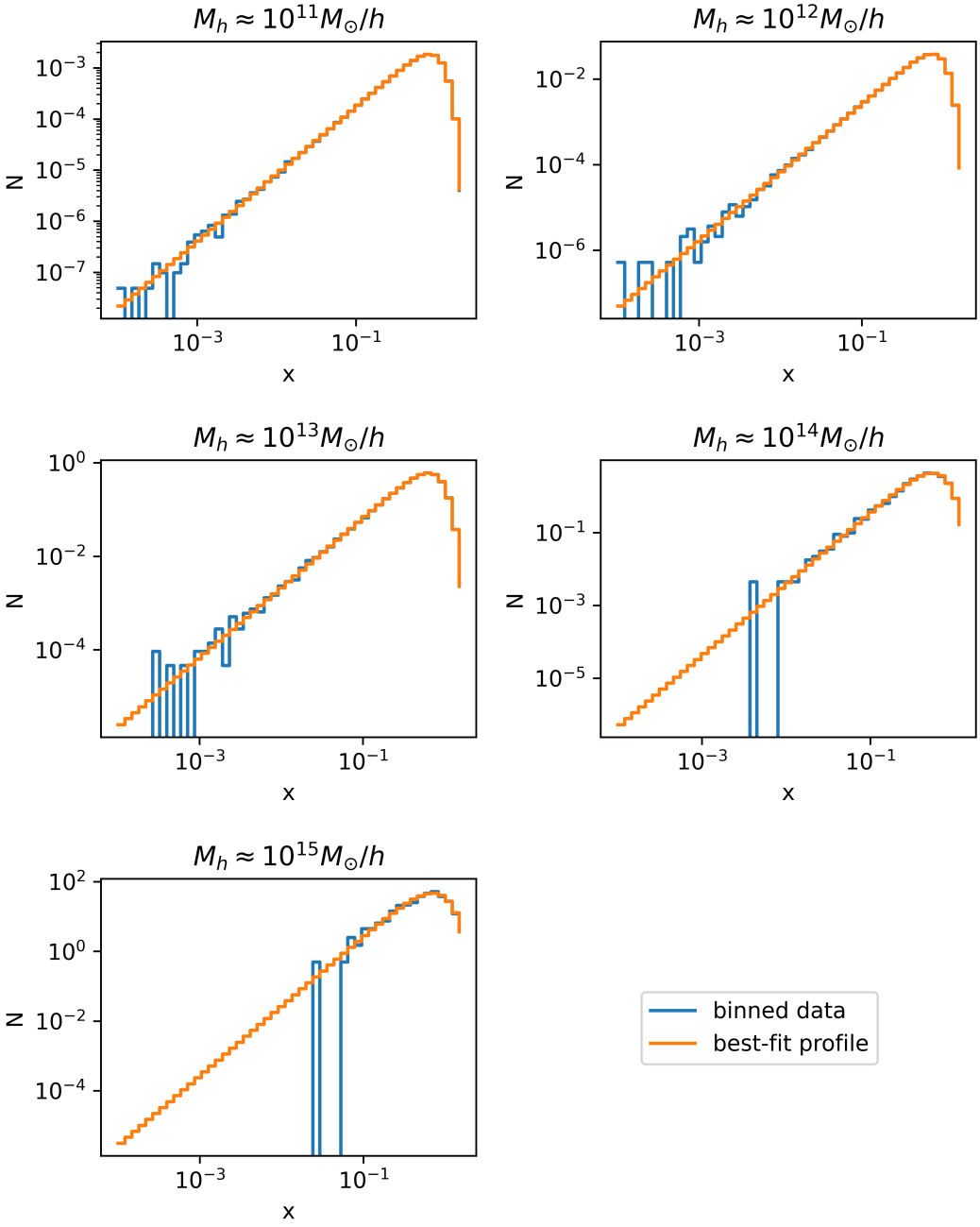

Figure 3: In each of the panels a different binned dataset (blue) and corresponding model fit (orange) is shown. The used model is Poissonian. Note that the axes are in log space, such that the obvious and seemingly weird gaps at lower radii are caused by for example missing only one or two count in that particular bin. The fits look good by visual inspection.

```
    mask = (observations > 0)
    G = 2*np.sum(observations[mask]*(np.log(observations[mask]/expectations[mask])))
    Q = 1-gammainc(k_dof*0.5, G*0.5) # using regularized lower incomplete gamma function
    return G, Q
```

Q1.py