Workflow Explanation Document

Cluster Membership Process

Sean McAdam
mcadams@wwu.edu

Contents

Chapter 1

Introduction to the Membership Process

The membership process takes a look at an observed set of data, and analyzes it to find any overdensities that are consistent with predictions made by a simulated cluster model.

1.1 Purpose and Scope of this document

- **Purpose:** To detail the membership workflow so that users can replicate and extend the work on this cluster and others.
- **Scope:** This guide details the required data and packages needed to run the process, and the details and recommended workflow of the process.

1.2 Document Structure

- **2. Prerequisites** – Lists the requirements packages and data.
- **3. Workflow Overview** – Summarizes the entire process.
- **4-8. Detailed Steps** – Goes step-by-step on how to use the notebooks.
- **9. Conclusion** – Wraps up the guide, and lists areas to be improved

Chapter 2

Prerequisites

This process involves a lot of plotting and tweaking parameters, so it's written in a series of Jupyter notebooks. Because the membership calculation can take upwards of 45-60 minutes, the program cannot be run via Google Colab due to runtime restrictions.

2.1 Required Python Packages

- **General functions:** `numpy`, `pandas`
- **Plotting:** `matplotlib`, `collections`
- **Math:** `shapely`, `scipy`, `math`
- **Astronomy specific:** `astropy` , `astroquery`

2.2 Required Data

1. **Observed data with the following values:**
   (a) RA (°), DEC (°), PMRA (mas/yr), PMDEC (mas/yr), Parallax (mas)
   (b) Magnitudes (SDSS 'ugriz' by default, but can work in other filter sets)
   (c) Extinction estimates
2. **Simulated cluster stars with same values as above**
3. **Isochrone based on cluster estimates**
4. **Supplementary dataset(s) to compare members with**

Chapter 3

Workflow Overview

1. **Finding the completeness limit:** Corrects magnitudes for extinction, makes a histogram of the observed stars for a chosen magnitude, and applies a best fit line to find where the star count begins falling off.

2. **Data processing:** Truncates the observed data down to the extinction-corrected completeness limit. Calculates and appends Galactic 'l' and 'b' coordinates to each star in the observed dataset.

3. **Field star extrapolation** Generates a set of simulated field stars with data extrapolated from the observed field star densities in 'l' and 'b' space.

4. **Membership calculation** counts stars in the 5D position-motion space, subtracts simulated from observed to reveal overdensities, and calculates membership probabilities

5. **Comparisons** Matches observed stars with stars from other datasets to compare membership probabilities with members from established literature or other targets

Chapter 4

Finding the completeness limit

As we look further away from us, we'd expect to see more stars, but the light that reaches us also becomes dimmer. The completeness limit tells us the magnitude where our sensor begins hitting the limit of its resolution, and not every star is recorded in the data.

Packages:

```
# General imports
import pandas as pd
import numpy as np

# Plotting
import matplotlib.pyplot as plt
```

4.1 Make a histogram for a chosen magnitude

Decide which magnitude is being used to determine the completeness limit. I'll be using the DECam 'i' magnitude here.

First, import the observed data, and drop any stars that are missing proper motion or parallax values.

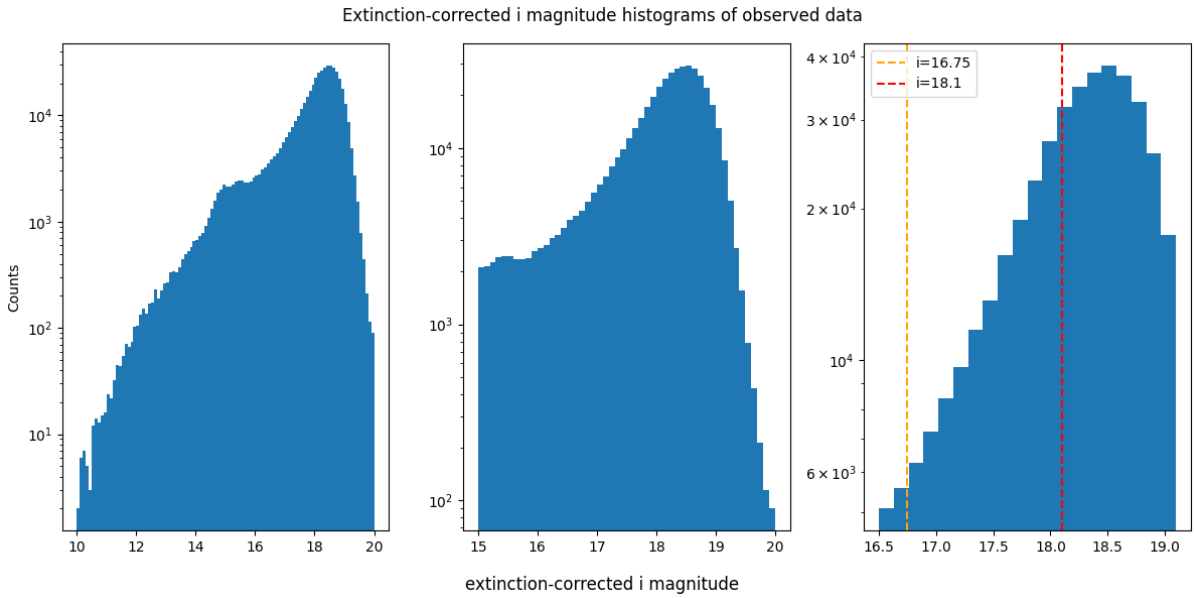
```
observed_df_raw = pd.read_csv('C:/Users/smick/Desktop/Research/data/
    NGC6569_gaia_withRVs.csv')
observed_df_raw = observed_df_raw.dropna(subset=['pmra', 'pmdec', '
    parallax'])
```

Make a new entry for brightened and deredenned magnitudes by subtracting away the extinction estimate for each star, making their apparent magnitudes as we expect to see them without dust or other line-of-sight obstructions.

```
observed_df_raw['imag_extcorr'] = observed_df_raw['imag'] -
    observed_df_raw['Ai']
```

Now We'll plot this extinction-corrected magnitude data to see where the counts of stars begins dropping off. Make a series of logarithmic histograms with increasingly narrow range to eyeball where around where the limit is.

```
full_hist = axs[0].hist(observed_df_raw['imag_extcorr'], range=(10, 20)
    , bins=100)
zoom1_hist = axs[1].hist(observed_df_raw['imag_extcorr'], range=(15,
    20), bins=100)
zoom2_hist = axs[2].hist(observed_df_raw['imag_extcorr'], range=(16.5,
    19.1), bins=100)
```



The slope looks constant between about 16.75 and 18.1, so that's where we want to calculate a best fit line.

4.2 Apply a best fit line

Create a new histogram for the range seen with a constant slope. Create new arrays for the counts and for the edges of each bin.

```
slope_hist = np.histogram(observed_df_raw['imag_extcorr'],
    range=(16.5,18.1), bins=100)
counts = slope_hist[0]
edges = slope_hist[1]
```

Take the logarithm of the counts

```
log_counts=np.log(counts)
```

Loop through the bin edges, and calculate the center point of each bin. These will be the points we use for the best fit line.

```
for i in range(len(log_counts)):
    center = (edges[i]+ edges[i+1]) / 2
    centers.append(center)
```

Calculate the coefficients for the best fit line

```
coefficients = np.polyfit(centers,log_counts,1)
```

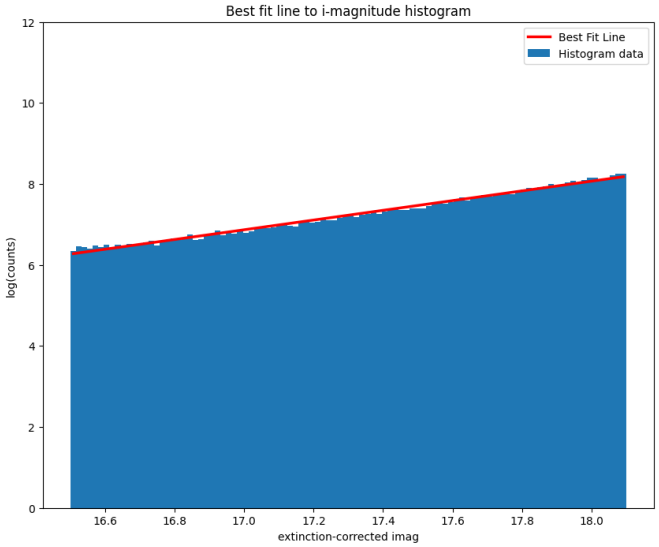
Evaluate the coefficients at the center values to produce the line.

```
fit_line = np.polyval(coefficients, centers)
```

Plot the histogram info as a bar chart, and then plot the best fit line.

```
plt.bar(centers, log_counts, width=(edges[1] - edges[0]), label='
    Histogram data')
plt.plot(centers, fit_line, label='Best Fit Line', color='red')
```

Plot a quick histogram of the range in which the best fit line is being applied to make sure everything is looking as we expect. If the best fit line looks good, now we want to extrapolate that line out to the magnitudes where we start losing stars.



We'll extrapolate the best fit line next, extending it out to fainter magnitudes, and compare it to the faint end of the histogram. First, pull the slope and intercept from the coefficient array.

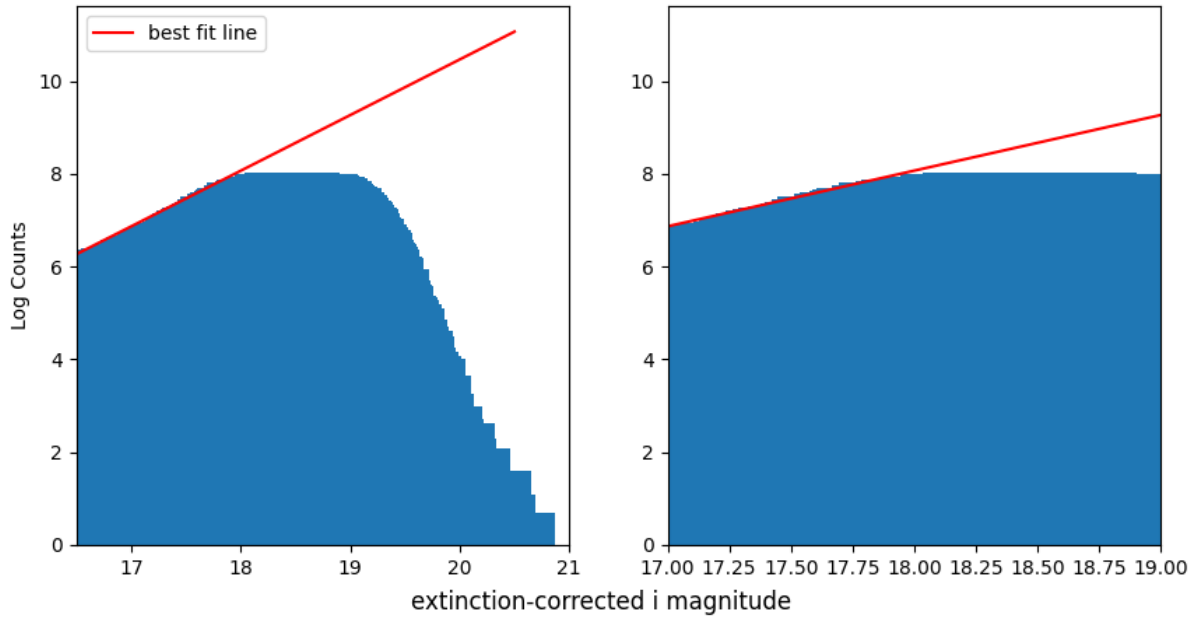
```
slope = coefficients[0]
intercept = coefficients[1]
```

Next, create a range of magnitudes for the desired range. It's important that the range and number of bins here matches that of the faint histogram.

```
extrapolated_magnitudes = np.linspace(16.5, 20.5, 400)
```

Then we'll use those magnitudes, the slope of the best fit line, and the intercept from the best fit line to create the line. Plot the line along with the histogram data as a bar chart.

```
extrapolated_log_counts = slope * extrapolated_magnitudes + intercept
```



4.3 Find where the star count falls off

Now that we can see where the line deviates, let's establish the exact magnitude where we start seeing a 10% difference in actual stars compared to expected stars.

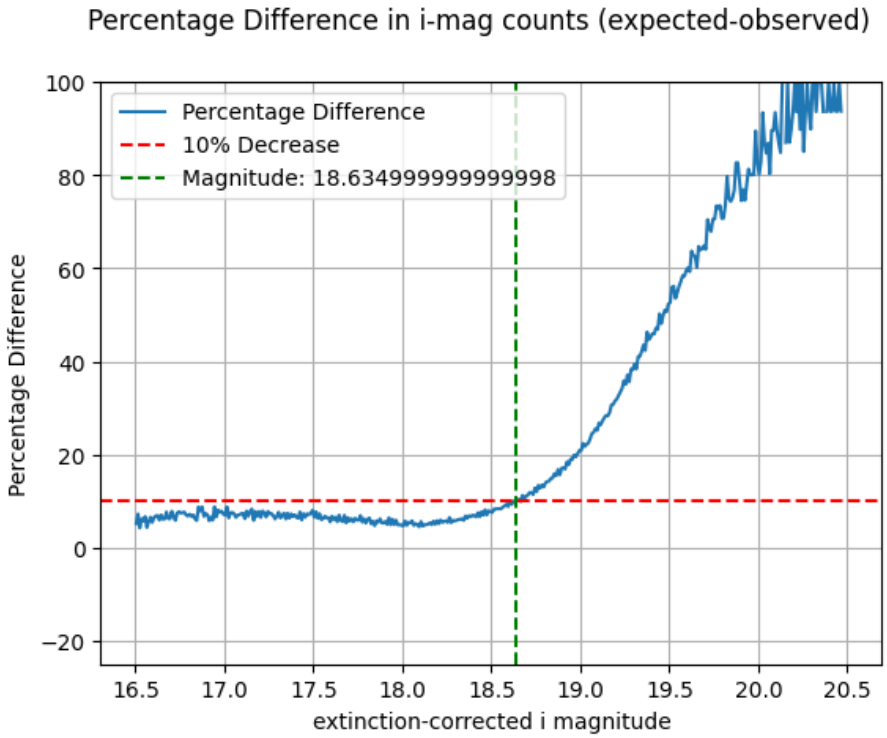
First, we'll calculate the percentage difference between the extrapolated logarithmic counts, and the counts from our faint histogram. If the two counts have mismatched lengths, this will throw an error.

```
percentage_difference = 100 * (extrapolated_log_counts -
    log_faint_counts) / extrapolated_log_counts
```

Now we need to loop through this array of percentage differences to find where we have a 10 % difference.

```
for i, perc_diff in enumerate(percentage_difference):\
    if perc_diff >= 10:
        magnitude_with_10_percent_decrease = faint_centers[i]
        break
print(f"The␣earliest␣magnitude␣with␣a␣10%␣decrease␣in␣stars:␣{
    magnitude_with_10_percent_decrease}")
```

We can then visualize this point by looking at a plot of the percentage difference between expected and observed.



Chapter 5

Data Processing

This notebook takes the observed stars, creates entries for the extinction-corrected magnitudes, then truncates them to the extinction-corrected completeness limit found in the previous section. We'll then calculate Galactic 'l' and 'b' coordinates for each star in the observed data.

Packages:

```
# General imports
import pandas as pd
import numpy as np
# Handling Gala .fits file
from astropy.io import fits
# To convert coordinates to Galactic l and b
from astropy.coordinates import SkyCoord
import astropy.units as u
```

5.1 Correcting for extinction

Begin by importing the observed stars, and the cluster simulation.

```
observed_df_raw = pd.read_csv('C:/Users/smick/Desktop/Research/data/
    NGC6569_gaia_withRVs.csv')
observed_df_raw = observed_df_raw.dropna(subset=['pmra', 'pmdec', '
    parallax'])
```

The cluster simulation will not be used in this notebook, but it's helpful to take it from a .fits to be used as a .csv for the rest of the notebook. We first have to open the file, convert to an array, convert that array to little-endian format, and then to a dataframe.

```
with fits.open('C:/Users/smick/Desktop/Research/data/NGC6569_gala.fits'
    ) as hdul:
     gala_data = hdul[1].data

gala_array = np.array(gala_data)
gala_array = gala_array.byteswap().view(gala_array.dtype.newbyteorder('
    <'))
gala_df = pd.DataFrame({name: gala_array[name] for name in gala_array.
    dtype.names})
```

Now, take the observed star's magnitudes and subtract away the estimate for extinction. Do this for any magnitude necessary.

```
observed_df_raw['gmag_extcorr'] = observed_df_raw['gmag'] -
    observed_df_raw['Ag']
```

5.2 Truncating to extinction-corrected completeness limit

Now we simply need to limit the datasets to stars brighter than our completeness limit. Check the numbers of stars in the observed dataset after truncation to get an idea of how many are lost after the cut.

```
observed_df_cut = observed_df_raw[(observed_df_raw['imag_extcorr'] > 9)
    & (observed_df_raw['imag_extcorr'] <= 18.63)]
```

5.3 Calculate Galactic 'l' and 'b' coordinates

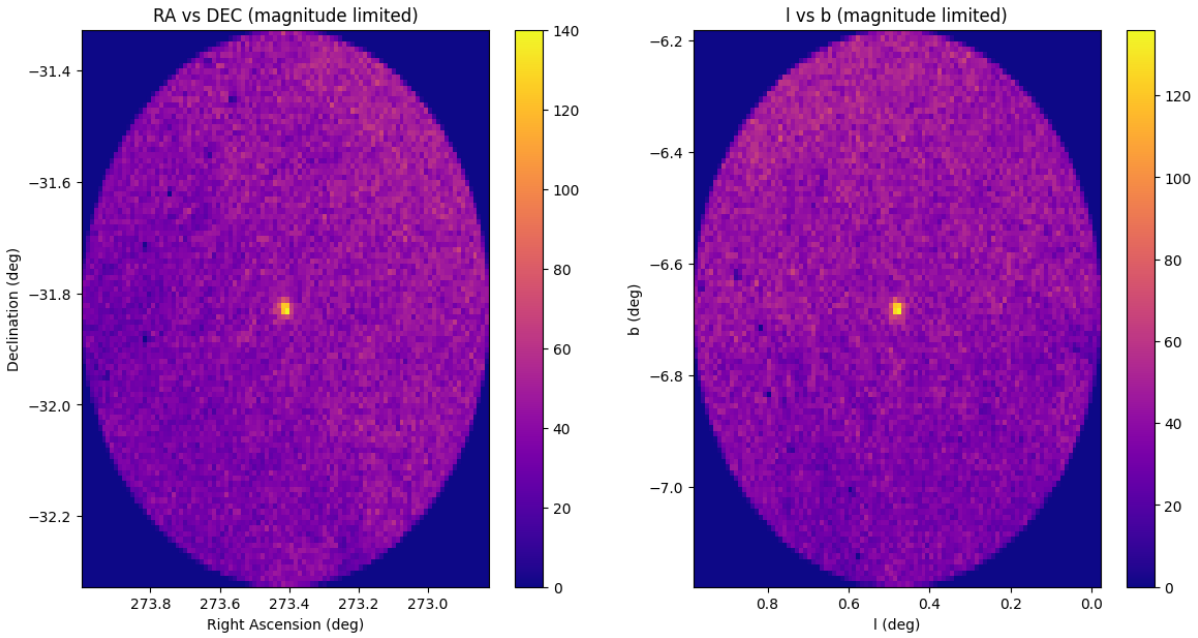
Lastly, we'll use astropy and the observed star's RA,DEC coordinates to calculate new Galactic 'l' and 'b' coordinates. Start by creating an SkyCoord object for the RA,DEC coordinates.

```
coords = SkyCoord(ra=observed_df_cut['radeg'].values,
                  dec=observed_df_cut['decdeg'].values,
                  unit=(u.degree,u.degree),
                  frame='icrs')
```

Take this object, and create new entries in the dataframe for the Galactic 'l' and 'b' coordinates. For clusters near the galactic center, particularly with ranges around 0, it's important to wrap the values at 180 degrees.

```
    observed_df_cut['l'] = coords.galactic.l.wrap_at(180 * u.degree).
        deg
    observed_df_cut['b'] = coords.galactic.b.deg
```

Do a quick side by side with the RA and DEC plot to make sure thngs look how we'd expect.



Finish by saving a copy of this dataframe, and the gala stars dataframe as .csv's.

```
observed_df_cut.to_csv('observed_stars_mag_limited.csv', index=False)
gala_df.to_csv('gala_stars.csv')
```

Chapter 6

Field Star Extrapolation

For this field of view, we can see a gradient in 'l' and 'b' space that is more dense at the cluster center, becoming less dense as 'b' decreases. We can take advantage of this gradient to simulate a field star population by looking at the density per row, and randomizing the positions of the stars within that row while retaining their proper motions.

Packages:

```
# General imports
import pandas as pd
import numpy as np
# To convert coordinates to Galactic l and b (for simulating field
    stars)
from astropy.coordinates import SkyCoord
import astropy.units as u
# Plotting
import matplotlib.pyplot as plt
# Math stuff
import math
from scipy.stats import norm
import shapely.geometry as geom
import shapely.ops as ops
from shapely import affinity
```

6.1 Mask out central stars

Start by defining the cluster center based on available literature, making sure the values are in units of degrees.

```
ra_center_hms = (18, 13, 38.9)
dec_center_dms = (-31, 49, 35)
# Convert to decimal degrees
ra_center = 15 * (ra_center_hms[0] + ra_center_hms[1] / 60 +
    ra_center_hms[2] / 3600)
dec_center = dec_center_dms[0] - dec_center_dms[1] / 60 -
    dec_center_dms[2] / 3600
```

Make a SkyCoord object for the cluster center next, and create new arrays for the values. Wrap if needed.

```
cluster_coords = SkyCoord(ra=ra_center,
                  dec=dec_center,
                  unit=(u.degree,u.degree),
                  frame='icrs')
```

```
center_l = cluster_coords.galactic.l.wrap_at(180 * u.degree).deg
center_b = cluster_coords.galactic.b.deg
```

Establish a bounding box around that cluster center that encompasses as much of the tidal radius as necessary. Define the half width, and the edges using the half width and cluster center.

```
square_half_width = 0.05

cluster_l_min = center_l - square_half_width
cluster_l_max = center_l + square_half_width
cluster_b_min = center_b - square_half_width
cluster_b_max = center_b + square_half_width
```

Now we'll use this box to create a mask that allows us to remove the stars at the dense cluster center.

```
outside_cluster_mask = ~(
    (observed_df_cut['l'] >= cluster_l_min) & (observed_df_cut['l'] <=
       cluster_l_max) &
    (observed_df_cut['b'] >= cluster_b_min) & (observed_df_cut['b'] <=
       cluster_b_max)
)
observed_l_outside = observed_df_cut['l'][outside_cluster_mask]
observed_b_outside = observed_df_cut['b'][outside_cluster_mask]
```

6.2 Creating rows in 'b' space

Now that we have a set of real field stars, we can use this to extrapolate from and create a simulated set of field stars. Importantly, we can use this to simulate what the field would look like in front of and behind the cluster center, using the information from the field right around it.

We'll use the minima and maxima of the observed data to create the number of rows needed to meet the desired resolution, 0.001 degrees per row in this case. Start by finding the mins and maxes for 'l' and 'b'.

```
b_min, b_max = np.min(observed_df_cut['b']), np.max(observed_df_cut['b'
   ])
l_min, l_max = np.min(observed_df_cut['l']), np.max(observed_df_cut['l'
   ])
```

Next, we'll calculate the radius of our field of view, using 'b' so we can figure out how many rows to bin by.

```
R = ((center_b - b_min) +      (b_max - center_b)) / 2
```

Now set the number of rows needed to achieve the desired resolution

```
N-rows = 100
```

Define the edges, midpoints, and limits in 'l'

```
b_edges = np.linspace(b_min, b_max, N_rows +1)
b_mids = 0.5 * (b_edges[:-1] + b_edges[1:])
full_lmin = l_min
full_lmax = l_max
```

6.3 Calculating row-wise density using shapely

Because we are working with rectangular rows in a circular field of view, we need to account for the overlap at the edges of each row, where it exceeds the FOV and no stars are found. At the same time, we need to exclude the area of the bounding box from the calculations.

We'll start by creating a shape the dimensions of the observed data's field of view, and a shape for the bounding box.

```
center = geom.Point(center_l, center_b)
circle_polygon = center.buffer(R, resolution=256)

cluster_box_polygon = geom.Polygon([
    (cluster_l_min, cluster_b_min),
    (cluster_l_min, cluster_b_max),
    (cluster_l_max, cluster_b_max),
    (cluster_l_max, cluster_b_min),
])
```

Now, for each row, we need to loop through and determine the area of that row contained both within the field of view, and outside of the cluster bounding box. We'll do this by using shapely's intersection function to find the points of overlap between the various shapes, and then the area.

At the same time, while we're going row-by-row, we'll use our masked dataset to determine how many stars are contained within that row. Lastly, we use the area and the counts to then determine the density of the field in that row.

```
for i in range(N_rows):
    b_lower = b_edges[i] # pull 'b' boundaries
    b_upper = b_edges[i + 1]

    # Define the four points of each row rectangle
    row_rect = geom.Polygon([
        (full_lmin, b_lower),
        (full_lmin, b_upper),
        (full_lmax, b_upper),
        (full_lmax, b_lower),
    ])

    # Find the intersection of row and circle:
    row_in_circle = row_rect.intersection(circle_polygon)

    # Calculate area of that intersection
    row_area_in_circle = row_in_circle.area

    # Repeat to calculate portion of this row inside the circle AND
       inside cluster bounding box. This is the area around cluster
       center that we don't want to count.
    row_in_circle_and_box = row_in_circle.intersection(
       cluster_box_polygon)
    box_overlap_area = row_in_circle_and_box.area

    # Subtract away the cluster bounding box area
    row_area_outside_box = row_area_in_circle - box_overlap_area
```

```
    row_areas[i] = row_area_outside_box # this is the area of the row
       that we will use for density calculation

    # Define a mask for stars within this row
    in_row = (
        (observed_b_outside >= b_lower) & (observed_b_outside < b_upper
           ) &
        ((observed_l_outside - center_l)**2 + (observed_b_outside -
           center_b)**2 <= R**2)
    )
    # Count
    row_counts[i] = np.sum(in_row)

# Use counts and areas to calculate density per row
density_per_row = row_counts / row_areas
```

With all of this row-wise information, the next step is to generate a number of random points within each row that meets number required for the density in that row. We'll first define a function that randomly samples a given number of points within a polygon.

```
def sample_points_in_polygon(polygon, n_points):
    rng = np.random.default_rng()
    minx, miny, maxx, maxy = polygon.bounds

    points = []
    # Keep generating until we have n_points inside 'polygon'
    while len(points) < n_points:
        # Generate random x and y coords within in bounding box
        x = rng.uniform(minx, maxx)
        y = rng.uniform(miny, maxy)
        p = geom.Point(x, y) # define point as pair of x and y coords
        if polygon.contains(p):
            points.append([x, y]) # add point if it falls within area
               of row

    return np.array(points)
```

Now we need to loop through each row again, calculating using row densities and the stars within that row to make copies of the stars, and then use our function to define new positions for those star copies.

```
for i in range(N_rows):

    # Define row boundaries in b
    b_lower = b_edges[i]
    b_upper = b_edges[i + 1]

    # Create the row rectangle
    row_rect = geom.Polygon([
        (l_min, b_lower),
        (l_min, b_upper),
        (l_max, b_upper),
        (l_max, b_lower),
    ])

    # Intersect with the circle polygon and calculate area
    row_in_circle = row_rect.intersection(circle_polygon)
    row_area_total = row_in_circle.area
```

```
    # Retrieve the row's star density
    row_density = density_per_row[i]

    # Set number of stars needed to populate row
    n_new = int(round(row_density * row_area_total))

    # Identify the stars outside the cluster bounding box in this row
    in_row_field_mask = (
        (observed_df_cut['b'] >= b_lower) & (observed_df_cut['b'] <
            b_upper)
        & ((observed_df_cut['l'] - center_l)**2 + (observed_df_cut['b']
            - center_b)**2 <= R**2)
        & outside_cluster_mask
    )
    row_field_stars = observed_df_cut[in_row_field_mask]

    # Sample star properties from row_field_stars with replacement
    rng = np.random.default_rng()
    chosen_indices = rng.integers(0, len(row_field_stars), size=n_new)
        # generate a list of random indices to be pulled
    new_star_props = row_field_stars.iloc[chosen_indices].copy() #
        create copies of the random stars

    # Generate n_new random positions throughout the entire row,
        including the cluster bounding box
    new_lb = sample_points_in_polygon(row_in_circle, n_new)

    # Overwrite l and b of the new stars
    new_star_props.loc[:, 'l'] = new_lb[:, 0]
    new_star_props.loc[:, 'b'] = new_lb[:, 1]

    # Collect this row's new stars
    simulated_row_list.append(new_star_props)
```

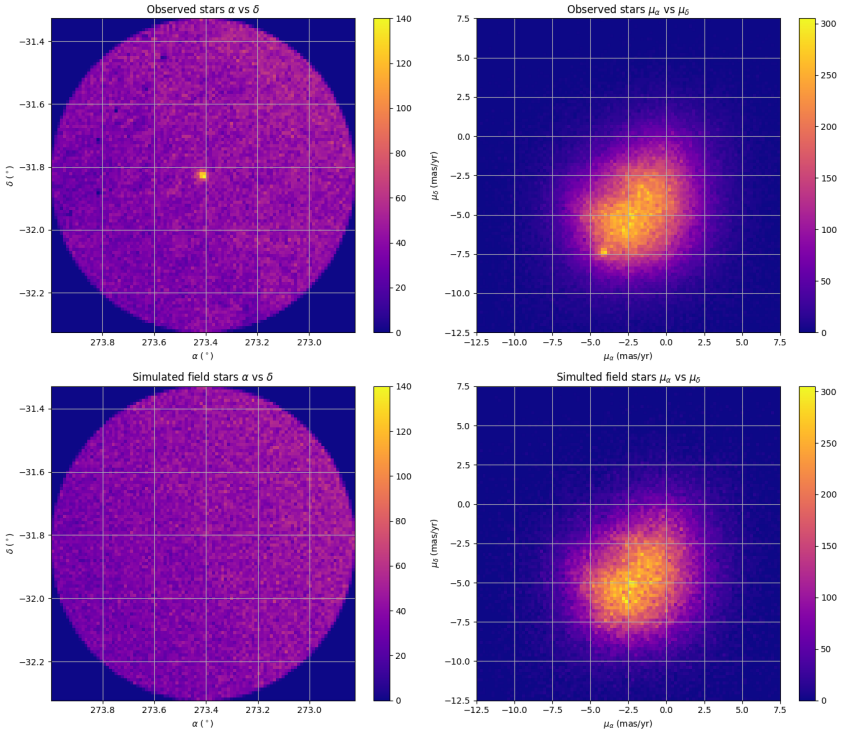
We can now take these new synthetic stars, concatenate all the rows into a dataframe, and calculate the RA and DEC coordinates.

```
simulated_df = pd.concat(simulated_row_list, ignore_index=True)

coords = SkyCoord(l=simulated_df['l'].values * u.degree,
                  b=simulated_df['b'].values * u.degree,
                  frame='galactic')

icrs = coords.icrs  # set ICRS coordinate frame
simulated_df['radeg'] = icrs.ra.degree
simulated_df['decdeg'] = icrs.dec.degree
```

Take a look at the new data, ensuring the simulated field has similar density in both position and proper motion space to that of the original data, minus the presence of the cluster.



Chapter 7

Membership Calculations

This notebook takes the observed and simulated datasets, and subtracts the field stars from the full set of observed stars to reveal any overdensities in the residual array. This residual array is then analyzed to measure the ratio of residual stars to the number of observed stars originally in the voxel to create the residual ratio. The residual ratio is then weighted against the cluster likehlihood map, to determine which overdensities line up with cluster predictions.

Packages:

```
# General imports
import pandas as pd
import numpy as np
# Plotting
import matplotlib.pyplot as plt
import matplotlib.cm as cm
import matplotlib.image as mpimg
import matplotlib.patches as patches
import matplotlib.gridspec as gridspec
import matplotlib.colors as mcolors
from matplotlib.patches import Circle
# Math stuff
import math
from scipy.stats import norm
from scipy.stats import gaussian_kde
from scipy.ndimage import gaussian_filter
```

7.1 Bin and count stars in 5D space

Begin by importing the three datasets we have generated so far: the magnitude-truncated observed stars, the simulated field stars, and the simulated cluster stars.

We'll first create bins for the position-space. We want to span the tidal radius with 10 bins, so read in the tidal radius, making sure it's in degrees. Divide the tidal radius by 10 to get the resolution needed.

```
tidal_radius_as = 589.7 # (Pallanca, et al., 2023)
tidal_radius_deg = 589.7 / 3600
radec_resolution = tidal_radius_deg / 10
```

For proper motions, we want the bin size to be twice as large as the average uncertainty. This ensures that a star's error in its measurements will not push it over into its

neighboring bin. Calculate the mean uncertainty in the PMRA, PMDEC and Parallax dimensions.

```
pmra_uncertainty = np.mean(observed_df_cut['pmra_error'])
pmdec_uncertainty = np.mean(observed_df_cut['pmdec_error'])
px_uncertainty = np.mean(observed_df_cut['parallax_error'])
```

We need to also set the limits of the binning range. Set the position limits to the minima and maxima of the truncated observed data. For the proper motion limits, determine the range of proper motion values based off the distribution seen in the selected field of view. In this step, also determine the position centers for use in plotting later on.

```
dec_min, dec_max = np.min(observed_df_cut['decdeg']), np.max(
    observed_df_cut['decdeg'])
dec_center = (dec_min + dec_max) / 2

ra_min, ra_max = np.min(observed_df_cut['radeg']),
     np.max(observed_df_cut['radeg'])
ra_center = (ra_min + ra_max) / 2

pmra_min, pmra_max = -12.5, 7.5
pmdec_min, pmdec_max = -15.0, 5.0
plx_min, plx_max = -5, 5
```

Now define the resolution for each bin using the values generated previously.

```
ra_resolution = radec_resolution
dec_resolution = radec_resolution
pmra_resolution = 2 * pmra_uncertainty
pmdec_resolution = 2 * pmdec_uncertainty
plx_resolution = 2 * px_uncertainty
```

Use the resolutions and limits to determine the number of bins needed. Do this for each dimension.

```
n_ra_bins = int((ra_max - ra_min) / ra_resolution) + 1
```

We'll now use the linspace command to take the minima, maxima, and number of bins to generate the bins themselves. Again, do this for each dimension.

```
ra_bins = np.linspace(ra_min, ra_max, n_ra_bins)
```

To count the stars, we'll now use the digitize function to assign indices that reflect which bin a star's values falls into. Because of how digitize works, we first need to create a function that defines which bins our out-of-range values fall into.

```
def digitize_and_adjust(values, bins, out_of_range_low_bin,
    out_of_range_high_bin):
    # Assign indices
    indices = np.digitize(values, bins, right=True)
    # Adjust indices for out-of-range values
    adjusted_indices = []
    for value, index in zip(values, indices):
        if value < bins[0]:
            adjusted_indices.append(out_of_range_low_bin)
        elif value > bins[-1]:
            adjusted_indices.append(out_of_range_high_bin)
        elif value == bins[0]:
            adjusted_indices.append(1)  # because digitize places only
                values equal to lower edge of the range in bin 0, we
                need to tell it bin 1 instead so the star gets counted
                properly
        else:
            adjusted_indices.append(index)
    return np.array(adjusted_indices, dtype=int)
```

Now define the out-of-range bins as the the last and second-to-last bins. Do this for each dimension.

```
ra_out_of_range_low_bin = len(ra_bins)
ra_out_of_range_high_bin = len(ra_bins) +1
```

Next, we'll use the custom digitize function, along with the number of bins, and the specified out-of-range bins to assign indices to each star's values. We'll do this for each dimension, and for each dataset.

```
observed_ra_indices = digitize_and_adjust(observed_df_cut['radeg'],
    ra_bins, ra_out_of_range_low_bin, ra_out_of_range_high_bin)
```

We'll now use the indices to search through each unique combinations of bins, what we call a "voxel", and count the number of stars that matches. Do this for each dataset

```
np.add.at(observed_voxel_counts, (observed_ra_indices,
    observed_dec_indices,
                                  observed_pmra_indices,
                                     observed_pmdec_indices,
                                  observed_plx_indices), 1)
```

We now have an array of voxel counts for each dataset that describes the density of stars in the 5-D spatio-kinematic space.

7.2 Create cluster and field likelihood maps

We'll now take our voxel densities, and create likelihood maps that describe the overall distribution of stars in both of the simulations. We expect to see a wide distribution in the field stars, but a highly-concentrated pack of voxels in the cluster data.

The field stars are simulated with the proper motions of real stars, and therefor have

the errors included by nature. The cluster model, on the other hand, will need to undergo a Gaussian smoothing that simulates a spread of errors in each dimension. Start by defining the bandwith to smooth by, in sigmas, for each dimension, and then apply the Gaussian filter to the cluster counts.

```
sigma = [1, 1, 1, 1, 0.1]
gala_smoothed_counts = gaussian_filter(gala_voxel_counts, sigma=sigma)
```

We'll then normalize these counts to create the cluster likelihood map.

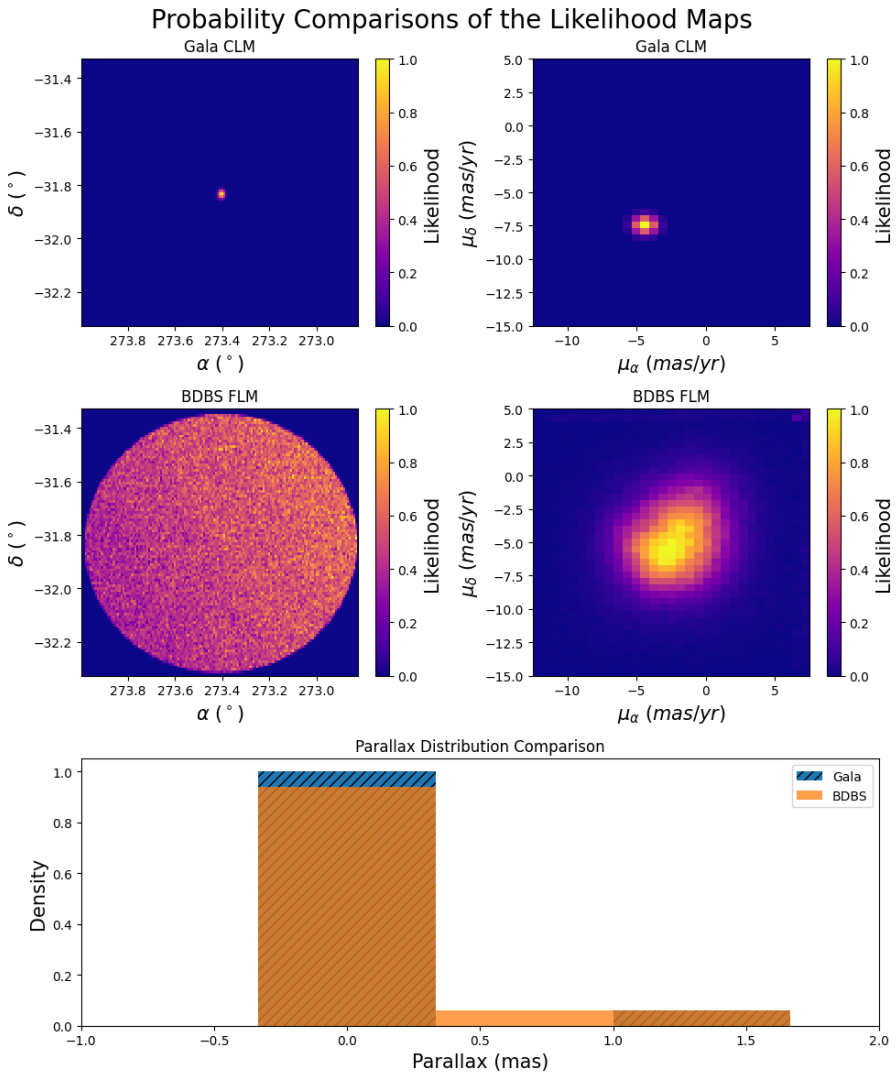
```
clm = gala_smoothed_counts / np.max(gala_smoothed_counts)
```

Repeat the normalization for the field likelihood map.

```
flm = simulated_voxel_counts/ np.max(simulated_voxel_counts)
```

We can then plot these likelihood maps to get an idea of the distributions we're looking at. Collapse down the dimensions not needed to display the maps in 2D. Renormalize them after summing, and plot.

```
clm_ra_dec =clm.sum(axis=(4,3,2))
clm_ra_dec /= clm_ra_dec.max()
```



7.3 Subtract and create voxel dataframe

We'll now carry out the subtraction, and assemble a new dataframe that contains all of the relevant voxel information. Somewhat surprisingly, the easy part is doing the subtraction of voxel counts in the 5D space.

```
residuals = observed_voxel_counts - simulated_voxel_counts
```

Now ensure we are working with floats, and assign values to NaNs in case any get generated.

```
residuals = residuals.astype(float)
residuals[np.isnan(residuals)] = -np.inf
```

Next we need to flatten to a 1D array, and sort into descending order. Then, unravel back into 5D voxels, yielding a tuple of arrays the same shape as the residuals. This will allow us to later match the 5D coordinates to each entry.

```
flattened_residuals = residuals.flatten()
sorted_indices = np.argsort(flattened_residuals)[::-1]
all_voxels = np.unravel_index(sorted_indices, residuals.shape)
```

Now we want to create a consolidated array of voxel indices so we can tell which voxel a given star belongs to. Do this for each dataset.

```
observed_voxel_indices = np.stack([observed_ra_indices,
   observed_dec_indices, observed_pmra_indices, observed_pmdec_indices,
    observed_plx_indices], axis=1)
```

Next we grab the object IDs from the relevant dataframes.

```
observed_n6569_oids = observed_df_cut['n6569_oid'].values
simulated_n6569_oids = simulated_df['n6569_oid'].values
```

And then find all the OIDs that exist in the observed, but not in the simulated.

```
residual_n6569_oids = np.setdiff1d(observed_n6569_oids,
   simulated_n6569_oids)
```

[[**Note:**
Because we are duplicating field stars to populate inside the cluster bounding box, we are also duplicating their field star OIDs. The subtraction process inside this box will always leave all the original stars, because by nature there are no OIDs from within the box in the simulated date. Is this potentially why we'll later see such a high density of 0 probability stars within the bounding box later on?
]]
Now mask to identify which rows in the observed data correspond to the residual OIDs.

```
residual_mask = np.isin(observed_n6569_oids, residual_n6569_oids)
residual_full_indices = np.where(residual_mask)[0]
```

Get the voxel indices for the residual stars

```
residual_voxel_indices = observed_voxel_indices[residual_full_indices]
```

Then convert each row of voxel indices into a typle that can be used as a dictionary key.

```
residual_voxel_tuples = [tuple(idx) for idx in residual_voxel_indices]
```

Now create a dataframe for the residual stars that contains the voxel index, the OID, and the star's indices from the observed data.

```
residual_stars_df = pd.DataFrame({
    'voxel_idx': residual_voxel_tuples,
    'n6569_oid': residual_n6569_oids,
    'full_index': residual_full_indices
})
```

Group the residual stars by their indices for easier sorting.

```
grouped_residual_stars = residual_stars_df.groupby('voxel_idx')
```

Now we need to define a helper function that looks at a bin, and returns the edges of that bin.

```
def bin_edges(bins, index):
    if index <= 0 or index >= len(bins):
        return (np.nan, np.nan)
    else:
        return (bins[index - 1], bins[index])
```

And then build the dictionaries for fast lookups within the voxel sorting loop. This step allows the sorting loop to quickly grab the star indices in a given voxel. Do this for both the observed and the simulated stars.

```
observed_dict = defaultdict(list)
for i, idx in enumerate(observed_voxel_indices):
    observed_dict[tuple(idx)].append(i)
```

Now we'll move into the loop that will go through each voxel in the flattened residual array, and return a dataframe that contains the relevant voxel information and OIDs for matching stars.

We'll skip any of the voxels that are empty, which we've assigned -infinity values to, and iterate over the residuals in descending order. Use *zip(*all_voxels)* and *flattened_residuals[sorted_indices]* to pair each voxel index with its corresponding residual value.

```
for voxel_idx, residual_value in zip(zip(*all_voxels),
   flattened_residuals[sorted_indices]):
    if residual_value == -np.inf:
        continue
```

Now use the helper function to convert the voxel indix into the actual coordinate ranges.

```
    ra_range = bin_edges(ra_bins, voxel_idx[0])
    dec_range = bin_edges(dec_bins, voxel_idx[1])
    pmra_range = bin_edges(pmra_bins, voxel_idx[2])
    pmdec_range = bin_edges(pmdec_bins, voxel_idx[3])
    plx_range = bin_edges(plx_bins, voxel_idx[4])
```

Turn the voxel index array into a typle so it can be used as a dictionary.

```
   voxel_tuple = tuple(voxel_idx)
```

Check if the voxel has any residual stars, skip if not.

```
    if voxel_tuple in grouped_residual_stars.groups:
        residual_stars = grouped_residual_stars.get_group(voxel_tuple)
        residual_star_count = len(residual_stars)
    else:
        residual_star_count = 0
    if residual_star_count == 0:
        continue
```

Find the row indices of all observed and simulated stars in this voxel using the dictionary.

```
    observed_matching_indices = observed_dict.get(voxel_tuple, [])
    simulated_matching_indices = simulated_dict.get(voxel_tuple, [])
```

Append all of the relevant information and return the dataframe.

```
    voxel_info_list.append({
        'RA Range': ra_range,
        'DEC Range': dec_range,
        'PMRA Range': pmra_range,
        'PMDEC Range': pmdec_range,
        'Parallax Range': plx_range,
        'Residual Count': residual_value,
        'CLM Density': clm[voxel_idx],
        'FLM Density': flm[voxel_idx],
        'Observed Star Count': len(observed_matching_indices),
        'Simulated Star Count': len(simulated_matching_indices),
        'Residual Star Indices': residual_stars['full_index'].values,
        'Residual Star IDs': residual_stars['n6569_oid'].tolist()
    })

voxel_df = pd.DataFrame(voxel_info_list)
```

7.4 Calculate weighted probabilities

Use the residual and observed star counts from the new voxel dataframe to calculate a per-voxel residual ratio. This tells us the probability of selecting a star that belongs to the residual dataset from a given voxel. Replace any NaNs with 0s.

```
voxel_df['Residual Ratio'] = voxel_df['Residual Count'] / voxel_df['
    Observed Star Count']
voxel_df['Residual Ratio'] = voxel_df['Residual Ratio'].fillna(0)
```

We now want to look at this value and compare it to the cluster and field likelihood maps. There are two ways of doing this, and we can compare the results later on.

7.4.1 CLM as a gatekeeper

The first way to go about this is to use the cluster likelihood map as a simple gatekeeper, that says "keep the residual value for this voxel if the cluster model predicts there to also be stars there."

This function creates a Gatekeeper Residual entry, which is just a copy of the residual ratio value anywhere the CLM density is greater than a certain threshold.

```
voxel_df['Gatekeeper Residual'] = np.where(
    voxel_df['CLM Density'] > 0.25,
    voxel_df['Residual Ratio'],
    0
)
```

7.4.2 Applying both the FLM and CLM

If we want to also apply the field likelihood map, we can use Bayesian inference to combine the probabilities.

$$P_w = \frac{R \times CLM}{R \times CLM + FLM \times (1 - CLM)}$$

For example, consider a voxel that has high densities in both the CLM and FLM. For a residual ratio of 1, a CLM density of 0.75, and a field density of 0.75, the weighted probability becomes

$$P_w = \frac{1 \times 0.75}{1 \times 0.75 + 0.75 \times (1 - 0.75)} = 0.8$$

Calculate this probability, and apply it to any stars where we have a residual ratio above 0.5. That covers any voxel where there were twice as many stars in the observed dataset as were in the simulated.

```
voxel_df['Gala Weighted Residual'] = np.where(
    voxel_df['Residual Ratio'] >= 0.50,
    res_bayesian_probability,
    voxel_df['Gala Weighted Residual']
)
```

7.4.3 Append probabilities to observed stars

Now that we have our different measures of probability, we can append those values to the real stars, along with any relevant voxel information.

First, pull OIDs and values from the observed stars

```
observed_stars_info = observed_df_cut.set_index('n6569_oid')[['radeg',
   'decdeg', 'pmra', 'pmdec', 'parallax', 'rmag_extcorr', 'gmag_extcorr
   ','zmag_extcorr', 'imag_extcorr']]
```

Create an empty list to store the entries.

```
observed_probs = []
```

Begin looping through each voxel, pulling the voxel index values.

```
for _, voxel_row in voxel_df.iterrows():
    voxel_idx = voxel_row[['RA Range', 'DEC Range', 'PMRA Range', '
       PMDEC Range', 'Parallax Range']].values
```

Specify the variables from the residual voxel dataframe to be included.

```
    residual_ratio = voxel_row['Residual␣Ratio']
    flm_density = voxel_row['FLM␣Density']
    gala_weighted_residual = voxel_row['Gala␣Weighted␣Residual']
    gatekeeper_residual = voxel_row['Gatekeeper␣Residual']
    clm_density = voxel_row['CLM␣Density']
    residual_matching_indices = voxel_row['Residual␣Star␣Indices']
```

Skip any empty voxels.

```
    if residual_matching_indices.size == 0:
        continue
```

Append information from the observed stars and the residual voxel dataframe to the new observed probabilities list, and convert to a dataframe.

```
    matching_observed_stars = observed_stars_info.iloc[
       residual_matching_indices]
    for oid, star_data in matching_observed_stars.iterrows():
        observed_probs.append({
            'n6569_oid': oid,
            'radeg': star_data['radeg'],
            'decdeg': star_data['decdeg'],
            'pmra': star_data['pmra'],
            'pmdec': star_data['pmdec'],
            'Parallax': star_data['parallax'],
            'gmag_extcorr': star_data['gmag_extcorr'],
            'rmag_extcorr': star_data['rmag_extcorr'],
            'imag_extcorr': star_data['imag_extcorr'],
            'zmag_extcorr': star_data['zmag_extcorr'],
            'Residual_Ratio': residual_ratio,
            'CLM␣Density': clm_density,
            'FLM␣Density': flm_density,
            'Gala␣Weighted␣Residual': gala_weighted_residual,
            'Gatekeeper␣Residual': gatekeeper_residual,
            'Voxel␣Index': voxel_idx
        })

# Convert the results to a DataFrame
observed_probs_df = pd.DataFrame(observed_probs)
```

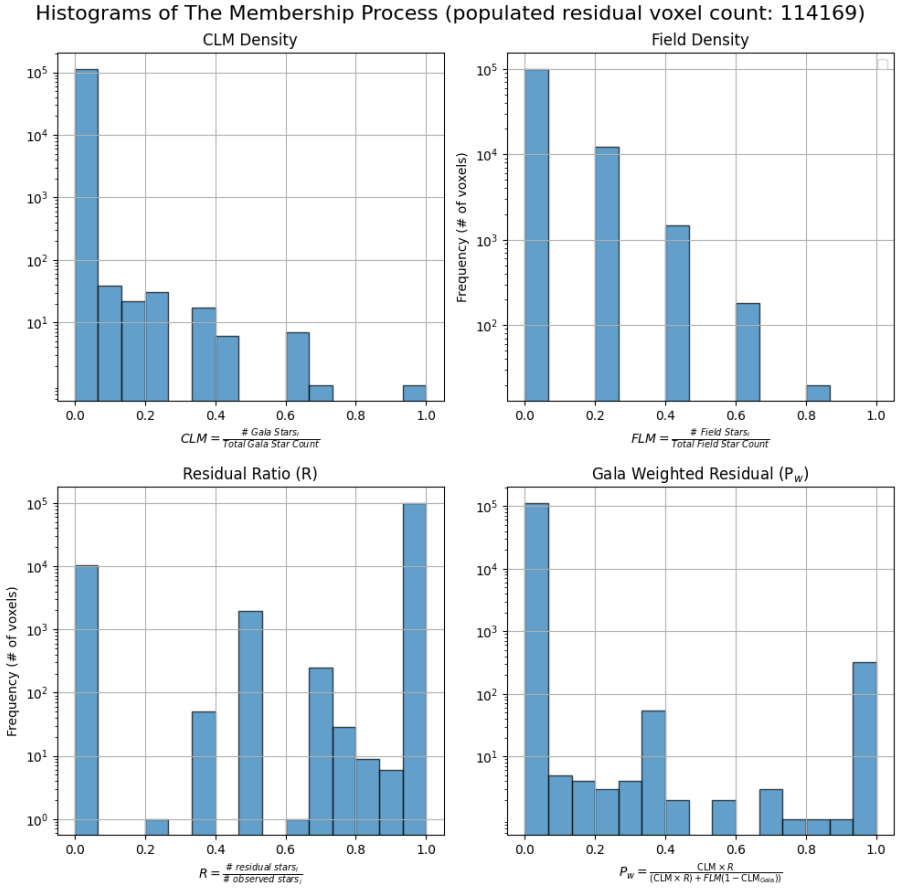
7.5 Plotting

Now that we have our probabilities assigned to real stars, we can plot the stars in the various spaces to see how the memberships have been assigned.

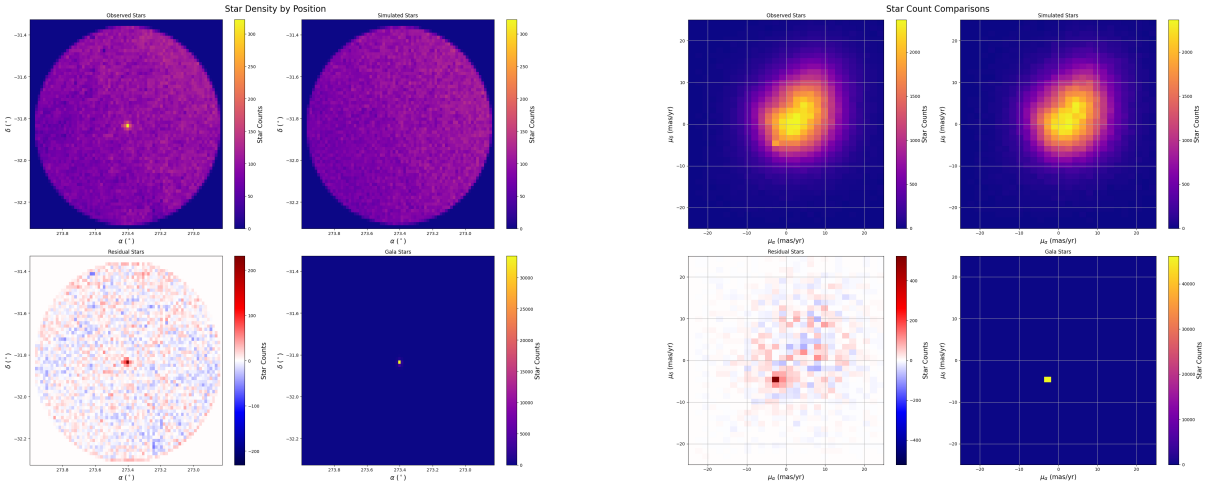
Separate the stars into groups of low, med-high, and high probabilities for visualization. Do this for both types of probabilities for comparison.

```
low_probs = observed_probs_df[observed_probs_df['Gala␣Weighted␣Residual
    '] < 0.5]
med_and_high_probs = observed_probs_df[observed_probs_df['Gala␣Weighted
    ␣Residual'] >= 0.5]
high_probs = observed_probs_df[observed_probs_df['Gala␣Weighted␣
    Residual'] >= 0.85]
```

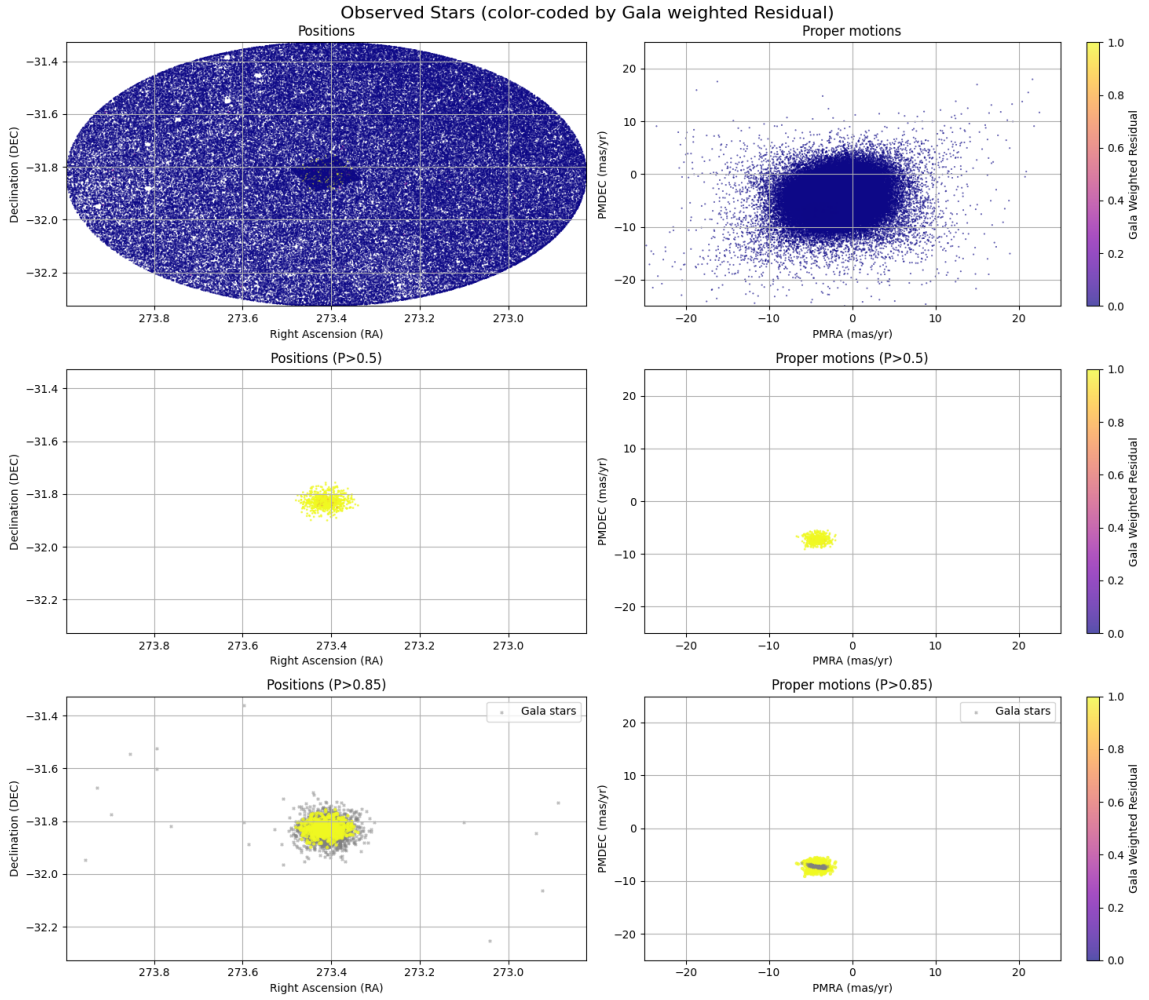
First, we want to plot the distributions of the CLM, FLM, Residual Ratio, and Gala Weighted Residual. We should expect a relatively Gaussian spread in the residual ratio, though there will almost always be a vast majority of ratio 1 voxels. This is due to the fineness in the binning, and can be reduced by making the bin sizes coarser to reduce the number of single star voxels. Too coarse,though, and we begin losing any meaning in the results because all the stars will live in the same few voxels.



We know the Bayesian did its job if we see the number of 1 values reduced by orders of magnitude like above. This process can be repeated, swapping out the Gala Weighted Residual with the Gatekeeper Residual in the lower right panel.
It's helpful to also get a position and motion representation of the simulated, observed, residual and cluster prediction datasets.



Next, we can take a look at the stars in position and motion space, and color-code them by their probabilities. Collapse the arrays down in the same way as we did previously, and make a series of position and motion plots that filter by probabilities.Overlay the cluster model stars to see where the CLM will be nonzero. The overdensity of 0 probabilities inside the cluster bounding box can be seen in the top left.



We can also view the high probability stars in a color-magnitude diagram, and com-

pare them against an isochrone generated from the cluster parameters. We'll need the distance modulus here to calculate the apparent magnitudes. Calculate the relevant colors, too.

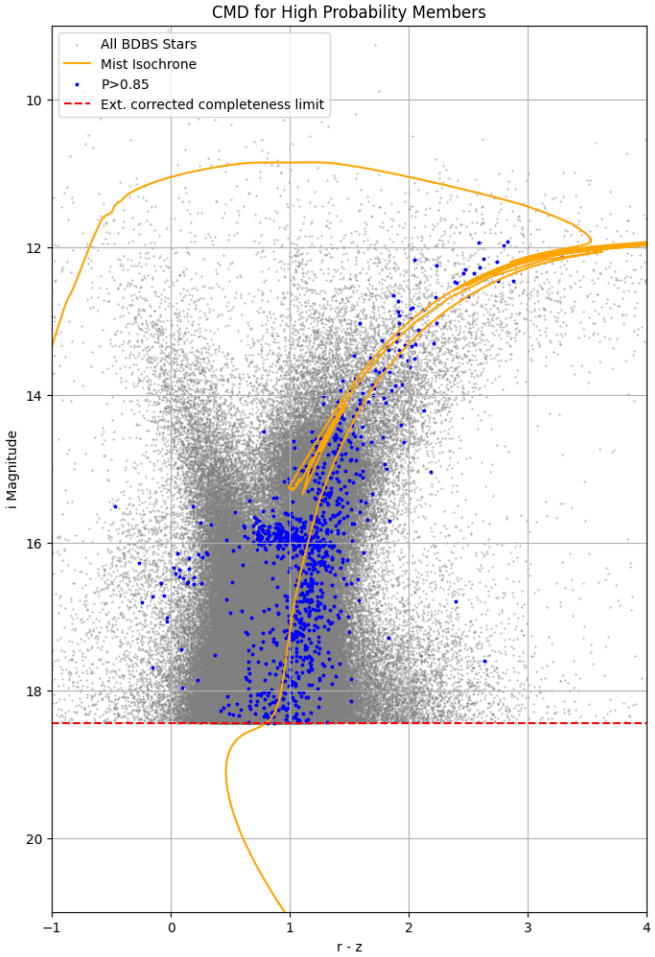
```
iso_mu = mist_iso['SDSS_u'] + distmod
iso_mg = mist_iso['SDSS_g'] + distmod
iso_u_g = mist_iso['SDSS_u'] - mist_iso['SDSS_g']
```

Include the completeness limit as a horizontal line to show where we're cutting the data off. The stars with high probabilities should follow closely to the isochrone if they are indeed members of the cluster.

In the case shown here, NGC 6569 is expected to have two distinct age populations, and we might be seeing the presence of the horizontal red giant branch of the secondary population.

Repeat the plotting process for the Gatekeeper residual to inspect how the two measures of probability compare to one another.

Export the results to a new .csv so we can take a look at how the memberships get assigned to stars from other datasets.



Chapter 8

Comparisons

We can now use the probabilities that have been assigned to the observed stars, and use their object IDs (oID) to match them with stars from other datasets.

8.1 Append membership info to new stars

Import the probabilities that we generated in the last notebook, the set of cluster stars used to create the CLM, and the datasets we want to compare with. Make sure the column naming is what we'd expect.

Merge the observed probability dataframe with the new datasets, using the oID to match on.

```
Johnson_crossmatches = Johnson_matches.merge(observed_probs_df[['
    n6569_oid', 'Gatekeeper␣Residual']], on='n6569_oid', how='left')
```

In case there are any stars that aren't assigned a membership probability, replace any NaNs with 0s.

```
Johnson_crossmatches['Gatekeeper␣Residual'] = Johnson_crossmatches['
    Gatekeeper␣Residual'].fillna(0)
```

8.2 Create markers for stars and plot

We need to understand why some stars are assigned probabilities and others aren't, so we'll create a series of markers that identifies if they are spatially or kinematically close to the cluster, both, or neither.

Begin by defining the cluster center in position and motion space, as well as a threshold within which we consider to be "close to the cluster". Define the core and tidal radius.

```
ra_center = 15 * (ra_center_hms[0] + ra_center_hms[1] / 60 +
    ra_center_hms[2] / 3600)
dec_center = dec_center_dms[0] - dec_center_dms[1] / 60 -
    dec_center_dms[2] / 3600
pmra_center = -4.125 # (PM data from Vasiliev & Baumgardt, 2021)
pmdec_center = -7.259
pm_threshold = 0.5
core_radius_deg = 19.9 / 3600
tidal_radius_deg = 589.7 / 3600
```

Next we'll define a function that takes the position and motion parameters, and checks them against the desired spatial and kinematic ranges.

```
def assign_membership_markers(
    df,
    ra_col,
    dec_col,
    pmra_col,
    pmdec_col,
    ra_center,
    dec_center,
    pmra_center,
    pmdec_center,
    pm_threshold,
    tidal_radius
):
```

Calculate a radial distance from the cluster center.

```
    d_ra = (df[ra_col] - ra_center) * np.cos(np.deg2rad(dec_center))
    d_dec = df[dec_col] - dec_center
    radius = np.sqrt(d_ra**2 + d_dec**2)
```

Create a mask for stars within the chosen radius.

```
    spatial_flag=(radius <= tidal_radius_deg)
```

Create another mask for stars within the proper motion range.

```
    pm_flag = (
        (df[pmra_col] >= pmra_center - pm_threshold) &
        (df[pmra_col] <= pmra_center + pm_threshold) &
        (df[pmdec_col] >= pmdec_center - pm_threshold) &
        (df[pmdec_col] <= pmdec_center + pm_threshold)
    )
```

Set the conditions as inside one or the other, inside both, and inside neither.

```
    conditions = [
        (spatial_flag & ~pm_flag),  # Spatial only
        (~spatial_flag & pm_flag),  # PM only
        (spatial_flag & pm_flag),   # Both
        (~spatial_flag & ~pm_flag)  # Neither
    ]
```

Assign labels for the various flags.

```
    markers = ['Only␣in␣tidal␣radius', 'Only␣in␣PM␣range', 'In␣both', '
        In␣neither']
```

Lastly, assign the flags with labels to the stars and return the results.

```
    df['membership_marker'] = np.select(conditions, markers, default='
        In␣neither')

    return df
```

Use the function to assign the markers to each dataset as needed.

```
Johnson_crossmatches = assign_membership_markers(
    df=Johnson_crossmatches,
    ra_col='radeg',
    dec_col='decdeg',
    pmra_col='pmra',
    pmdec_col='pmdec',
    ra_center=ra_center,
    dec_center=dec_center,
    pmra_center=pmra_center,
    pmdec_center=pmdec_center,
    pm_threshold=pm_threshold,
    tidal_radius=tidal_radius_deg
)
```

For plotting, we'll create a dictionary for the labels and symbols we want to use.

```
markers_dict = {
    'Only in tidal radius': 'o',    # circle
    'Only in PM range': '^',        # triangle
    'In both': '*',                 # star
    'In neither': 's'               # square
}
```

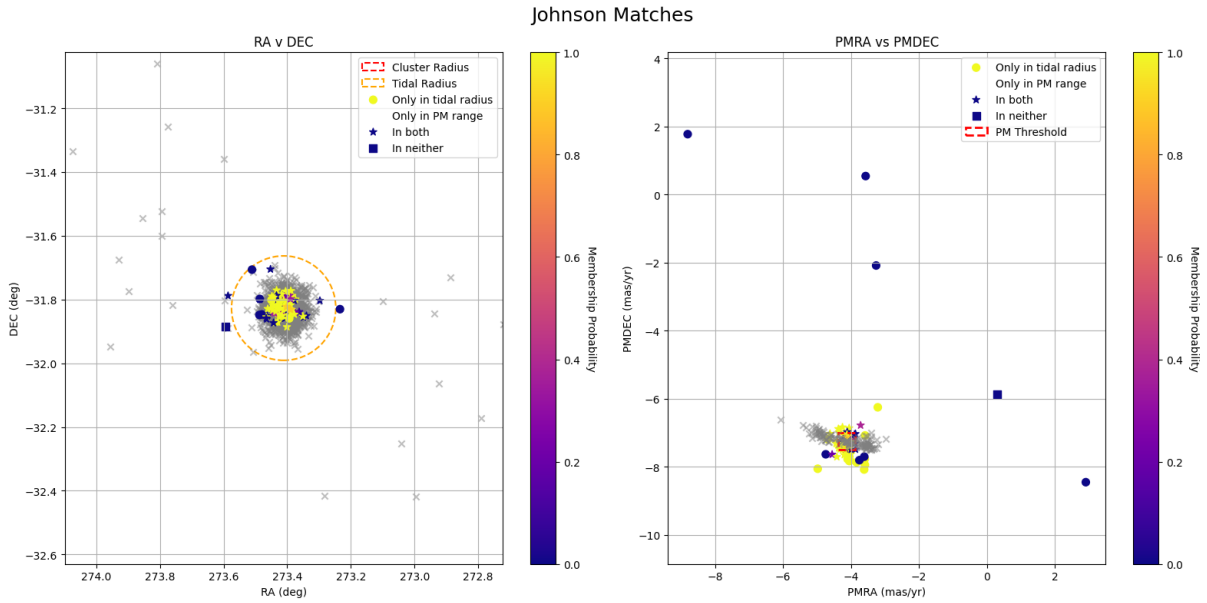
We want to add circles to the position plots for the core and tidal radii, and a rectangle to the motion plots to indicate the threshold used for sorting.

```
core_radius = Circle(
    (ra_center, dec_center), core_radius_deg,
    color='red', fill=False, linestyle='--', linewidth=1.5,
    label='Cluster Radius'
)
box = Rectangle(
    (lower_left_x, lower_left_y), box_size, box_size,
    fill=False,
    edgecolor='red',
    linewidth=2,
    linestyle='--',
    label='PM Threshold'
)
```

And we'll need to assign the categories to the stars,then plot.

```
for cat, marker in markers_dict.items():
    # Filter the subset
    subset = Johnson_crossmatches[Johnson_crossmatches['
        membership_marker'] == cat]
    sc1 = ax1.scatter(
        subset['radeg'],
        subset['decdeg'],
        c=subset['Gatekeeper Residual'],  # or your membership
            probability column
        cmap=cmap,
        norm=norm,
        marker=marker,
        s=50,
        alpha=1,
        label=cat
    )
```

Plot the cluster model stars in to show where the probabilities align with the CLM.



In this view, we can take a look at which stars are being assigned membership probabilities relative to their proximity to the cluster in position and motion space. We can also see where memberships are rejected due to the absence of cluster model stars in that area.

Chapter 9

Conclusions (coming soon)

9.1 Expected results

9.2 Known issues and areas to improve

9.2.1 Overdensity within cluster bounding box

9.2.2 Using velocity dispersion as a motion threshold

9.2.3 Using local significance as a weight