Numerical Recipes: Hand-In 3

Rens Kievit (s1948415)

April 20, 2023

Abstract

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

Plotting styles in this report are set using the following code

```python

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
```

Listing 1: Matplotlib Plotting Styles

1 Satellite Galaxies Around a Massive Central

In this section we will investigate the spherical distribution of satellite galaxies around a massive central galaxies, and attempt to fit a function to simulated data. Their density distribution n can be described as

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

Here x is the radius relative to the virial radius, i.e. $x \equiv r/r_{vir}$ with $x < x_{max} = 5$. a, b and c are free parameters, $\langle N_{sat} \rangle$ is the mean number of satellites per halo and $A = A(a, b, c)$ normalizes this profile such that $\int\int\int_V n(x)dV = \langle N_{sat} \rangle$. In this work we will mainly look at tt the number of satellites in the infinitesimal range $[x, x + dx\rangle$. This is given by

$$N(x)dx = n(x)4\pi x^2 dx \tag{2}$$

```
# Main Equations
def n(x, A, Nsat, a, b, c):
    """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 N(x, A, Nsat, a, b, c):
    """Number of satellites at a distance x. This is the
    function n(x, ..) integrated over the full sphere at x"""
    return 4.*np.pi*x*x*n(x, A, Nsat, a, b, c)
```

Listing 2: Satellite galaxy distribution code

1.1 Maximization

We start by searching for the maximum of the distribution given by equation 2, for this we will assume $a = 2.4$, $b = 0.25$, $c = 1.6$. $x_{max} = 5$, $\langle N_{sat} \rangle = 100$ and $A = 256/(5\pi^{3/2})$. Instead of searching for the maximum, we instead search for the minimum of $-N(x)dx$ which gives the equivalent resulting x. Visually inspecting the distribution (Figure 1), we see a clear peak at $x \sim 0.5$. To be safe we set the edges of our initial bracket at $x_{min} = 0$ and $x_{max} = 5$ and then apply a bracketing algorithm to find a three-point bracket around the minimum. Then we use this bracket as input to the golden section search algorithm to find the x-value at the peak. We find the following brackets and minimization results:

```
Minimization Results
Bracket: [5.00000000e+00 1.00000000e-04 8.09010814e+00]
x at Max: 0.22998248152335327
N(x) Max: 267.84553313361295
```

Listing 3: Results of the maximization algorithm.

We also show the distribution and the exact location of this peak in Figure 1. We can see that the algorithms have perfectly discovered the maximum of this distribution.

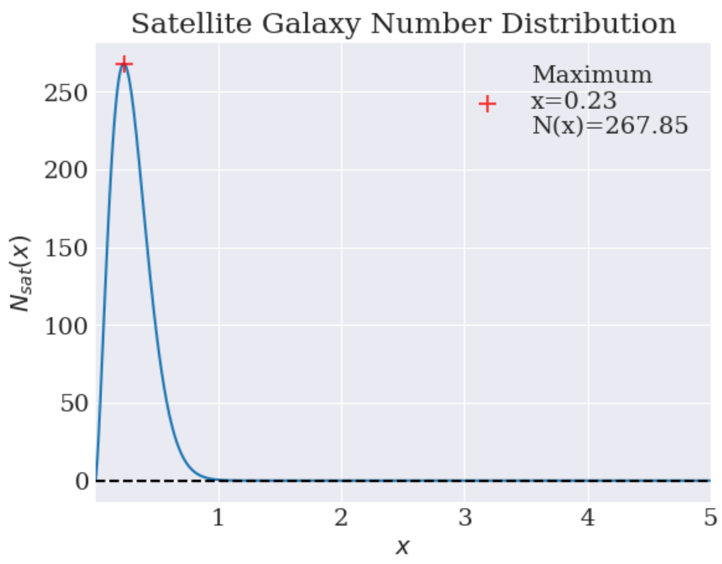


Figure 1: Distribution of the number of galaxies in the infinitesimal range $[x, x + dx\rangle$ described by equation 2. The red cross indicates the position of the maximum of this distribution discovered using the golden search algorihtm as described in Section 1.1

```python
def maximization():
    """Code for Q1a"""
    # Constants
    a = 2.4
    b = 0.25
    c = 1.6
    xmin = 1e-4 # this cannot be zero because of the factor (x/b)^(a-3) and a-3 < 0
    xmax = 5
    Nsat = 100
    A = 256./(5.*np.pi**(3./2.))

    # Maximizing a function f is equal to minimizing -f
    minim_func = lambda x: -1*N(x, A, Nsat, a, b, c)

    # Make a three-point bracket surrounding the minimum. As initial
    # edges we take the edges of the interval [0, 5]
    bracket, _ = make_bracket(minim_func, [xmin, xmax])
    x_at_max, _ = golden_section_search(minim_func, bracket)
    max_val = N(x_at_max, A, Nsat, a, b, c)

    print(f'Maximum of N(x) found at x = {x_at_max}, N(x) = {max_val}')

    xx = np.linspace(xmin, xmax, 1000)
    yy = N(xx, A, Nsat, a, b ,c)
    plt.plot(xx, yy)
    plt.scatter(x_at_max, max_val, c='red', marker='+', alpha=0.75, s=100,zorder=3, label=f'Maximum\nx={x_at_max:.2f}\nN(x)={max_val:.2f}')
    plt.axhline(y=0, c='black', ls='--')
    plt.xlim(xmin,xmax)
    plt.xlabel(r'$x$')
    plt.ylabel(r'$N_{sat}(x)$')
    plt.title('Satellite Galaxy Number Distribution')
    plt.legend()
    plt.savefig('results/maxi.png', bbox_inches='tight')

    with open ('results/maxi_results.txt', 'w') as file:
        file.write(f"""Minimization Results
Bracket: {bracket}
x at Max: {x_at_max}
N(x) Max: {max_val}""")
```

Listing 4: Code calling the maximization algorithm

```python
# MINIMIZATION
def parabola_min_analytic(a, b, c, fa, fb, fc):
    """Analytically computes the x-value of the minimum of a parabola
    that crosses a, b and c
    """
    top = (b-a)**2 * (fb-fc)  - (b-c)**2 * (fb-fa)
    bot = (b-a) * (fb-fc) - (b-c) * (fb-fa)
    return b - 0.5*(top/bot)

def make_bracket(func, bracket, w=(1.+np.sqrt(5))/2, dist_thresh=100, max_iter=10000):
    """Given two points [a, b], attempts to return a bracket triplet
    [a, b, c] such that f(a) > f(b) and f(c) > f(b).
    Note we only compute f(d) once for each point to save computing time"""
    a, b = bracket
    fa, fb = func(a), func(b)
    direction = 1 # Indicates if we're moving right or left
    if fa < fb:
        # Switch the two points
        a, b = b, a
        fa, fb = fb, fa
        direction = -1 # move to the left

    c = b + direction * (b - a) *w
    fc = func(c)

    for i in range(max_iter):
        if fc > fb:
            return np.array([a, b, c])  , i+1
        d = parabola_min_analytic(a, b, c, fa, fb, fc)
        fd = func(d)
        if np.isnan(fd):
            print(f'New point d:{d} gives fd:{fd}. Breaking function')
            return np.array([a,b,c]), i+1
        # We might have a bracket if b < d < c
        if (d>b) and (d<c):
            if fd > fb:
                return np.array([a, b, d]), i+1
            elif fd < fc:
                return np.array([b, d, c]), i+1
            # Else we don't want this d
            #print('no parabola, in between b and c')
            d = c + direction * (c - b) * w
        elif (d-b) > 100*(c-b): # d too far away, don't trust it
            #print('no parabola, too far away')
            d = c + direction * (c - b) * w
        elif d < b:
            pass#print('d smaller than b')

        # we shifted but didn't find a bracket. Go again
        a, b, c = b, c, d
        fa, fb, fc = fb, fc, fd

    print('WARNING: Max. iterations exceeded. No bracket was found. Returning last values')
    return np.array([a, b, c]), i+1

def golden_section_search(func, bracket, target_acc=1e-5, max_iter=int(1e5)):
    """Once we have a start 3-point bracket surrounding a minima, this function iteratively
    tightens the bracket to search of the enclosed minima using golden section search."""
    w = 2. -  (1.+np.sqrt(5))/2 # 2 - golden ratio
    a, b, c = bracket
    fa, fb, fc = func(a), func(b), func(c)

    for i in range(max_iter):
        # Set new point in the largest interval
        # We do this separately because the bracket propagation can just not be generalized sadly
        if np.abs(c-b) > np.abs(b-a): # we tighten towards the right
            d = b + (c-b)*w
            fd = func(d)
            if fd < fb: # min is in between b and c
                a, b, c = b, d, c
                fa, fb, fc = fb, fd, fc
            else: # min is in between a and d
                a, b, c = a, b, d
                fa, fb, fc = fa, fb, fd
        else: # we tighten towards the left
            d = b + (a-b)*w
            fd = func(d)
            if fd < fb: # min is in between a and b
                a, b, c = a, d, b
                fa, fb, fc = fa, fd, fb
            else: # min is in between d and c
                a, b, c = d, b, c
                fa, fb, fc = fd, fb, fc

        if np.abs(c-a) < target_acc:
            return [b,d][np.argmin([fb, fd])], i+1 # return the x point corresponding to the lowest f(x)

    print("Maximum Number of Iterations Reached")
    return b, i+1
```

Listing 5: Code for the maximization algorithm

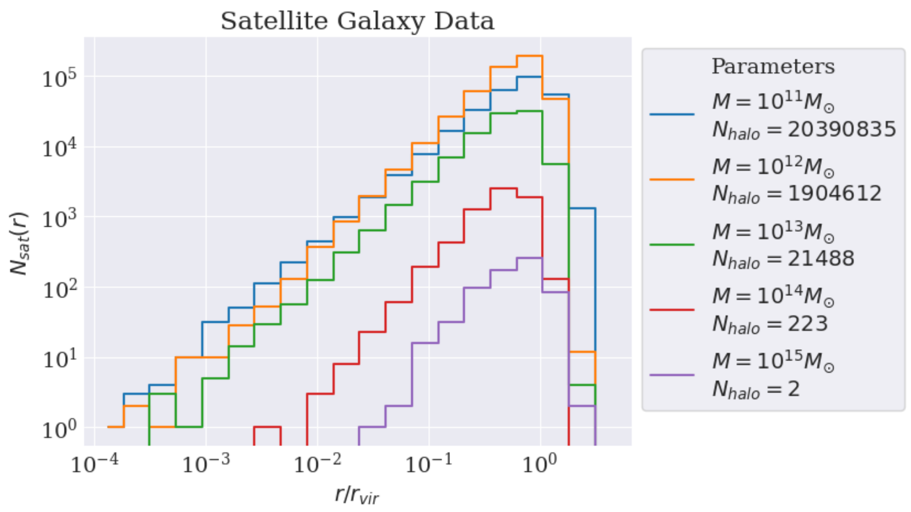


Figure 2: Binned satellite galaxy distance data divided into five mass bins of their massive galaxy halo as indicated on the right. The total number of halos over which the satellite galaxies are distributed is also indicated on the right.

1.2 Data Fitting

We import the five datafiles provided for this hand-in which contain satellite galaxies for halos in increasing mass bins ranging from $10^{11}M_\odot$ up to $10^{15}M_\odot$. Each file contains a number of halos, N_{halo}, and the distances of each satellite galaxy from its center massive galaxy. For each dataset we can immediately compute $\langle N_{sat}\rangle$ by dividing the total number of satellite galaxies by N_{halo}. We bin the data in 20 bins from $x = 10^{-4}$ to $x = 5$ in log-space. We set the lower limit above $x = 0$ for computational reasons because equation 2 is not defined there. We opt to show the distributions in log-space to better visualize the low x region. The decision for 20 bins is relatively arbitrary but appears to result in plots that are easily interpretable. We show the binned data together with N_{halo} for each dataset in Figure 2.

In the next two subsections we will first describe our two approaches to fit equation 2 to these distributions. In Section 1.3 we present the results of our fitting algorithms applied to the data, combined with a brief statistical analysis.

```python
def readfile(filename):
    """Code to read in the halo data, copied from the hand in instructions:
    https://home.strw.leidenuniv.nl/~daalen/Handin_files/satellites2.py"""
    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=np.float64)
    f.close()
    return radius, nhalo  # Return the virial radius for all the satellites in the file, and the number of halos
def make_plot_alldata():
    """Make a plot showcasing all raw, binned data for the report"""
    basename = 'data/satgals_m1'
    xmin = 1e-4 # cannot take zero because it messes with log and powers
    xmax = 5
    Nbins = 20
    do_log = True
    fig, ax = plt.subplots(1,1)

    for i in range(1, 6):
        radius, nhalo = readfile(f'{basename}{i}.txt')
        n, bin_edges = hist(radius, xmin, xmax, Nbins, do_log)

        Nsat = len(radius)/nhalo
        bin_centers = np.zeros(len(bin_edges)-1)
        for j in range(len(bin_centers)):
            bin_centers[j] = bin_edges[j] + 0.5*(bin_edges[j+1] - bin_edges[j])

        ax.step(bin_centers, n, where='mid', label=rf'$M=10^{{{i+10}}}M_{{\odot}}$'+'\n'+rf'$N_{{halo}} = {nhalo}$')#+'\n'+rf'$\left<N_{{sat}}\right>= {Nsat:.3E}$')

    xlim, ylim = ax.get_xlim(), ax.get_ylim() # For future plotting of individual sets
    ax.set_xlabel(r'$r/r_{vir}$')
    ax.set_ylabel(r'$N_{sat}(r)$')
    ax.set_title('Satellite Galaxy Data')

    ax.set_xscale('log')
    ax.set_yscale('log')
    plt.legend(title='Parameters', bbox_to_anchor=(1, 1), frameon=True, fancybox=True)
    plt.savefig('results/satellite_data.png', bbox_inches='tight')

    return xlim, ylim
```

Listing 6: Code to import data, and make the plot of Figure 2

```python
def fit_data():#xlim, ylim):
    """Code for Q1b-d"""
    basename = 'data/satgals_m1'
    xmin = 1e-4 # cannot take zero because it messes with log and powers
    xmax = 5
    Nbins = 20
    do_log = True
    no_bins = False
    guess = np.array([2.4, 0.25, 1.5])

    fitres_txt = ""
    full_fitres_txt = ""
    stats_txt = ""
    fig, axs = plt.subplots(3, 2, sharex=True, sharey=True, tight_layout=True, figsize=(8, 12))

    # do all of the below for each dataset separately
    for i in range(1,6):
        ax = axs.flatten()[i-1]
        radius, nhalo = readfile(f'{basename}{i}.txt')
        n, bin_edges, bin_centers = hist(radius, xmin, xmax, Nbins, do_log, return_centers=True)

        Nsat = len(radius)/nhalo
        print(f'Imported Dataset M1{i}. {len(radius)} Objects.')

        # 1b. Start with fitting a chi squared distribution to this using the Levenberg-Marquardt algorithm
        # the biggest adaptation to it is that sigma is iteratively computed such that \sigma^2 = \mu
        print('Starting Chi Squared Fitting..')
        params_chi2, chi2, niter_chi2 = fit_satellite_data_chisq(bin_centers, n, Nsat, guess, bin_edges)
        print(f'\nChi Squared Fit:\n\Chi^2 = {chi2}\na, b, c = {params_chi2}\n')

        # 1c. Now fit a Poisson distribution to this data using the Quasi-Newton method
        print('Starting Poisson Fitting..')
        if no_bins:
            params_poisson, logL, niter_poisson = fit_satellite_data_poisson(radius, None, Nsat, guess, bin_edges, no_bins)
        else: # feed it only nbins data points if wanted
            params_poisson, logL, niter_poisson = fit_satellite_data_poisson(bin_centers, n, Nsat, guess, bin_edges, no_bins)
        print(f'\nPoisson Fit:\nlog L = {logL}\n<Nsat> = {Nsat}\na, b, c = {params_poisson[0]}, {params_poisson[1]}, {params_poisson[2]}\n')

        # Bin the Poisson and Chi squared models to match the datA
        xx = np.logspace(np.log10(xmin), np.log10(xmax), 100)
        chi2_binned = nhalo * compute_mean_satellites(xx, *params_chi2, bin_edges, Nsat)
        poisson_binned = nhalo * compute_mean_satellites(xx, *params_poisson, bin_edges, Nsat)

        # Statistical Tests
        DoF = Nbins - 4 # degrees of freedom

        # G-test for the chi squared model because it is binned
        # mask out all bins without observations: lim_O->0 [O ln(O/E)] = 0 for E != 0
        zero_mask = n != 0
        G_chi2 = 2. * np.sum(n[zero_mask] * np.log((n/chi2_binned)[zero_mask]))
        G_poisson = 2. * np.sum(n[zero_mask] * np.log((n/poisson_binned)[zero_mask]))
        Q_chi2 = (gammainc(DoF/2., G_chi2/2.)/gamma(DoF/2.))
        Q_poisson = (gammainc(DoF/2., G_poisson/2.)/gamma(DoF/2.))
        print(f'G_chi2 = {G_chi2}, G_poisson = {G_poisson}')
        print(f'Q_chi2 = {Q_chi2}, Q_poisson = {Q_poisson}')

        # Plotting
        ax.step(bin_centers, n, label='Data', where='mid')
        ax.scatter(bin_centers, n, c='black', marker='X', s=25, zorder=5, label='Fit Points')
        ax.step(bin_centers, chi2_binned, where='mid', label=r'$\
chi^2$ Fit', ls='--')
        ax.step(bin_centers, poisson_binned, where='mid', label='Poisson Fit', ls='--')

        ax.set_title(rf'M = $10^{{{i+10}}} M_{{\odot}}$')
        ax.set_xscale('log')
        ax.set_yscale('log')
        ax.set_ylim(10**-5.5, 10**5.5)
        ax.legend()

        fitres_txt += f'$10^{{{i+10}}}$ & {Nsat:.2E} & {params_chi2[0]:.2f} & {params_chi2[1]:.2f} & {params_chi2[2]:.2f} & {chi2:.2E} & \\\\ \n'
        fitres_txt += f' & & {params_poisson[0]:.2f} & {params_poisson[1]:.2f} & {params_poisson[2]:.2f} & & {logL:.2E}\\\\ \n'
        full_fitres_txt += f'$10^{{{i+10}}}$ & {Nsat} & {params_chi2[0]} & {params_chi2[1]} & {params_chi2[2]} & {chi2} & \\\\ \n'
        full_fitres_txt += f' & & {params_poisson[0]} & {params_poisson[1]} & {params_poisson[2]} & & {logL}\\\\ \n'
        stats_txt += f'$10^{{{i+10}}}$ & $\chi^2$ & {G_chi2} & {Q_chi2} \\\\ \n'
        stats_txt += f'$10^{{{i+10}}}$ & Poisson & {G_poisson} & {Q_poisson} \\\\ \n'

    # Figure Labels
    for ax in axs[:,0]:
        ax.set_ylabel(r'$N_{sat}(r)$')
    for ax in axs[-1]:
        ax.set_xlabel(r'$r/r_{vir}$')
    plt.suptitle('Fit Results')
    plt.savefig('results/fitresults.png', bbox_inches='tight')

    fitres_txt = fitres_txt[:-3] # remove the last '\\ '
    stats_txt = stats_txt[:-3]
    # Textfile writing
    with open('results/fitresults.txt', 'w') as file:
        file.write(fitres_txt)
        file.close()
    with open('results/stats.txt', 'w') as file:
        file.write(stats_txt)
        file.close()
```

Listing 7: Code that calls the fitting procedures and performs the statistical tests

1.2.1 Chi-Squared

These data are discrete counts, so therefore they should be fit by a Poisson distribution. However we start with an 'easy' χ^2 fit with Poisson variance (i.e. $\sigma^2 = \mu$) to compare to the proper unbiased fit. This means that we want to minimize the function

$$\chi^2 = \frac{(y_i - \mu(x_i|\boldsymbol{p}))}{\mu(x_i|\boldsymbol{p})}. \tag{3}$$

Where x_i and y_i are the bin center and bin counts respectively. μ is a function of the parameter describing the expected value in any particular bin, which is given by

$$\mu\left(x_i|\boldsymbol{p}\right)) = \tilde{N}_i = \int_{x_i}^{x_{i+1}} N(x)dx. \tag{4}$$

We minimize Equation 3 for each dataset separately using the Levenberg-Marquardt algorithm. As a good starting guess we use the same parameters as in Section 1.1, i.e. $a = 2.4$, $b = 0.25$, $c = 1.6$. Another important thing to note is that the normalization constant A is a function of these three parameters which keep changing. We therefore compute a new value for A each time the parameters are shift by first integrating over $N(x)$ from $x_{min} = 10^{-4}$ to $x_{max} = 5$ using Romberg integration and $A = 1$. We then compute A by dividing $\langle N_{sat}\rangle$ by the result of this integral.

```python
## CHI SQUARED FITTING ##
def compute_mean_satellites(x, a, b, c, bin_edges, Nsat):
    """Chi squared function specifically for the distribution of satellite
    galaxies around a massive central, n(x, ..) which we attempt to fit
    using the assumption of Poisson variance \sigma^2 = \mu. Therefore
    sigma is not used, but we need to pass it for function interoperability"""
    # mean = variance = int(N(x))dx over the bin i
    N_fit = lambda x: N(x, 1, Nsat, a, b, c)
    integral = romberg_integration(N_fit, bin_edges[0], bin_edges[-1], 8)
    A = Nsat/integral
    N_fit = lambda x: N(x, A, Nsat, a, b, c)

    if len(x) == 1: # Evaluate only at a single data point
        # the np.min clause is to ensure we never get an index errors
        bin_idx = np.min([np.argmin(bin_edges-x), len(bin_edges)])
        return romberg_integration(N_fit, bin_edges[bin_idx], bin_edges[bin_idx+1], 8)

    mean_ar = np.zeros(len(bin_edges)-1)
    for i in range(len(mean_ar)):
        mean_ar[i] = romberg_integration(N_fit, bin_edges[i], bin_edges[i+1], 8)

    return mean_ar

def fit_satellite_data_chisq(bin_centers, n, Nsat, guess, bin_edges):
    """Function applying the Levenberg-Marquadt algorithm to implement the
    'easy' fit to the data, with some slight modifications"""
    # Need to add in the lambda function so we can pass in the bin_edges we found

    mean_func = lambda x, a, b, c: compute_mean_satellites(x, a, b, c, bin_edges, Nsat)

    # Fit 'fit_func' to the data using a minimiztation of chi^2 defined by chisq_func. It doesn't matter what values
    # we use for sigma, because we will never use it. We set it to 0 here to ensure it's never used
    return levenberg_marquardt(bin_centers, n, None, mean_func, guess, linear=False,
                                    chisq_like_poisson=True)
```

Listing 8: Code for the chi-squared fitting computing $\tilde{N}_i$ and calling the Levenberg-Marquardt algorithm

```
def determine_implicit_pivot_coeff(mat):
    """Determines the coefficients for implicit pivotting in Crout's Algorithm. It does this by finding
       the absolute maximum value of each row in the matrix, and storing its inverse.

       NOTE: Requires a Matrix object (this script) as input. This ensures correspondence with row_order
    """
    row_max_inverse = np.zeros(mat.num_rows)
    for i in range(mat.num_rows):
        row = mat.matrix[i]
        row_max = row[np.argmax(np.abs(row))]
        row_max_inverse[i] = 1. / row_max

    return row_max_inverse


def lu_decomposition(coefficients, implicit_pivoting=True, epsilon=1e-13):
    """Decomposes a matrix into:
        -L: A matrix with non-zero elements only in the lower-triangle, and ones on the diagonal
        -U: A matrix with non-zero elements only in the upper-triangle, including the diagonal
       These matrices are presented and stored into one.
       The decomposition is done using Crout's Algorithm
    """
    if type(coefficients) == np.ndarray:
        A = Matrix(values=coefficients)
    else:
        A = coefficients

    # Combat round-off erors to dodge division by zero
    A.matrix[np.abs(A.matrix)<epsilon] = epsilon

    if implicit_pivoting:
        row_max_inverse = determine_implicit_pivot_coeff(A)

    imax_ar = np.zeros(A.num_columns)
    # First pivot the matrix
    for i in range(A.num_columns):
        # A.matrix[i:, i] selects all elements on or below the diagonal
        if implicit_pivoting:
            pivot_candidates = A.matrix[i:, i] * row_max_inverse[i:]
        else:
            pivot_candidates = A.matrix[i:, i]

        pivot_idx = i + np.argmax(np.abs(pivot_candidates))
        imax_ar[i] = pivot_idx
        A.swap_rows(i, pivot_idx)

    for i in range(A.num_columns):
        # A.matrix[i:, i] selects all elements on or below the diagonal
```

```python
        diag_element = A.matrix[i, i]  # Use to scale alpha factors

        for j in range(i + 1, A.num_rows):  # This leaves a zero at the end, not the best fix this!
            A.matrix[j, i] /= diag_element
            for k in range(i + 1, A.num_rows):  # j+1):
                A.matrix[j, k] -= A.matrix[j, i] * A.matrix[i, k]

    return A


def solve_lineqs_lu(LU, b):
    """"Performs the steps to solve a system of linear equations after a matrix A has been LU decomposed. It
    does this by first applying forward substitution to solve Ly = b, and then applies backward subsitutttion
    to solve Ux = y.

    Inputs:
        LU: The decomposed L and U matrices, stored in a single Matrix instance
        b: The constraints of the linear equations, ndarray

    Outputs:
        x: Matrix instance containing the solution such that Ax = b
    """
    if type(b) == np.ndarray:
        x = Matrix(values=b)
    else:
        x = b
    # Begin by swapping the x's in the right order
    x.matrix = x.matrix[LU.row_order]

    # Forward Subsitutions. Solves Ly = b
    for i in range(0, x.num_rows):
        x.matrix[i] -= np.sum(LU.matrix[i, :i] * x.matrix[:i])

    # Backward Substitutions. Solves Ux = y
    for i in range(x.num_rows-1, -1, -1):
        x.matrix[i] = (1./LU.matrix[i,i])*(x.matrix[i] - np.sum(LU.matrix[i, i+1:]*x.matrix[i+1:]))

    return x

def outer_product(v, w):
    """Compute the outer product of two vectors. This is a matrix A with
            A_ij = v_i * w_j
    NOTE: This function doesn't assume the vectors are of the same size, and
    this function is not symmetric (outer(v,w) != outer(w,v))
    """
    A = np.zeros((v.shape[0], w.shape[0]))
    for i in range(v.shape[0]):
        A[i] = v[i] * w
    return A


class Matrix():
    """Matrix Class for linear algebra"""

    def __init__(self, values=None, num_rows=None, num_columns=None
, dtype=np.float64):
        """Check inputs and create a corresponding matrix or vector"""
        if values is not None:
            self.num_rows = values.shape[0]
            try:
                self.num_columns = values.shape[1]
            except IndexError:
                self.num_columns = 1
                #print(f'Warning! Values has dim=1. Making vector with shape ({self.num_rows}, {self.num_columns})')
            if type(values) == np.ndarray:
                self.matrix = np.array(values, dtype=dtype)
            else:
                print(f'Datatype of values {type(values)} not recognized. Initializing matrix with zeros.')
                self.matrix = np.zeros((num_rows, num_columns), dtype=dtype)
        else:
            self.num_rows = num_rows
            self.num_columns = num_columns
            self.matrix = np.zeros((num_rows, num_columns))

        # Use row order to track rows that have been shuffled
        self.row_order = np.arange(self.num_rows)

    def swap_rows(self, idx1, idx2):
        """Extract rows from a matrix, and switch them. Track the change in row_order"""
        self.matrix[[idx1, idx2]] = self.matrix[[idx2, idx1]]
        self.row_order[[idx1, idx2]] = self.row_order[[idx2, idx1]]

    def scale_row(self, idx, scalar):
        """Multiply all elements of row {idx} by a factor {scalar}"""
        self.matrix[idx] *= scalar

    def add_rows(self, idx1, idx2, scalar):
        """Add row {idx2} multiplied by scalar to row {idx1}"""
        self.matrix[idx1] += scalar * self.matrix[idx2]


def make_param_func(params, i):
    """Given a list of parameters and an index i, return a function with
    p_i as the variable for use in differentation algorithms"""
    # should be a better way to do this right
    first_half_p = params[:i]
    if not i == len(params)-1: # Avoid indexing errrors
        second_half_p = params[i+1:]
    else:
        second_half_p = []
    return lambda p: [*first_half_p, p, *second_half_p]


def make_alpha_matrix(xdata, sigma, func, params,
                      h_start=0.1, dec_factor=2, target_acc=1e-10): #derivative params
    """Make a Matrix object containing the sum of N products of derivatives
    where the element i,j is the product of df/dxi and df/dxj. Each value i
    can be weighted by its uncertainty sigma if desired. If this is not
    required one can set sigma = 1 to 'ignore' this step"""
    N = len(xdata) # Number of data points
    M = len(params) # Number of parameters
    A = Matrix(num_columns=M, num_rows=M)

    func_derivatives = np.zeros((M, N))

    # Build up all M derivatives
    for i in range(M):
        param_func = make_param_func(params, i)
        # Adjust Ridders method to do this in one go? Big speed upgrade.
        for j in range(N):
            yp = lambda p: func([xdata[j]], *param_func(p))
            dy_dpi, _ = ridders_method(yp, [params[i]], h_start, dec_factor, target_acc)
            func_derivatives[i][j] = dy_dpi

    # Build up A-matrix
    for i in range(M):
        A.matrix[i][i] = alpha_kl(func_derivatives[i], func_derivatives[i], sigma)
        for j in range(i):
            A.matrix[i][j] = alpha_kl(func_derivatives[i], func_derivatives[j], sigma)
            A.matrix[j][i] = A.matrix[i][j]

    return A


def compute_chi_sq(x, y, sigma, func, params):
    """Compute the chi squared value between N points x, y with
    y uncertainty sigma and a function func with parameters params
    Setting sigma = 1 reduces this to just lesat squares"""
    return np.sum(((y - func(x, *params))**2)/(sigma*sigma))

def compute_chi_sq_likepoisson(x, y, func, params):
    """Compute the chi squared value between N points x, y with
    y uncertainty sigma and a function func with parameters params
    under a Poisson distribution assumption, i.e. \sigma = \mu"""
    mean = func(x, *params)
    return np.sum(((y - mean)**2) / (mean))


def make_nabla_chi2(xdata, ydata, sigma, func, params,
                    h_start=0.1, dec_factor=2, target_acc=1e-5,
                    chisq_like_poisson=False):
    """"""
    M = len(params)
    chisq_derivatives = np.zeros(M)
    for i in range(M):
        param_func = make_param_func(params, i)
        if chisq_like_poisson:
            chi2_func_p = lambda p: compute_chi_sq_likepoisson(xdata, ydata, func, param_func(p))
        else:
            chi2_func_p = lambda p: compute_chi_sq(xdata, ydata, sigma, func, param_func(p))
        dchi_dpi, _ = ridders_method(chi2_func_p, [params[i]], h_start, dec_factor, target_acc)
```

```python
        chisq_derivatives[i] = dchi_dpi

    return beta_k(chisq_derivatives)


def weigh_A_diagonals(A, lmda):
    """"Weigh the diagonal elements of a square matrix A by a factor (1+lmda)"""
    if not A.matrix.shape[0] == A.matrix.shape[1]:
        raise ValueError(f"This Matrix object is not square: {A.matrix}")
    for i in range(A.matrix.shape[0]):
        A.matrix[i][i] *= (1. + lmda)
    return A


def alpha_kl(dydp1, dydp2, sigma):
    """"""
    return np.sum((1./(sigma**2.)) * dydp1 * dydp2)

def beta_k(dchi_dp):
    """"""
    return -0.5 * dchi_dp


def levenberg_marquardt(xdata, ydata, sigma, func, guess, linear=True,
                        w=10, lmda=1e-3, chi_acc=1e-3, max_iter=int(1e2),
                        epsilon = 1e-13, # fit procedure params
                        chisq_like_poisson=False,
                        h_start=0.1, dec_factor=2, target_acc=1e-13): # derivative params
    """"""

    if chisq_like_poisson:
        # sqrt becaues it computes the mean
        sigma = np.sqrt(func(xdata, *guess))
        if np.isnan(sigma).any():
            raise ValueError(f'NaN in sigma {sigma}')
        chi2 = compute_chi_sq_likepoisson(xdata, ydata, func, guess)
    else:
        chi2 = compute_chi_sq(xdata, ydata, sigma, func, guess)

    N = len(xdata) # Number of data points
    M = len(guess) # Number of parameters
    b = Matrix(num_columns=1, num_rows=M)
    params = guess

    # Can do this beforehand because the derivatives never change
    # if the functions depend linearly on the parameters
    if linear:
        A = make_alpha_matrix(xdata, sigma, func, params, h_start, dec_factor, target_acc)

    for iteration in range(max_iter):
        if linear:
            A_weighted = copy.deepcopy(A) # ensure no pointing goes towards A
            A_weighted = weigh_A_diagonals(A_weighted, lmda) # Make \alpha_prime
        else:
            A = make_alpha_matrix(xdata, sigma, func, params, h_start, dec_factor, target_acc)
            # Combat round-off errors and divisions by zero
            A.matrix[np.abs(A.matrix)<epsilon] = epsilon
            A_weighted = weigh_A_diagonals(A, lmda)


        b.matrix = make_nabla_chi2(xdata, ydata, sigma, func, params, h_start, dec_factor, target_acc)

        # Solve the set of linear equations for \delta p with LU decomposition
        LU = lu_decomposition(A_weighted, implicit_pivoting=True)
        delta_p = solve_lineqs_lu(LU, b).matrix

        # Evaluate new chi^2
        new_params = params + delta_p.flatten()

        if chisq_like_poisson:
            new_sigma = np.sqrt(func(xdata, *new_params))
            new_chi2 = compute_chi_sq_likepoisson(xdata, ydata, func, new_params)
        else:
            new_chi2 = compute_chi_sq(xdata, ydata, sigma, func, new_params)

        delta_chi2 = new_chi2 - chi2

        if delta_chi2 >= 0 or not np.isfinite(new_chi2): # reject the solution
            lmda = w*lmda
            #print(f'delta = {delta_chi2}. Reject. lambda = {lmda:.2E} chi2 = {chi2}')
            continue

        if np.abs(delta_chi2) < chi_acc:
            return params, new_chi2, iteration+1 # converged!

        # accept the step and make it
        params = new_params
        chi2 = new_chi2
        lmda = lmda/w
        if chisq_like_poisson:
            sigma = new_sigma
        #print(f'delta = {delta_chi2}. Accept. lambda = {lmda:.2E} chi2 = {chi2}')

    print("Max Iterations Reached")
    return params, new_chi2, iteration+1
```

Listing 9: Code for the Marquardt-Levenberg algorithm

1.2.2 Poisson

For the Poisson fit we have two options: we can either use the same binned data as is used for the χ^2 fit, or we can opt to not use bins at all. In this latter case we pretend as though we did bin the data, however with bin size sufficiently small such that each bin contains either 0 or 1 objects. Doing this allows us to make better use of the data at the cost of more computational load. In the worst case, namely the first dataset, this difference is a factor on the order of 10^4. Unfortunately while implemented, we have been unable to use this binless fitting as computing time exceeds the maximum alloted time of ten minutes. Therefore we opt to fit the data here using the same bins as in the previous section. The log-likelihood of a Poisson distribution is given as

$$-\ln \mathcal{L}(\boldsymbol{p}) = -\sum_{i=0}^{N-1} \left(y_i \ln\left[\mu(x_i|\boldsymbol{p})\right)\right] - \mu(x_i|\boldsymbol{p})) - \ln(y_i!)) \,. \tag{5}$$

When minimizing this equation we can ignore the last term, $\ln(y_i!)$ because it is independent of the parameters we are fitting and therefore constant. We are then left with only the first two terms in the sum. For the minimization we will make use of the downhill simplex method where we initialize one of the vertices of the first simplex using the same parameters as before ($a = 2.4$, $b = 0.25$, $c = 1.6$). We initialize the other three vertices by adding one to parameter i for vertex i.

We also attempted to implement the Quasi-Newton method to minimize the log likelihood function, however this also has proven unsuccesful due to extremely steep gradients which often led the line minimization algorithm to regions in parameter space where the likelihood function is not defined. This only led to NaN values as a result for the parameters. While not used, the code is still included at the end of this report.

```python
## POISSON FITTING ##
def poisson_fit_func(x, y, delta_x, bin_edges, Nsat, params, xmin, xmax, no_bins):
    """Procedure to be iteratively called in quasi_newton to fit a Poisson
    distribution to the satellite galaxy data. Adjusted from the chi^2 fit
    for optimization reasons. delta_x is the smallest non-zero difference
    between two x used as a proxy for bin size"""
    # Start by computing A corresponding to these a, b, c like handin 2

    N_fit = lambda x: N(x, 1, Nsat, *params)
    integral = romberg_integration(N_fit, xmin, xmax, 8)
    A = Nsat/integral
    N_fit = lambda x: N(x, A, Nsat, *params)

    mean_ar = np.zeros(len(x))
    for i in range(len(mean_ar)):
        if no_bins:
            int_min, int_max = x[i] - delta_x, x[i]+delta_x
        else:
            int_min, int_max = bin_edges[i], bin_edges[i+1]

        mean_ar[i] = romberg_integration(N_fit, int_min, int_max, 8)

    if no_bins:
        ll = -1.*np.sum(np.log(mean_ar)) # plus an integral we take as constant
    else:
        ll = -1.*np.sum(y*np.log(mean_ar) - mean_ar) # plus a factor of y! which is constant

    return ll

def fit_satellite_data_poisson(x, y, Nsat, guess, bin_edges,
    no_bins):
    """Computes the Poisson likelihood of a function in the limit
    where the data is essentially unbinned. This function
    automatically
    computes the binsize required to obtain this.
    mean_func should be the function that computes the mean of the
    distribution, which should of course be linked to the fit
    function"""

    if no_bins:
        # Find the smallest difference between two x neighbouring x
    . Need to sort the
        # array first to find this. Sorting immediately gives us
    max(x) and min(x) as well
        # NOTE the below could have been done a lot quicker with
    numpy!
        x_sorted = merge_sort(x)
        diff_x = x_sorted[1:] - x_sorted[:-1]
        diff_sorted = merge_sort(diff_x)
        smallest_diff = diff_sorted[diff_sorted >0][0] # smallest
    non-zero element
        # This smallest difference sets the "bin size"
    else:
        smallest_diff = None

    # Fitting function and procedure
    fit_func = lambda p: poisson_fit_func(x, y, smallest_diff,
    bin_edges, Nsat, p, bin_edges[0], bin_edges[-1], no_bins)
    #fit_params, n_iter = quasi_newton(fit_func, guess)
    fit_params, n_iter = downhill_simplex(fit_func, guess)
    logL = fit_func(fit_params)
    return fit_params, logL, n_iter
```

Listing 10: Code to compute the Poisson likelihood, and to call the minimization functions

```
# CODE FOR THE N-DIMENSIONAL DOWNHILL SIMPLEX
def compute_centroid(A):
    """Compute the centroid of N points in N dimensional space. x
    should
    be an NxN array of N vectors with N dimensions (in that order).
     The function
    then returns one ndarray of length N with the centroid
    coordinates."""
    return (1./A.shape[1]) * np.sum(A, axis=0)


def downhill_simplex(func, start, shift_func=lambda x: x+1,
    max_iter=int(1e5), target_acc=1e-10):
    """Finds the minimum of a function using the downhill simplex
    method
    INPUT:
        func: A function taking only one variable as input with
    dimension N
        start: N-Dimensional numpy array where the function starts
    searching for a minimum
        shift_func: A function taking only one float as input
    dictating how to mutate the
                        initial simplex vertices
```

```python
    """
    dim = start.shape[0] # = N
    # Store N+1 vertice vectors in this matrix. This ordering fails if we feed it directly to func,
    # but it allows us to choose a vertex as vertices[i]. The function problem we solve by just
    # transposing this this matrix.
    vertices = np.zeros((dim+1, dim))
    func_vals = np.zeros(dim+1) # Store the f(X) values in here

    # Create the simplex, add slight variation to each vector except the first using 'shift_func'
    vertices[0] = start
    func_vals[0] = func(vertices[0])

    for i in range(dim):
        vertices[i+1] = vertices[0]
        vertices[i+1][i] = shift_func(vertices[i+1][i])
        func_vals[i+1] = func(vertices[i+1])

    # Start algorithm
    for i in range(1, max_iter+1):
        # Sort everything by function value
        sort_idxs = merge_sort(key=func_vals)
        vertices = vertices[sort_idxs]
        func_vals = func_vals[sort_idxs]
        #print(f'Current best logL = {func_vals[0]} at ', vertices[0])

        # Check if we have reached our accuracy level by comparing the best and worst function evals.
        accuracy =(np.abs(func_vals[-1] - func_vals[0])/np.abs(0.5*(func_vals[-1] + func_vals[0])))
        if accuracy < target_acc:
            return vertices[0], i # corresponds to func_vals[0], so the best point

        # Compute the centroid of all but the last (worst) point
        centroid = compute_centroid(vertices[:-1])

        # Try out a new points
        x_try = 2.* centroid - vertices[-1]
        f_try = func(x_try)

        if f_try < func_vals[-1]:
            # There is improvement in this step
            if f_try < func_vals[0]:
                # We are the best point. Try expanding
                x_exp = 2*x_try - centroid
                f_exp = func(x_exp)
                if f_exp < f_try:
                    # expanded point is even better. Replace x_N
                    vertices[-1] = x_exp
                    func_vals[-1] = f_exp
                else:
                    # x_try was good, x_exp is not better
                    vertices[-1] = x_try
                    func_vals[-1] = f_try
            else:
                # Better than x_N, not better than x_0. Just accept the point
                vertices[-1] = x_try
                func_vals[-1] = f_try

        else:
            # This point is worse than what we had. First try contracting, new x_try
            x_try = 0.5*(centroid+vertices[-1])
            f_try = func(x_try)
            if f_try < func_vals[-1]:
                # contracting improved x
                vertices[-1] = x_try
                func_vals[-1] = f_try
            else:
                # Nothing worked, just contract all points towards the best points
                vertices = 0.5*(vertices[0] + vertices) # x0 doesnt shift here: 0.5(x0 + x0) = x0
                # need to evaluate all but x0 because they shifted
                for i in range(dim):
                    func_vals[i+1] = func(vertices[i+1])

    print('Maximum Number of Evaluations Reached')
    return vertices[0], i
```
Listing 11: Downhill simplex code

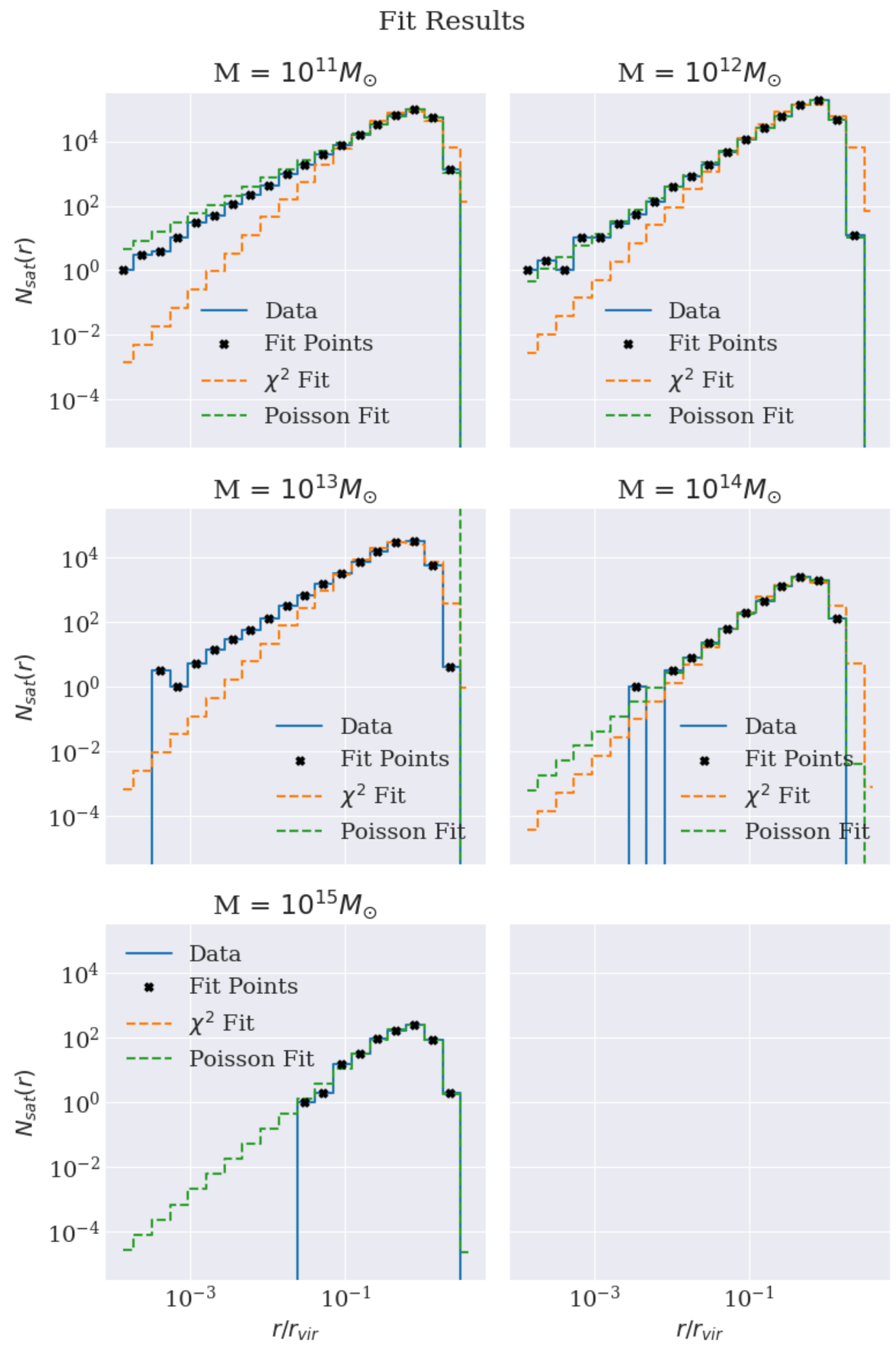
1.3 Fit Results

We apply the fitting methods described in the previous sections to each of the datasets `satgals_m1i.txt` with $i \in [1, 5]$, and present the results in Figure 3. We also include the parameters and χ^2 or $\ln \mathcal{L}$ values in Table 1

M	$\langle N_{sat} \rangle$	a	b	c	χ^2	$\ln \mathcal{L}$
10^{11}	1.37E-02	2.40	0.27	1.01	6.64E+12	
		1.18	1.20	3.64		1.67E+06
10^{12}	2.51E-01	2.40	0.26	1.03	9.92E+11	
		1.55	0.94	3.63		1.40E+06
10^{13}	4.37E+00	2.40	0.26	1.11	2.20E+09	
		0.44	1.22	3.92		-7.97E+05
10^{14}	2.91E+01	2.42	0.31	1.30	1.50E+06	
		1.93	0.61	2.55		-1.22E+04
10^{15}	3.30E+02	0.12	0.06	0.08	NAN	
		2.00	0.69	1.98		-2.47E+03

Table 1: Numerical results of both fitting procedures. For each of the five datasets we first present the resutls of the χ^2 fit, and the the results of the Poisson fit, this is clearly visible through which field out of χ^2 or $\ln \mathcal{L}$ is filled in. The values provided here are rounded to only 2 significant digits to let them fit on the page. However the full values are given at the end of this report.

In the figure we can see that the χ^2 fits appear to have performed quite badly for all datasets, underestimating the amount of satellite galaxies at low x and exteremely overestimating this in the highest x bin in the first three datasets. In the last two sets this pattern appears reversed and the peaks of the data- and model distributions do not match anymore. This is possibly caused by the fact



Fit Results
M = $10^{11}M_\odot$
M = $10^{12}M_\odot$
M = $10^{13}M_\odot$
M = $10^{14}M_\odot$
M = $10^{15}M_\odot$
$N_{sat}(r)$
r/r_{vir}
10^4
10^2
10^0
10^{-2}
10^{-4}
10^{-3}
10^{-1}
Data
Fit Points
χ^2 Fit
Poisson Fit

Figure 3:

that there is almost no data below $x \sim 10^{-2}$ which confuses the χ^2 statistic. The Poisson fits on the other hand appear to fit quite well to the data. In most of the plots the Poisson fit appears to at most be off by a few from the real data which could just be due to noise. Only in the $10^{11}M_\odot$ fit can we see that the Poisson fit appears to be biased towards higher values in the low x end, and appears to be 'off' by a few hundred to a thousand in the highest x bin.

But to draw some quantitative conclusions about these fits we have to perform some statistical test. We will do a G-test on the binned fit results and binned data. This G statistic is defined as

$$G = 2\sum_i^N O_i \ln\left(\frac{O_i}{E_i}\right). \tag{6}$$

Where O_i is the observed number of instances in a bin, which should always be an integer, and E_i is the expected number of observations in a bin. This latter value is the expected model value in a given bin, and therefore is not limited to being an integer. We can use this G statistic in a goodness of fit test for χ^2 to compute the Q-value, which is equivalent to the p-value. This Q is defined as

$$Q = 1 - \frac{\gamma\left(\frac{k}{2}, \frac{x}{2}\right)}{\Gamma\left(\frac{k}{2}\right)} \tag{7}$$

With

$$\gamma(s,x) = \int_0^x t^{s-1}e^-tdt \tag{8}$$

$$\Gamma(z) = \int_0^\infty t^{z-1}e^{-t}dt \tag{9}$$

the lower incomplete gamma function, and the gamma function respectively which we compute using `scipy.special.gammainc` and `scipy.special.gamma`. k is the number of degrees of freedom in the data, which is given by $N - M$ where N is the number of data points and M is the number of free parameters. In our case N is the number of data bins which we have set to 20, and M is the number of parameters we fit, 3, but also $\langle N_{sat}\rangle$ which sets the area under the curve. This means $M = 4$ and therefore the total number of degrees of freedom $k = 16$. We compute the G and Q values for each fit on each dataset, and present the results in Table 2.[1]

A Q value lower than $\sim$ 0.1 would indicate that our fit is bad. Unfortunately, this appears to be the case for all of the fits computed in this report. Most of the Q-values reported here take on the exact same value, which is an indication that our G values are extremely high and therefore also statistically extremely bad fits. We see one negative value in the G value for one of the Poisson fits, this in principle should never happen. Especially not a negative number that is as large as this is. This is another strong indication that there is most likely something wrong with the fit for that particular dataset.

[1]The code for this section was already given in Section 1.2 because it is done in the same loop as all the fitting procedures.

M	Method	G	Q
10^{11}	χ^2	28175.43917756349	0.0001984126984126984
10^{11}	Poisson	1087.0882149857707	0.0001984126984126984
10^{12}	χ^2	47702.887489199704	0.0001984126984126984
10^{12}	Poisson	5418.609065069538	0.0001984126984126984
10^{13}	χ^2	6250.032384770378	0.0001984126984126984
10^{13}	Poisson	-1811729.4716788493	nan
10^{14}	χ^2	344.76724586865157	0.0001984126984126984
10^{14}	Poisson	92.53161867435563	0.00019841269841252922
10^{15}	χ^2	nan	nan
10^{15}	Poisson	15.368470301292744	9.963597234230879e-05

Table 2: G-test values for each dataset and each fitting method.

The only thing one might be able to conclude from these statistics is the fact that, while the fits look reasonably good when inspected visually, when analyzing them quantiatively, with the G-statistic, they are actually quite bad. Although it is also important to note here that another statistic might have yielded better, or at least different, results than what is presented here.

Listing 12: Unrounded numerical values for the fitting parameters presented in Table 1.

```python
# QUASI-NEWTON
def line_minimization(func, x_vec, step_direction, method=golden_section_search, minimum_acc=0.1):
    """"""
    # Make a function f(x+lmda*n)
    minim_func = lambda lmda: func(x_vec + lmda * step_direction)

    # The parameter landscape is very prone to diverging, and the gradients are very steep. Attempt to keep steps small!
    inv_stepdirection = np.abs(1./np.sum(step_direction)) # roughly equal to 1
    bracket_edge_guess = [0, 1]#inv_stepdirection]  # keeps the steps realatively small to combat divergence
    bracket, _ = make_bracket(minim_func, bracket_edge_guess) # make a 3-point bracket surrounding a minimum

    # Use a 1-D minimization method to find the 'best' lmda
    minimum, _ = method(minim_func, bracket, target_acc=minimum_acc)
    return minimum



def compute_gradient(func, x_vec):
    """Computes the gradient of a multi-dimensional function by applying
    Ridder's method on each dimension separately"""
    dim = x_vec.shape[0]
    nabla_f = np.zeros(dim)

    for i in range(dim):
        # The function below transforms the multi-dimensional function func
        # into a function that only varies along dimension i
        func_1d = lambda xi: func([*x_vec[:i], xi, *x_vec[i+1:]])
        nabla_f[i] = ridders_method(func_1d, [x_vec[i]], target_acc=1e-5)[0][0] # we don't store the uncertainty now
    return nabla_f


def bfgs_update(H, delta, D):
    """Updates the approximated Hessian using the Broyden-Fletcher-Goldfarb-Shannon
    algorithm, used for optimization with the quasi-Newton method.
    INPUTS:
        H: NxN ndarray, approximation of the Hessian
        delta: N ndarray, last taken optimization step in x_vec
        D: N ndarray, difference between new and old gradients
    OUTPUTS:
        H': NXN ndarray, updated approximation of the Hessian
    """
    # Pre-compute some values for efficiency and clarity
    deltaD = delta @ D
    HD = H @ D
    DHD = D @ HD

    u = (delta/deltaD) - (HD/DHD)

    H_update1 = outer_product(delta, delta)/deltaD
    H_update2 = outer_product(HD, HD)/DHD
    H_update3 = DHD * outer_product(u, u)
    return H + H_update1 - H_update2 + H_update3
```

```python


def quasi_newton(func, start, target_step_acc=1e-3, target_grad_acc=1e-3, max_iter=int(1e3)):
    """"""
    # SETUP
    dim = start.shape[0]
    H = np.eye(dim)
    x_vec = start
    # Do this before the loop because we compute the gradient at x_i+1 in loop i

    gradient = compute_gradient(func, x_vec)

    for i in range(max_iter):
        step_direction = -H @ gradient
        step_size = line_minimization(func, x_vec, step_direction)
        # Make the step
        delta = step_size * step_direction
        x_vec += delta
        #print(step_direction, step_size)
        # Check if we are going to  make a small step
        if np.abs(np.max(delta/x_vec)) < target_step_acc:
            return x_vec, i
        #print(gradient)
        # Compute the gradient at the new point, and check relative convergence
        new_gradient = compute_gradient(func, x_vec)
        if np.abs(np.max((new_gradient - gradient)/(0.5*(new_gradient+gradient)))) < target_grad_acc:
            return x_vec, i

        # If no accuracies are reached yet, sadly we have to continue
        D = new_gradient - gradient
        gradient = new_gradient
        H = bfgs_update(H, delta, D)

    return x_vec, i

def outer_product(v, w):
    """Compute the outer product of two vectors. This is a matrix A with
            A_ij = v_i * w_j
    NOTE: This function doesn't assume the vectors are of the same size, and
    this function is not symmetric (outer(v,w) != outer(w,v))
    """
    A = np.zeros((v.shape[0], w.shape[0]))
    for i in range(v.shape[0]):
        A[i] = v[i] * w
    return A
```

Listing 13: Code for the Quasi-Newton algorithm which was too prone to divergence and is therefore sadly unused in this report