Homework 5

References

- Lectures 17-20 (inclusive).

Instructions

- Type your name and email in the "Student details" section below.
- Develop the code and generate the figures you need to solve the problems using this notebook.
- For the answers that require a mathematical proof or derivation you should type them using latex. If you have never written latex before and you find it exceedingly difficult, we will likely accept handwritten solutions.
- The total homework points are 100. Please note that the problems are not weighed equally.

In [59]:

```
import numpy as np
np.set_printoptions(precision=3)
import matplotlib.pyplot as plt
%matplotlib inline
import seaborn as sns
sns.set(rc={"figure.dpi":100, "savefig.dpi":300})
sns.set_context("notebook")
sns.set_style("ticks")

import scipy
import scipy.stats as st
import urllib.request
import os

def download(
    url : str,
    local_filename : str = None
):
    """Download a file from a url.

    Arguments
    url            -- The url we want to download.
    local_filename -- The filemame to write on. If not
                      specified
    """
    if local_filename is None:
        local_filename = os.path.basename(url)
    urllib.request.urlretrieve(url, local_filename)


import math
```

Student details

- **First Name:** Ben
- **Last Name:** McAteer
- **Email:** bmcateer@purdue.edu

Problem 1 - Clustering Uber Pickup Data

In this problem you will analyze Uber pickup data collected during April 2014 around New York City. The complete data are freely on Kaggle. The data consist of a timestamp (which we are going to ignore), the latitude and longitude of the Uber pickup, and a base code (which we are also ignoring). The data file we are going to use is uber-raw-data-apr14.csv. As usual, you have to make it visible to this Jupyter notebook. On Google Colab, just run this:

In [60]:
```
url = "https://github.com/PredictiveScienceLab/data-analytics-se/raw/master/lecturebook
download(url)
```

And you can load it using pandas:

In [61]:
```
import pandas as pd
p1_data = pd.read_csv('uber-raw-data-apr14.csv')
```

Here is a text view:

In [62]:
```
p1_data
```

Out[62]:

	Date/Time	Lat	Lon	Base
0	4/1/2014 0:11:00	40.7690	-73.9549	B02512
1	4/1/2014 0:17:00	40.7267	-74.0345	B02512
2	4/1/2014 0:21:00	40.7316	-73.9873	B02512
3	4/1/2014 0:28:00	40.7588	-73.9776	B02512
4	4/1/2014 0:33:00	40.7594	-73.9722	B02512
...	...	...	...	...
564511	4/30/2014 23:22:00	40.7640	-73.9744	B02764
564512	4/30/2014 23:26:00	40.7629	-73.9672	B02764
564513	4/30/2014 23:31:00	40.7443	-73.9889	B02764
564514	4/30/2014 23:32:00	40.6756	-73.9405	B02764
564515	4/30/2014 23:48:00	40.6880	-73.9608	B02764

564516 rows × 4 columns

As you see, there where about half a million Uber pickups during April 2014... Let's extract the lattitude and longitude data only (this is needed for passing them to scikit-learn algorithms). Here is how you can do this in pandas:

In [63]:
```
# Just use the column names as indices.
# The two brackets are required because you are actually
# passing a list of columns
loc_data = p1_data[['Lon', 'Lat']]
loc_data
```

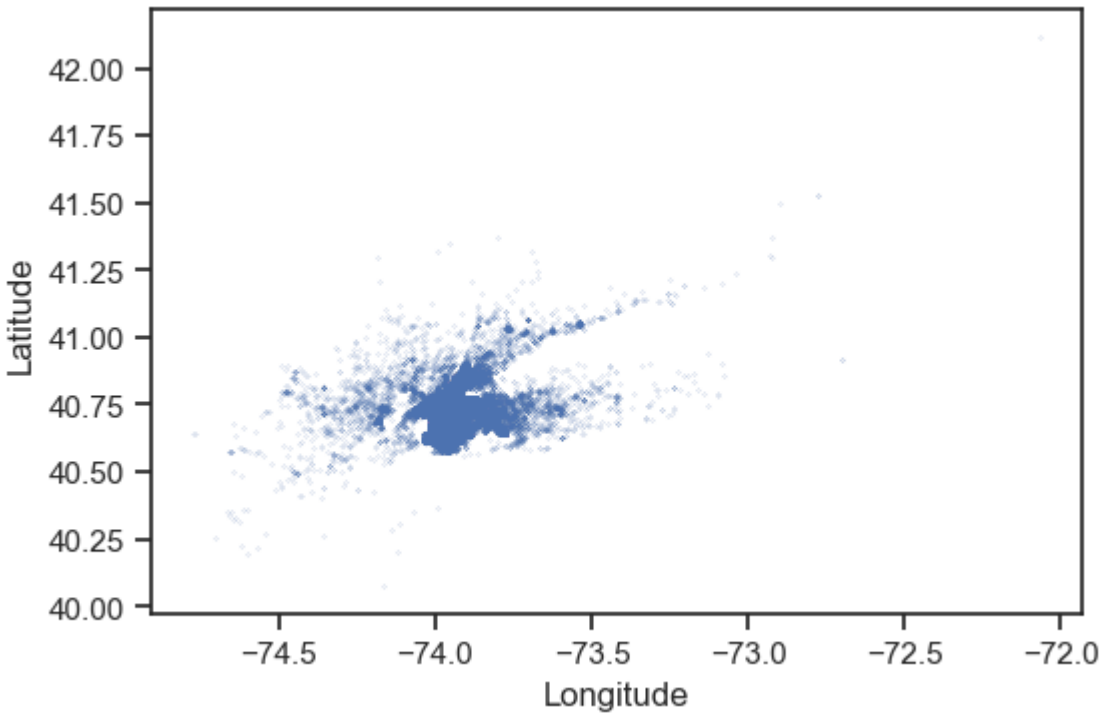
Out[63]:

	Lon	Lat
0	-73.9549	40.7690
1	-74.0345	40.7267
2	-73.9873	40.7316
3	-73.9776	40.7588
4	-73.9722	40.7594
...	...	...
564511	-73.9744	40.7640
564512	-73.9672	40.7629
564513	-73.9889	40.7443
564514	-73.9405	40.6756
564515	-73.9608	40.6880

564516 rows × 2 columns

Let's also visualize these points:

In [64]:
```
fig, ax = plt.subplots()
ax.scatter(loc_data.Lon, loc_data.Lat, s=0.01)
# ``s=0.01`` specifies the size. I am using a small size because
# these are too many points to visualize
ax.set_xlabel('Longitude')
ax.set_ylabel('Latitude');
```



This is nice, but it would be even nicer if we had a map of New York City on the background. We can make such a map on www.openstreetmap.org. We just need to have a box of longitude's and latitudes that overlaps with our data. Here is how to get such a *bounding box*:

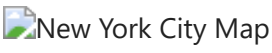
In [65]:
```
box = ((loc_data.Lon.min(), loc_data.Lon.max(),
        loc_data.Lat.min(), loc_data.Lat.max()))
box
```

Out[65]:
```
(-74.7733, -72.0666, 40.0729, 42.1166)
```

I have already extracted this picture for you and you can find it here. As always, it needs to be visible from the Jupyter notebook. On Google Colab run:

In [66]:
```
url = "https://github.com/PredictiveScienceLab/data-analytics-se/raw/master/lecturebook
download(url)
```

If you have it at the right place, you should be able to see the image here:



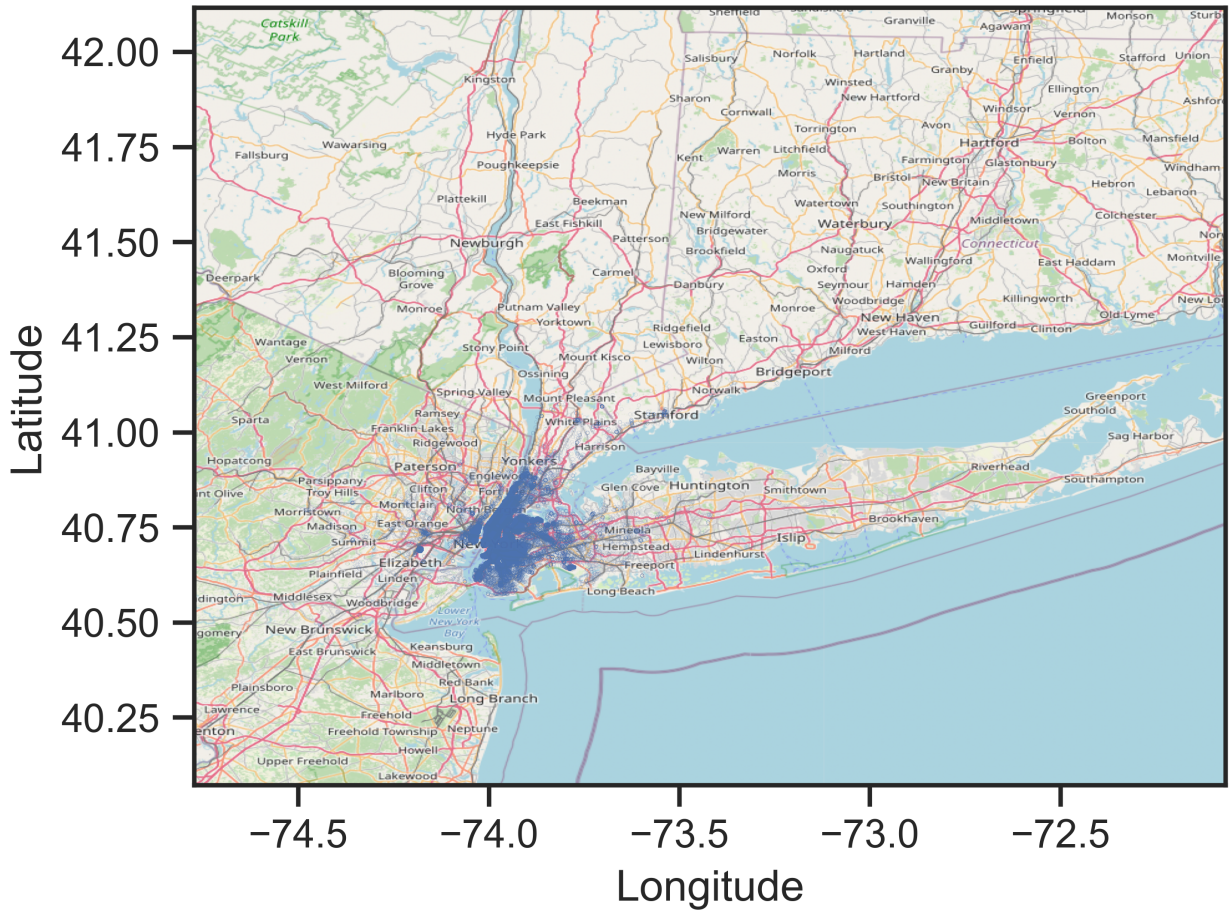
Now let's load the image as a matrix:

In [67]:
```
ny_map = plt.imread('ny_map.png')
```

And we can visualize it with `plt.imshow` and draw the Uber pickups on top of it. Here is how:

In [68]:
```
fig, ax = plt.subplots(dpi=600)
ax.scatter(
    loc_data.Lon,
    loc_data.Lat,
    zorder=1,
```

```
    alpha= 0.5,
    c='b',
    s=0.001
)
ax.set_xlim(box[0],box[1])
ax.set_ylim(box[2],box[3])
ax.imshow(
    ny_map,
    zorder=0,
    extent=box,
    aspect= 'equal'
)
ax.set_xlabel('Longitude')
ax.set_ylabel('Latitude');
```



Because we have over half a million data points, machine learning algorithms may be a bit slow. So, as you develop your code use use only 50K observations. Once you have a stable version of your code, modify the following code segment to make use of the entire dataset.

In [69]:

```
# While your are developing your code use this:
# p1_train_data = loc_data[:100000]
# When you have a stable code, use this:
p1_train_data = loc_data
```

Part A - Splitting New York City into Subregions

Suppose that you are assigned the task of splitting New York City into operating subregions with pretty much equal demand. When a pickup is requested in each subregion only the drivers in that region are called. Note that this can quickly become a very difficult problem very quickly. We are not looking for the best possible answer here. This would require posing and solving a suitable optimization problem. We are looking for a data-informed solution that is compatible with common sense.

Do (at least) the following:

- Use Kmeans clustering on the pickup data with different number of clusters;
- Visualize the labels of the clusters on the map using different colors (see the hands-on activities);
- Visualize the centers of the discovered Kmeans clusters (in red color);
- Use your common sense, e.g., make sure that you have enough clusters so that no region crosses the water (even if it is possible the drivers may have to pay tolls to cross). If it is impossible to get perfect results simply by Kmeans, feel free to ignore a small number of outliers as they could be handled manually;
- Use MiniBatchKMeans which is an much faster version of Kmeans suitable for large datasets (>10K observations);

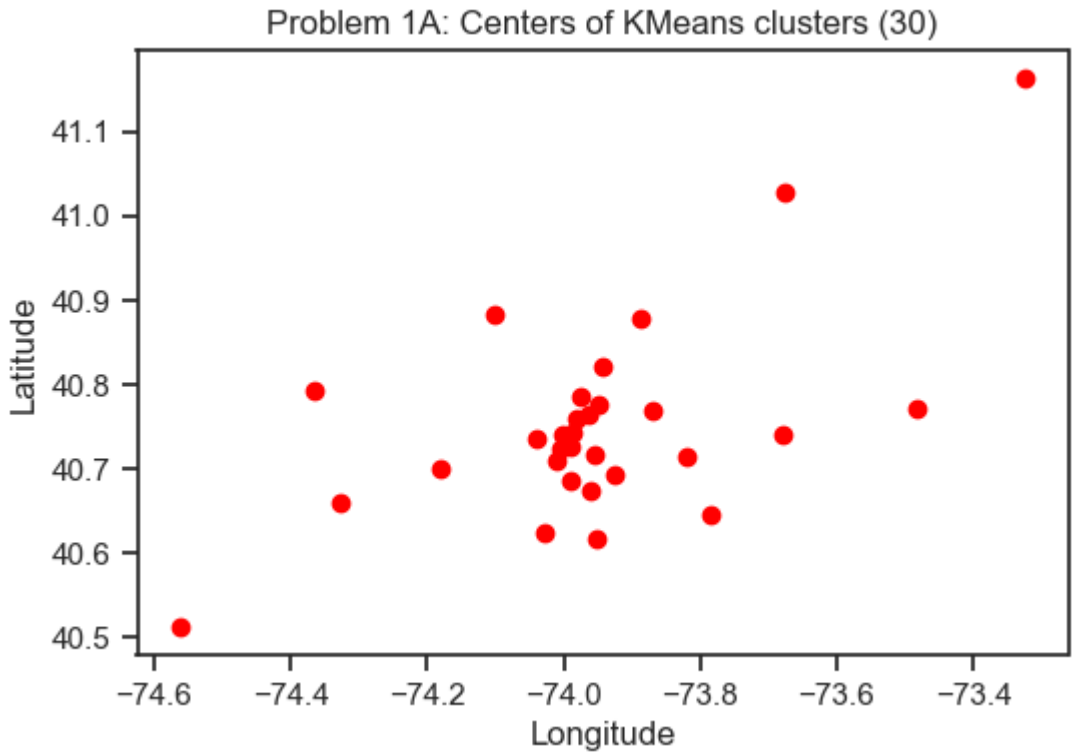
Answer with as many text blocks and code blocks as you like right below.

In [70]:

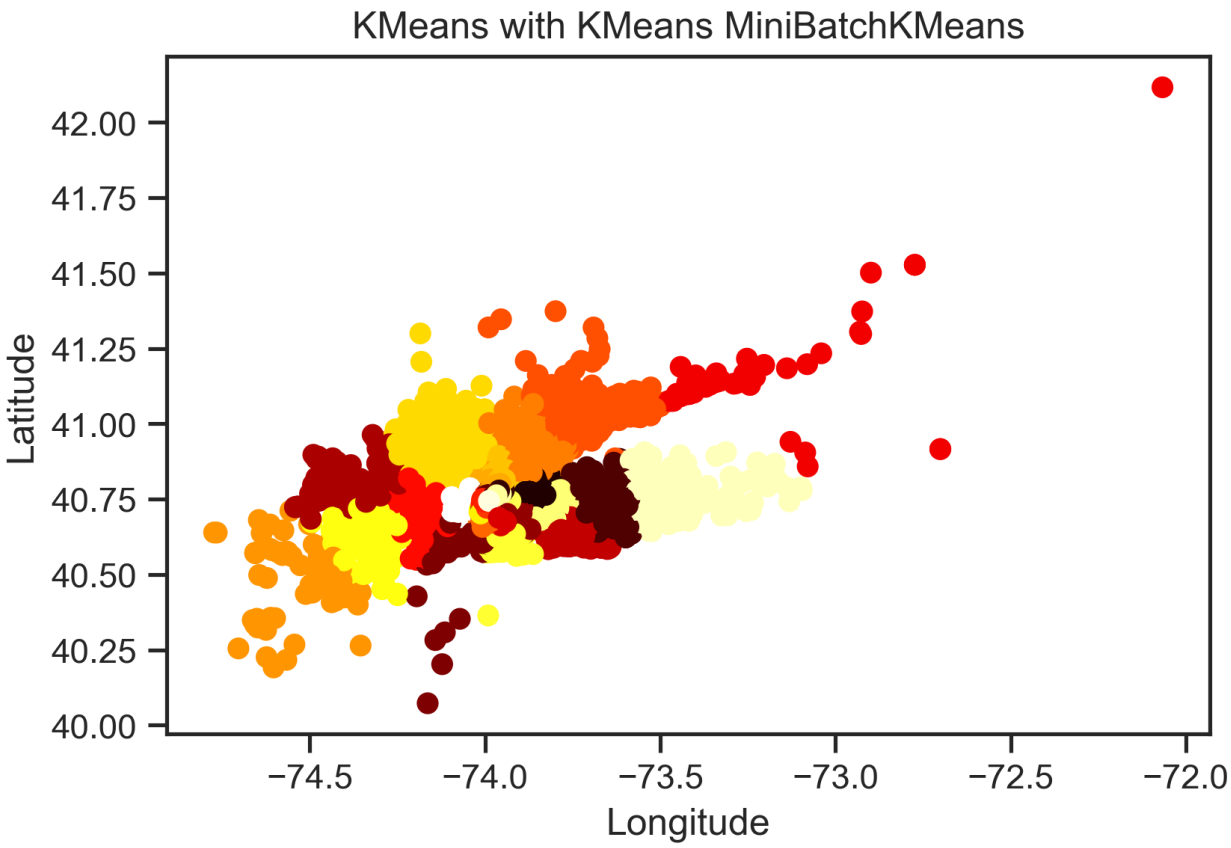
```
##part 1 A
#from sklearn.cluster import KMeans
from sklearn.cluster import MiniBatchKMeans

#model = KMeans(n_clusters=30).fit(p1_train_data)
#clustercenters = model.cluster_centers_

model2 = MiniBatchKMeans(n_clusters = 30, batch_size = 3072).fit(p1_train_data)
clustercenters2 = model2.cluster_centers_

plt.figure()
plt.scatter(clustercenters2[:,0],clustercenters2[:,1], color = 'red')
plt.xlabel('Longitude');
plt.ylabel('Latitude');
plt.title('Problem 1A: Centers of KMeans clusters (30)');


plt.figure()
labels = model2.predict(p1_train_data)
fig, ax = plt.subplots(dpi = 300)
plt.scatter(p1_train_data.Lon, p1_train_data.Lat, c=labels, cmap = 'hot')
ax.set_xlabel('Longitude');
ax.set_ylabel('Latitude');
plt.title('KMeans with KMeans MiniBatchKMeans');

fig, ax = plt.subplots(dpi=600)
ax.scatter(
    p1_train_data.Lon,
    p1_train_data.Lat,
    c=labels,
    cmap = 'hot',
```

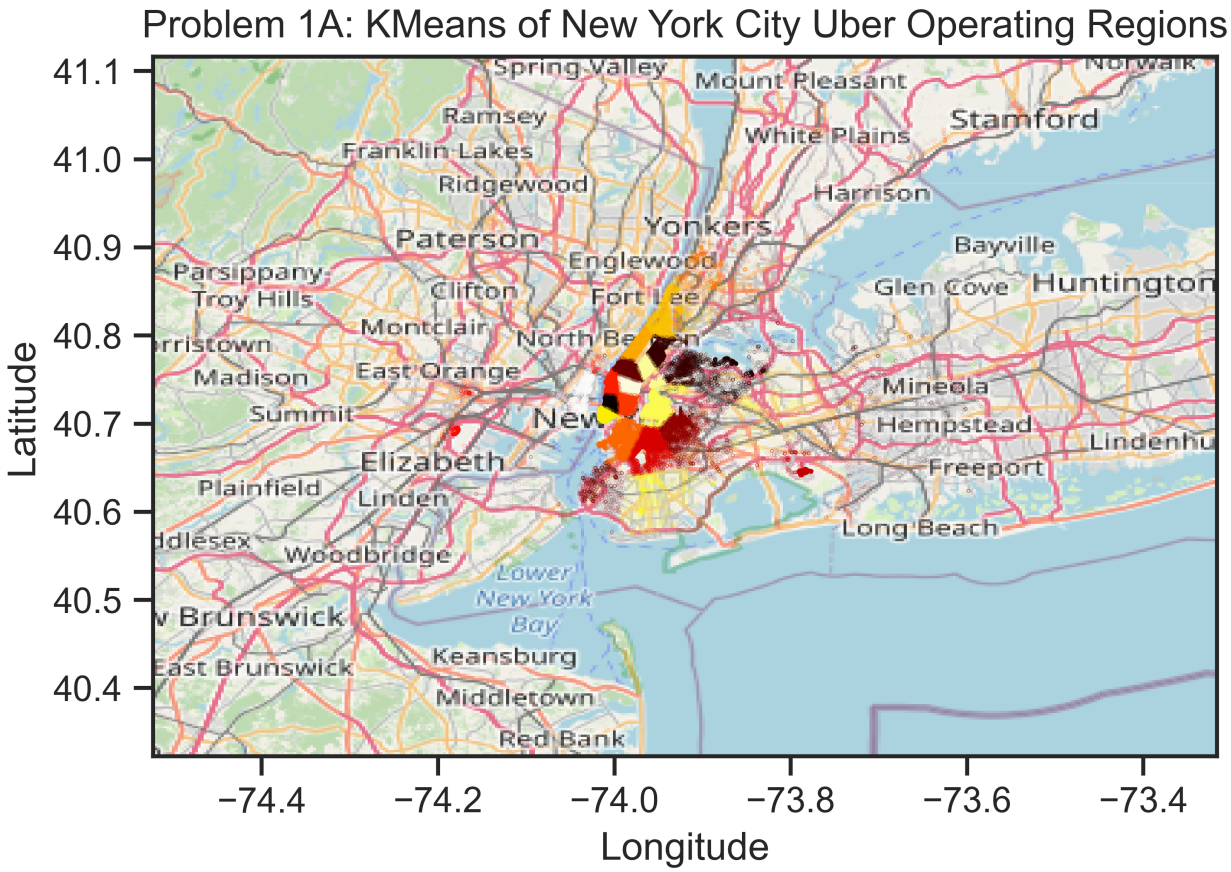
```
    zorder=1,
    alpha= 0.5,
    s=0.001
)
ax.scatter(clustercenters2[:,0],clustercenters2[:,0], color = 'red', s = 0.1)
ax.set_xlim(box[0]+.25,box[1]-1.25)
ax.set_ylim(box[2]+.25,box[3]-1)
ax.imshow(
    ny_map,
    zorder=0,
    extent=box,
    aspect= 'equal'
)
ax.set_xlabel('Longitude')
ax.set_ylabel('Latitude');
plt.title('Problem 1A: KMeans of New York City Uber Operating Regions')
```

Out[70]:
```
Text(0.5, 1.0, 'Problem 1A: KMeans of New York City Uber Operating Regions')
```




```
<Figure size 600x400 with 0 Axes>
```





Part B - Create a Stochastic Model of Pickups

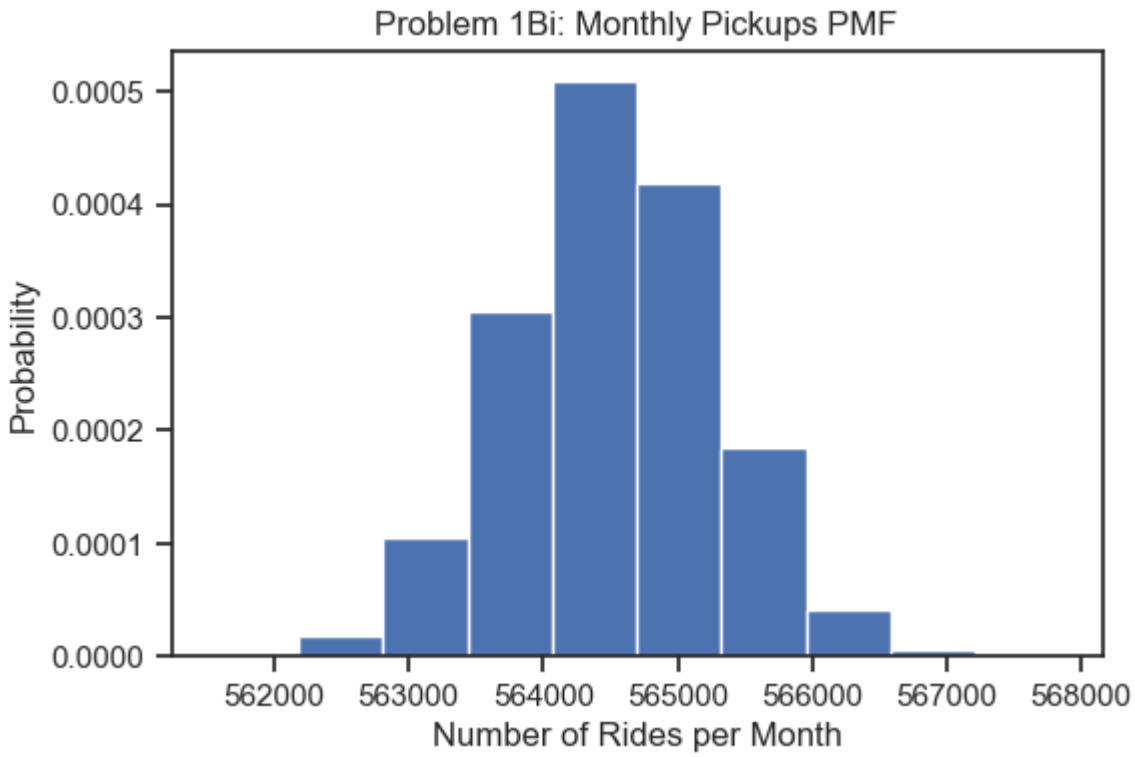
One of the key ingredients for a more sophisticated approach to optimizing the operations of Uber would involve the construction of a stochastic model of the demand for pickups. The ideal model for this problem is the Poisson Point Process. However, we are going to do something simpler, using the Gaussian mixture model and a Poisson random variable. The model will not have a time component, but it will allow us to sample the number and locations of pickups during a typical month. We will guide you through the process of constructing this model.

Subpart B.I - Random variable capturing number of monthly pickups

Find the rate of monthly pickups (ignore the fact that months may differ by a few days) and use it to define a Poisson random variable corresponding to the monthly number of pickups. Use `scipy.stats.poisson` to initialize this random variable. Sample from it 10,000 times and plot the histogram of the samples to get a feeling about the corresponding probability mass function.

In [71]:
```
##Part 1BI
from sklearn.mixture import GaussianMixture
from scipy.stats import poisson

pickupTime = p1_data['Date/Time']
days = 30 #if wanted to seperate into days
pickupRate = math.floor(len(pickupTime)) #rate of pickups per month


samples = 10000

r = poisson.rvs(pickupRate, size=samples)

plt.figure()
plt.hist(r, bins = 10, density = True)
plt.title('Problem 1Bi: Monthly Pickups PMF')
plt.xlabel('Number of Rides per Month')
plt.ylabel('Probability')
```

Out[71]:
```
Text(0, 0.5, 'Probability')
```



Subpart B.II - Estimate the spatial density of pickups

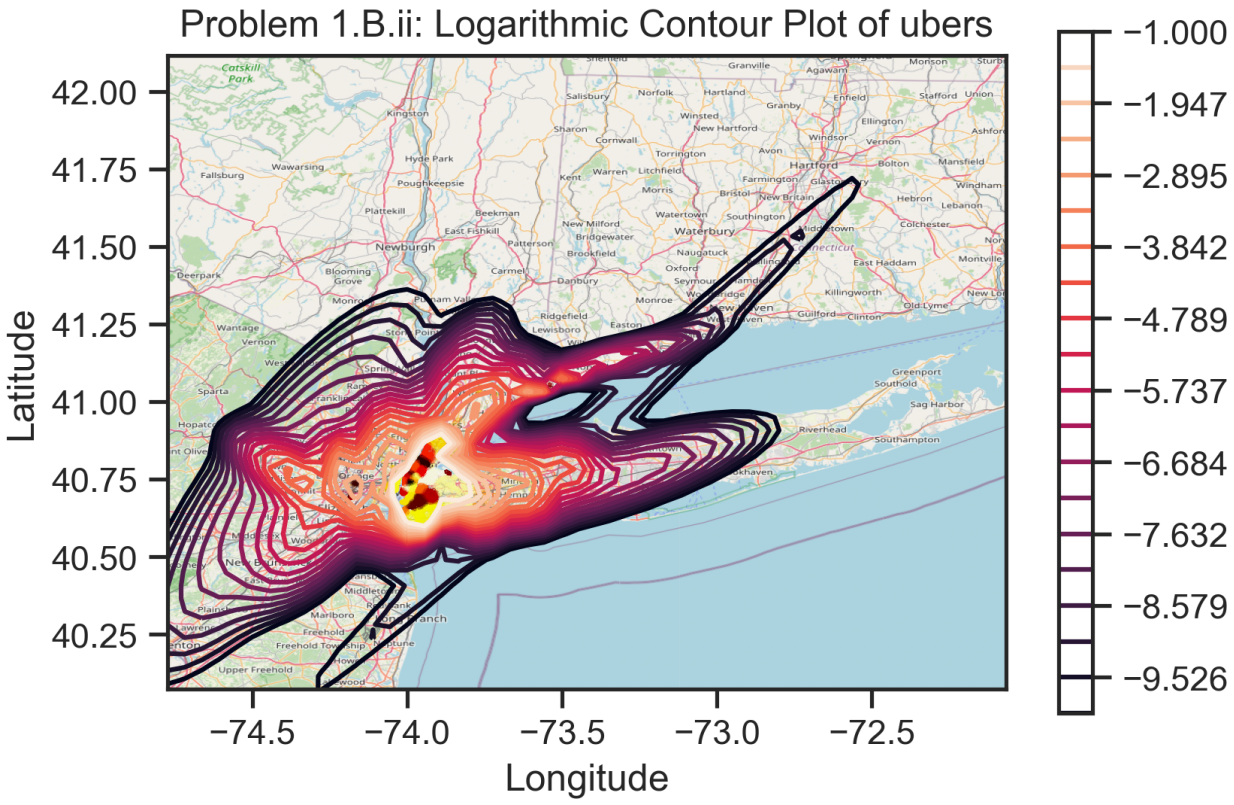
Fit a Gaussian Mixture model to the pickup data. **Do not use the Bayesian Information Criterion** to decide how many components to keep. This would take quite a bit of time for this problem. Simply use 40 mixture components. Plot the contour of the logarithm of the probability density on the New York City map.

In [94]:

```
##Part 1Bii

modelB = GaussianMixture(n_components=40).fit(p1_train_data)
pickupMeans = modelB.means_
labels = modelB.predict(p1_train_data)
#pickupCov = modelB.covariances_

x = np.linspace(box[0],box[1])
y = np.linspace(box[2],box[3])
X, Y = np.meshgrid(x, y)

# Get PDF on grid points
XX = np.array([X.ravel(), Y.ravel()]).T
Z = modelB.score_samples(XX)
Z = Z.reshape(X.shape)



fig, ax = plt.subplots(dpi=300)
ax.scatter(
    p1_train_data.Lon,
    p1_train_data.Lat,
    c=labels,
    cmap = 'hot',
    zorder=1,
```

```
    alpha= 0.5,
    s=0.001
)
ax.set_xlim(box[0],box[1])
ax.set_ylim(box[2],box[3])
ax.imshow(
    ny_map,
    zorder=0,
    extent=box,
    aspect= 'equal'
)
ax.set_xlabel('Longitude');
ax.set_ylabel('Latitude');
c = ax.contour(
    X,
    Y,
    Z,
    levels=np.linspace(-10, -1.0 , 20))
plt.colorbar(c)
plt.title('Problem 1.B.ii: Logarithmic Contour Plot of ubers')
```

Out[94]: `Text(0.5, 1.0, 'Problem 1.B.ii: Logarithmic Contour Plot of ubers')`



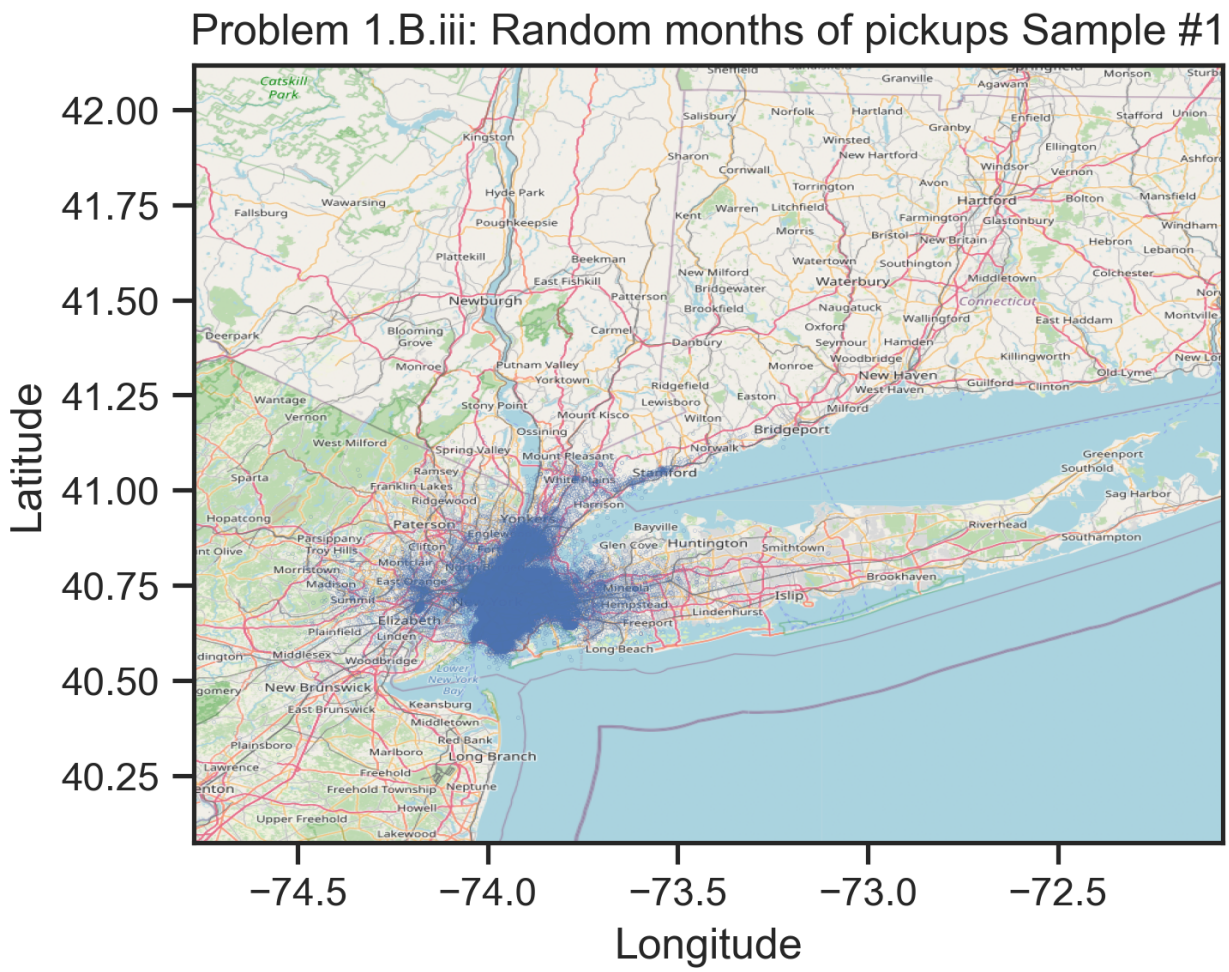
Subpart B.III - Sample some random months of pickups

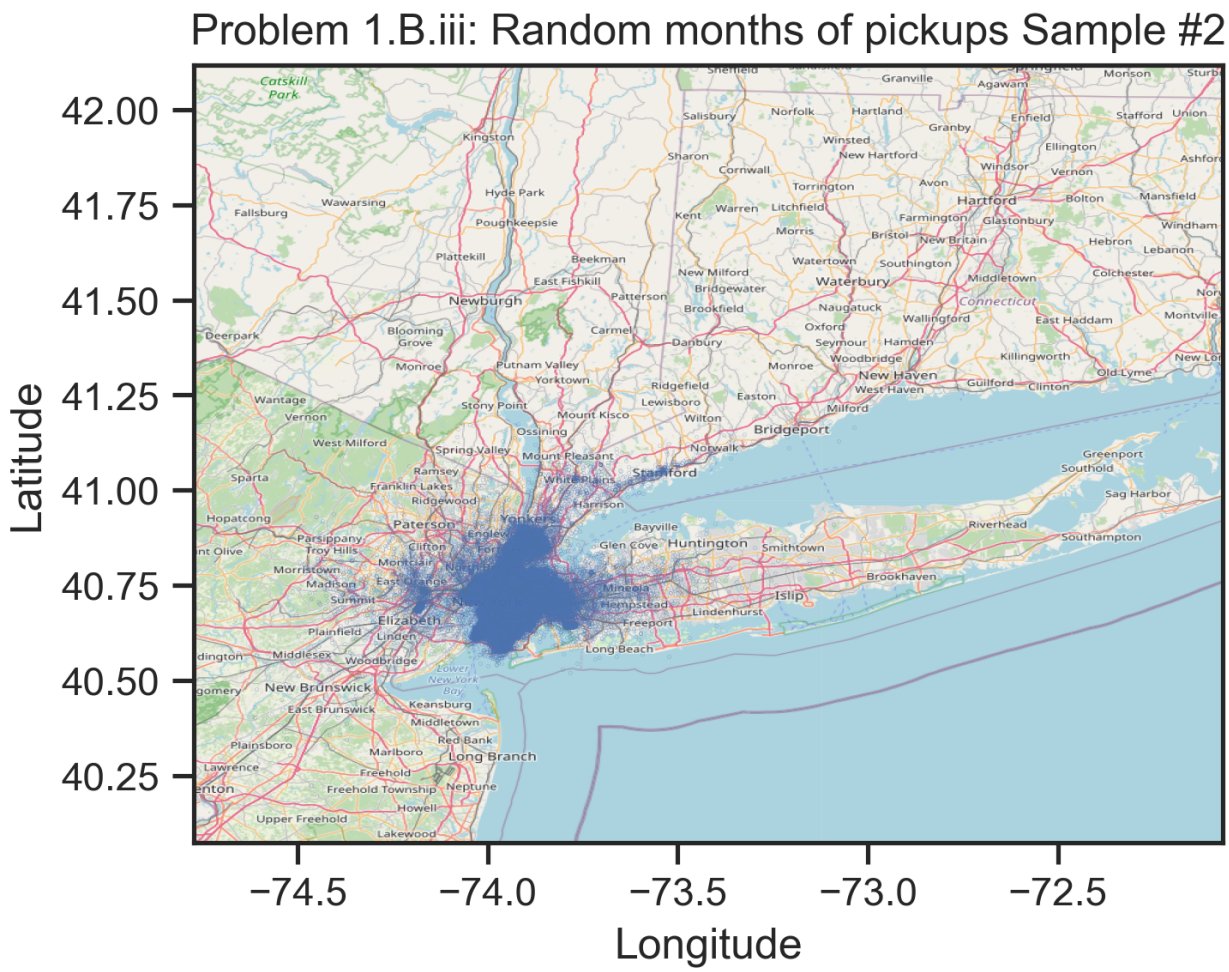
Now that you have a model that gives you the number of pickups and a model that allows you to sample a pickup location, sample five different datasets (number of pickups and location of each pick) from the combined model and visualize them on the New York map.

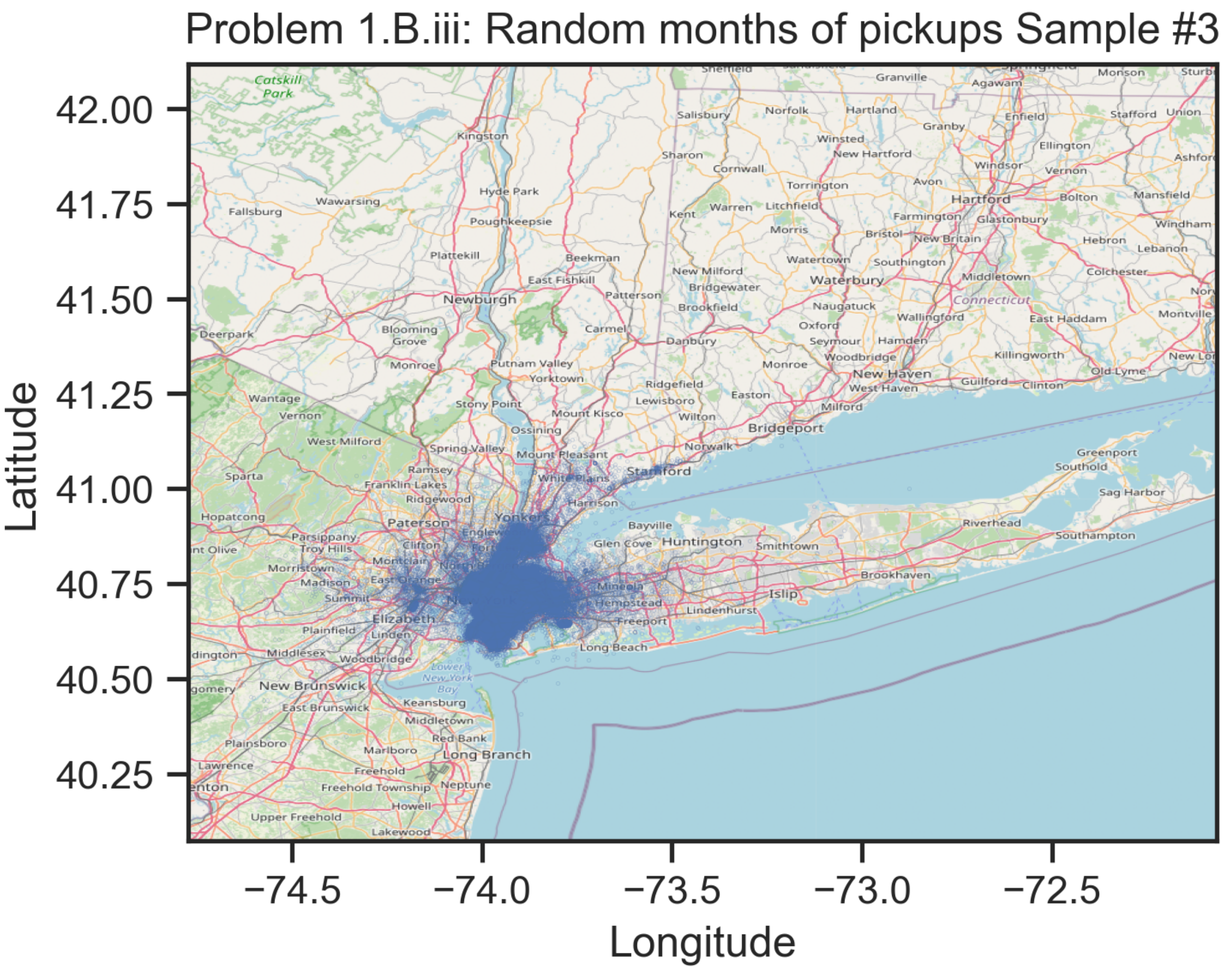
Hint: Don't get obsessed with making the model perfect. It's okay if a few of the pickups are on water...

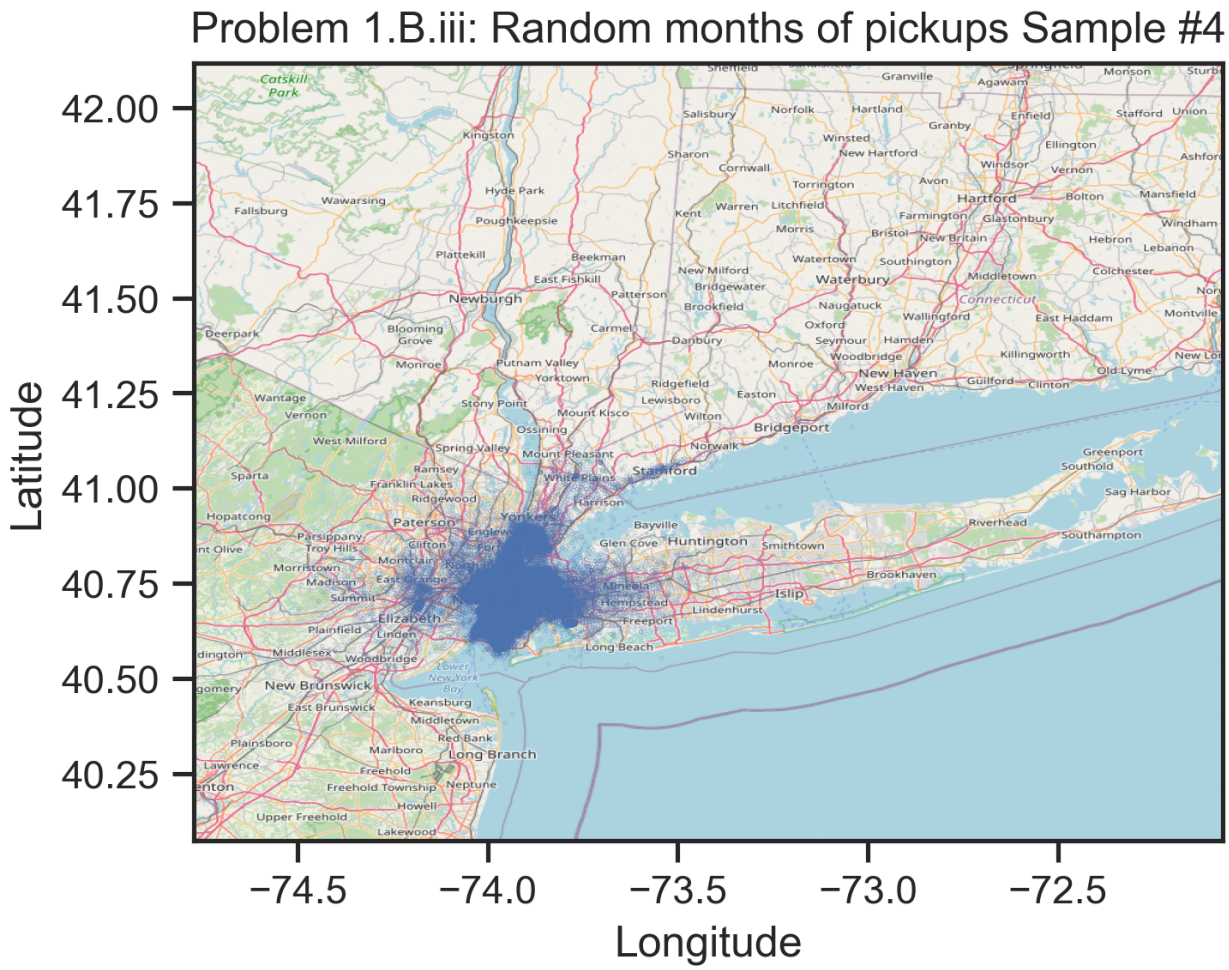
In [73]:

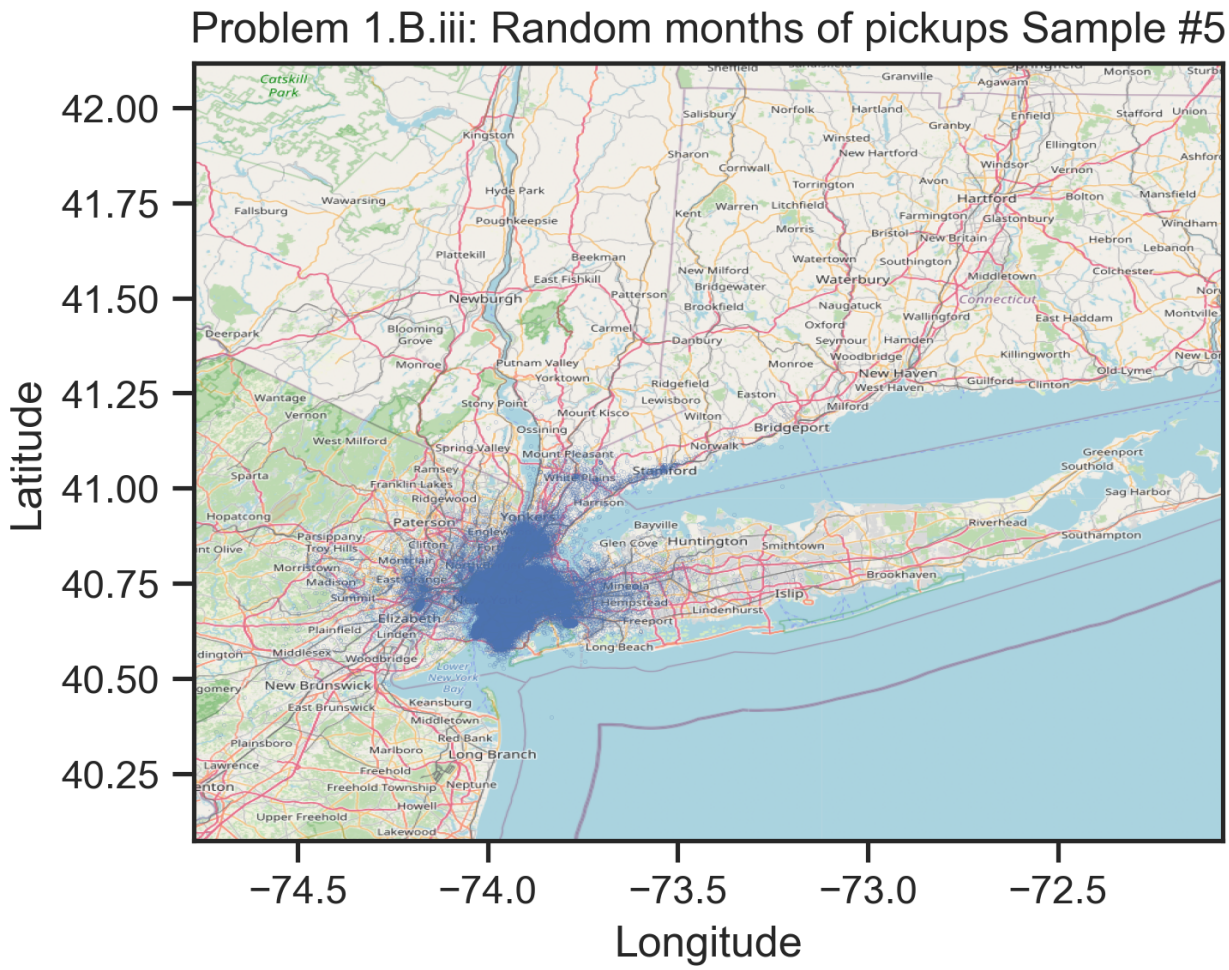
```
# part 1.b.iii
samplingnum = 5

for i in range(samplingnum):
  locations = modelB.sample(pickupRate)

  locations = locations[0] #reformat to make next two lines easy
  loc_lon = locations[:,0] #longitudes
  loc_lat = locations[:,1]#latitudes

  fig, ax = plt.subplots(dpi=300)
  ax.scatter(
    loc_lon,
    loc_lat,
    zorder=1,
    alpha= 0.5,
    c='b',
    s=0.01)
  ax.scatter(
    loc_data.Lon,
    loc_data.Lat,
    zorder=1,
    alpha= 0.5,
    c='b',
    s=0.001
    )
  ax.set_xlim(box[0],box[1])
  ax.set_ylim(box[2],box[3])
  ax.imshow(
      ny_map,
      zorder=0,
      extent=box,
      aspect= 'equal'
    )
  ax.set_xlabel('Longitude')
  ax.set_ylabel('Latitude');
  plt.title(f'Problem 1.B.iii: Random months of pickups Sample #{i+1}')
```


Problem 1.B.iii: Random months of pickups Sample #1
Latitude
42.00
41.75
41.50
41.25
41.00
40.75
40.50
40.25
−74.5
−74.0
−73.5
−73.0
−72.5
Longitude
Catskill Park
Kingston
Hyde Park
Poughkeepsie
Newburgh
Wawarsing
Fallsburg
Plattekill
Beekman
East Fishkill
Deerpark
Blooming Grove
Monroe
Carmel
Putnam Valley
Yorktown
Stony Point
Wantage
Vernon
West Milford
Mount Kisco
Ossining
Spring Valley
Mount Pleasant
Sparta
Ramsey
Franklin Lakes
Ridgewood
White Plains
Harrison
Hopatcong
Paterson
Yonkers
Parsippany
Troy Hills
Clifton
Englewood
Bayville
Glen Cove
Huntington
Smithtown
Riverhead
Southampton
Morristown
Madison
East Orange
Montclair
Summit
Mineola
Hempstead
Islip
Brookhaven
Elizabeth
Plainfield
Linden
Middlesex
Woodbridge
Freeport
Lindenhurst
Long Beach
New Brunswick
East Brunswick
Kearsburg
Middletown
Red Bank
Plainsboro
Marlboro
Long Branch
Lawrence
Freehold
Freehold Township
Neptune
Howell
Upper Freehold
Lakewood
Lower New York Bay
Salisbury
Norfolk
Hartland
Granville
Agawam
Granby
Enfield
Stafford
Union
Winsted
New Hartford
Sharon
Cornwall
Torrington
Avon
Windsor
Vernon
Ellington
Hartford
Bolton
Mansfield
Kent
Warren
Litchfield
Farmington
Glastonbury
Windham
Morris
Bristol
New Britain
Hebron
Lebanon
Watertown
Southington
Colchester
New Milford
Bridgewater
Waterbury
Middletown
Brookfield
Naugatuck
Connecticut
Wallingford
East Haddam
Oxford
Seymour
Hamden
Danbury
Woodbridge
Killingworth
Monroe
New Haven
West Haven
Guilford
Clinton
Old Lyme
Ridgefield
Lewisboro
Easton
Milford
Wilton
Bridgeport
Norwalk
Stamford
Greenport
Southold
Sag Harbor


Problem 1.B.iii: Random months of pickups Sample #2
Latitude
42.00
41.75
41.50
41.25
41.00
40.75
40.50
40.25
−74.5
−74.0
−73.5
−73.0
−72.5
Longitude
Catskill Park
Kingston
Wawarsing
Fallsburg
Hyde Park
Poughkeepsie
Plattekill
Beekman
East Fishkill
Newburgh
Deerpark
Blooming Grove
Monroe
Carmel
Putnam Valley
Yorktown
Stony Point
Wantage
Vernon
West Milford
Mount Kisco
Ossining
Spring Valley
Mount Pleasant
Sparta
Ramsey
Franklin Lakes
Ridgewood
White Plains
Harrison
Hopatcong
Paterson
Parsippany
Troy Hills
Clifton
Englewood
Yonkers
Bayville
Glen Cove
Huntington
Smithtown
Riverhead
Southampton
Greenport
Southold
Sag Harbor
Brookhaven
Islip
Morristown
Madison
Summit
Mineola
Hempstead
Lindenhurst
Freeport
Elizabeth
Plainfield
Linden
Middlesex
Woodbridge
Long Beach
New Brunswick
Keansburg
East Brunswick
Middletown
Red Bank
Plainsboro
Marlboro
Long Branch
Lawrence
Freehold
Freehold Township
Neptune
Howell
Upper Freehold
Lakewood
Ridgefield
Lewisboro
Easton
Wilton
Norwalk
Bridgeport
Milford
New Haven
West Haven
Guilford
Clinton
Old Lyme
Killingworth
Woodbridge
Seymour
Hamden
Oxford
Naugatuck
Waterbury
Brookfield
Danbury
New Milford
Bridgewater
Patterson
Connecticut
Wallingford
East Haddam
Middletown
Watertown
Southington
Bristol
New Britain
Farmington
Glastonbury
Hartford
Avon
Torrington
Litchfield
Morris
Warren
Kent
Cornwall
Sharon
New Hartford
Winsted
Granby
Salisbury
Norfolk
Hartland
Granville
Agawam
Enfield
Stafford
Union
Ellington
Windsor
Vernon
Bolton
Mansfield
Windham
Hebron
Lebanon
Colchester
Montville
Old Lyme
Lower New York Bay


Problem 1.B.iii: Random months of pickups Sample #3
Latitude
42.00
41.75
41.50
41.25
41.00
40.75
40.50
40.25
Longitude
−74.5
−74.0
−73.5
−73.0
−72.5


Problem 1.B.iii: Random months of pickups Sample #4
Latitude
42.00
41.75
41.50
41.25
41.00
40.75
40.50
40.25
−74.5
−74.0
−73.5
−73.0
−72.5
Longitude
Catskill Park
Kingston
Hyde Park
Poughkeepsie
Newburgh
Wawarsing
Fallsburg
Deerpark
Blooming Grove
Monroe
Wantage
Vernon
West Milford
Sparta
Hopatcong
Parsippany
Troy Hills
Morristown
Madison
Summit
Elizabeth
Plainfield
Linden
Middlesex
Woodbridge
New Brunswick
East Brunswick
Kearsburg
Middletown
Red Bank
Long Branch
Plainsboro
Marlboro
Freehold
Freehold Township
Neptune
Howell
Upper Freehold
Lakewood
Lawrence
Plattekill
Beekman
East Fishkill
Putnam Valley
Yorktown
Stony Point
Ossining
Spring Valley
Mount Pleasant
Ramsey
Franklin Lakes
Ridgewood
Paterson
Englewood
Clifton
Montclair
East Orange
Carmel
Patterson
Danbury
Ridgefield
Lewisboro
Mount Kisco
White Plains
Yonkers
Harrison
Stamford
Norwalk
Wilton
Bridgeport
Milford
West Haven
New Haven
Easton
Monroe
Woodbridge
Oxford
Seymour
Hamden
Waterbury
Naugatuck
Brookfield
New Milford
Bridgewater
Watertown
Southington
Middletown
Wallingford
Meriden
Killingworth
Guilford
Clinton
Old Lyme
East Haddam
Connecticut
Bayville
Glen Cove
Huntington
Smithtown
Islip
Brookhaven
Riverhead
Southampton
Greenport
Southold
Sag Harbor
Hempstead
Freeport
Lindenhurst
Long Beach
Lower New York Bay
Salisbury
Norfolk
Hartland
Granville
Agawam
Enfield
Stafford
Union
Granby
Winsted
New Hartford
Sharon
Cornwall
Torrington
Avon
Windsor
Vernon
Ellington
Hartford
Bolton
Mansfield
Kent
Warren
Litchfield
Farmington
Glastonbury
Morris
Bristol
New Britain
Windham
Hebron
Lebanon
Colchester
Morson
Sturbridge
Ashford



Problem 2 - Counting Celestial Objects

Consider this picture of a patch of sky taken by the Hubble Space Telescope:

In [74]:
```
url = 'https://raw.githubusercontent.com/PredictiveScienceLab/data-analytics-se/master/
download(url)
```

 galaxies

This picture includes many galaxies, but also some starts. We are going to create a machine learning model capable of counting the number of objects in such images. Our model is not going to be able to differentiate between the different types of objects, and it will not be very accurate, but it does form the basis of more sophisticated approaches. The idea is as follows:

- Convert the picture to points sampled according to the intensity of light.
- Apply Gaussian mixture on the resulting points.
- Use the Bayesian Information Criterion to identify the number of components in the picture.
- Associate the number of components with the actual number of celestial objects.

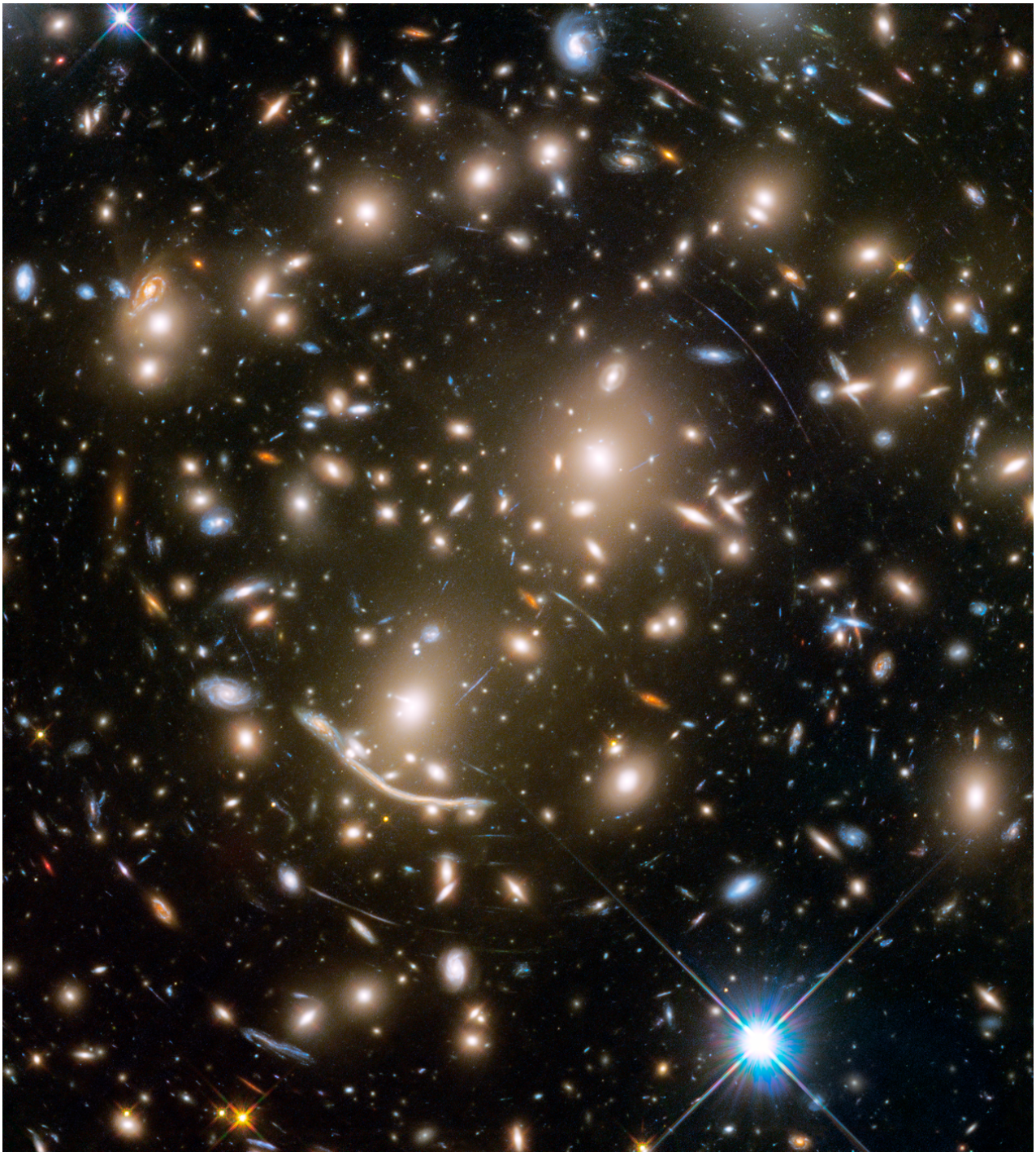
I will set you up with the first step. You will have to do the last three.

We are going to load the image with the Python Imaging Library (PIL) which allows us to apply a few

basic transformations to the image:

In [75]:
```
from PIL import Image
hubble_image = Image.open('galaxies.png')
# here is how to see the image
hubble_image
```

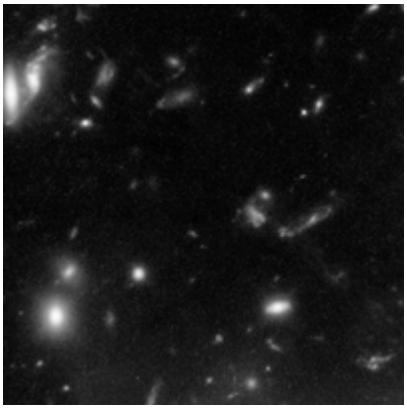
Out[75]:



Now we are going to convert it to gray scale and crop it to make the problem a little bit easier:

In [76]:
```
img = hubble_image.convert('L').crop((100, 100, 300, 300))
img
```

Out[76]:



Remember that black-and white images are matrices:

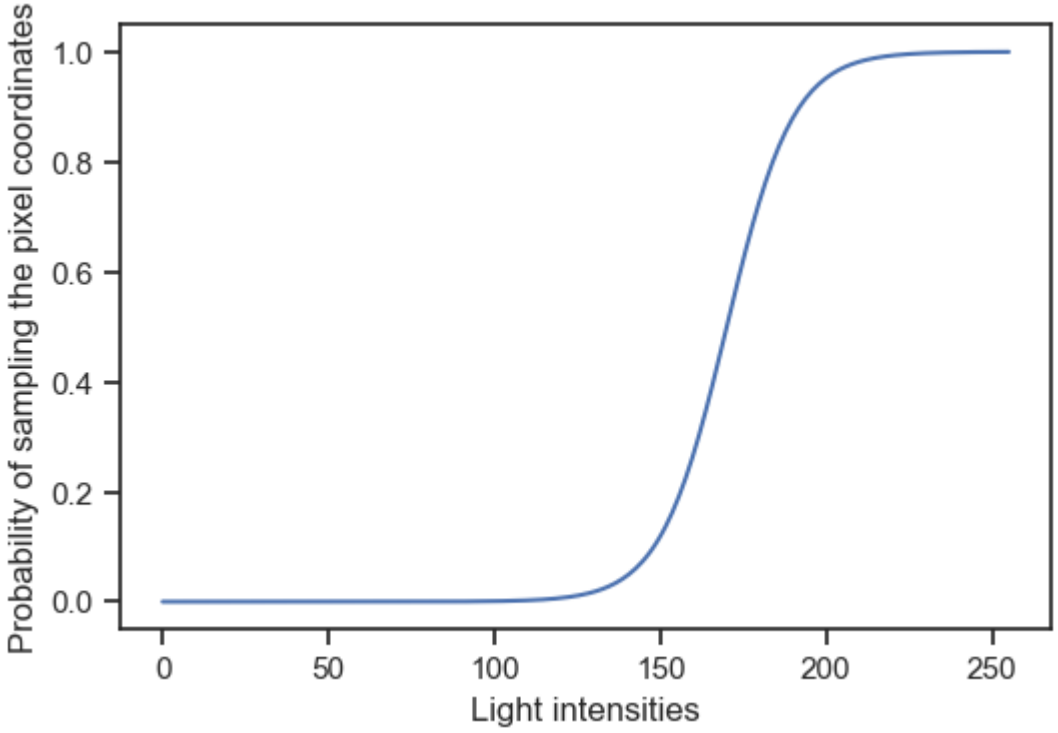
In [77]:
```
img_ar = np.array(img)
img_ar
```

Out[77]:
```
array([[ 7, 11, 11, ..., 28, 62, 88],
       [12, 12, 11, ..., 29, 47, 86],
       [18, 13, 11, ..., 24, 34, 87],
       ...,
       [24, 12, 15, ..., 43, 47, 40],
       [23, 12, 19, ..., 48, 49, 40],
       [18, 18, 23, ..., 50, 49, 41]], dtype=uint8)
```

The minimum number if 0 corresponding to black and the maximum number is 255 corresponding to white. Anything in between is some shade of gray.

Now, imagine that each pixel is associated with some cordinates. Without loss of generality, let's assume that each pixel is some coordinate in $[0,1]^2$. We will loop over each of the pixels and sample its coordinates in a way that increases with increasing light intensity. To achieve this, we will pass the intensity values of each pixels through a sigmoid with parameters that can be tuned. Here is this sigmoid:

In [78]:
```
intensities = np.linspace(0, 255, 255)
fig, ax = plt.subplots()
alpha = 0.1
beta = 255 / 1.5
ax.plot(
    intensities,
    1.0 / (1.0 + np.exp(-alpha * (intensities - beta)))
);
ax.set_xlabel('Light intensities')
ax.set_ylabel('Probability of sampling the pixel coordinates');
```



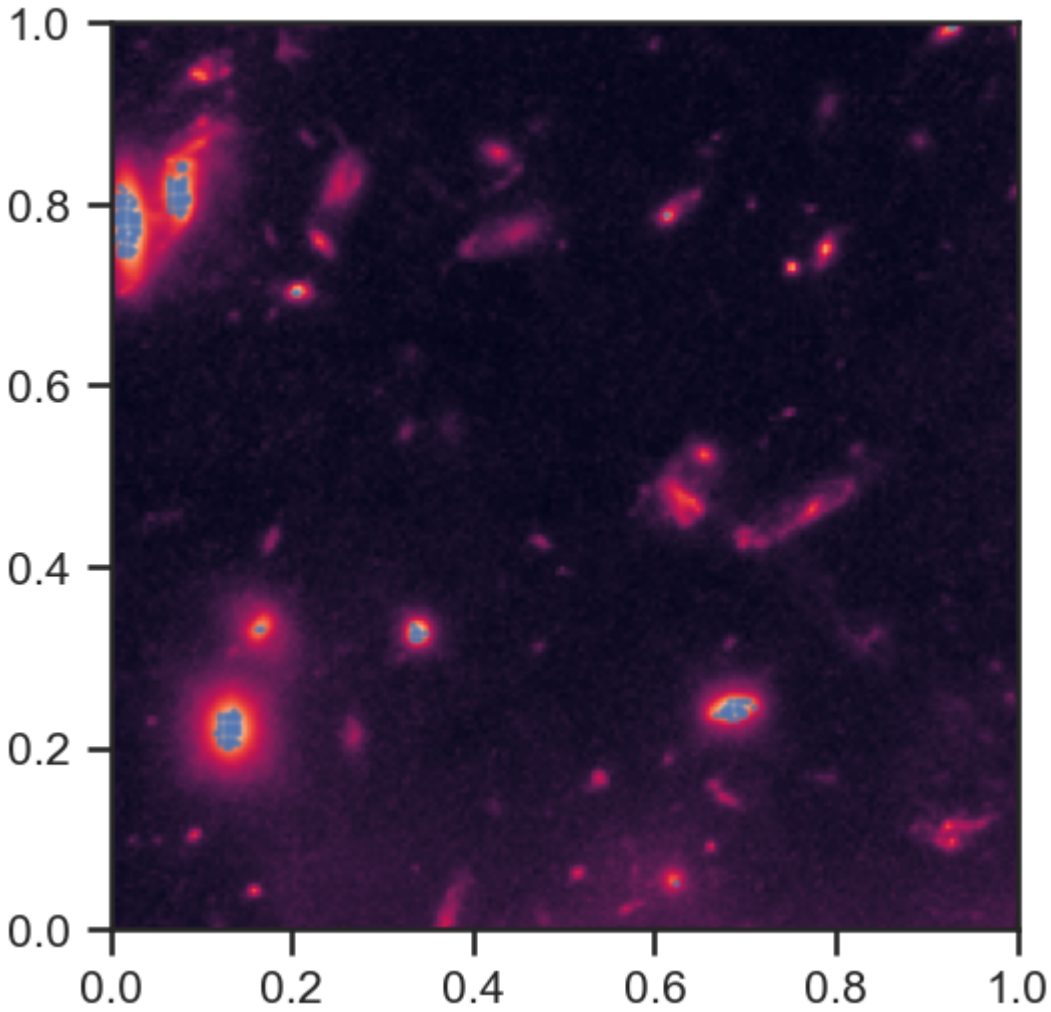
And here is the code that samples the pixel coordinates. I am organizing it into a function because we may want to use it with different pictures:

In [79]:

```
def sample_pixel_coords(img, alpha, beta):
    """
    Samples pixel coordinates based on a probability defined as the sigmoid of the inte

    Arguments:

        img    -     The gray scale pixture from which we sample as an array
        alpha      -     The scale of the sigmoid
        beta       -     The offset of the sigmoid
    """
    img_ar = np.array(img)
    x = np.linspace(0, 1, img_ar.shape[0])
    y = np.linspace(0, 1, img_ar.shape[1])
    X, Y = np.meshgrid(x, y)
    img_to_locs = []
    # Loop over pixels
    for i in range(img_ar.shape[1]):
        for j in range(img_ar.shape[0]):
            # Calculate the probability of the pixel by looking at each
            # light intensity
            prob = 1.0 / (1.0 + np.exp(-alpha * (img_ar[j, i] - beta)))
            # Pick a uniform random number
            u = np.random.rand()
            # If u is smaller than the desired probability,
            # the consider the coordinates of the pixel sampled
            if u <= prob:
                img_to_locs.append((Y[i, j], X[-i-1, -j-1]))
    # Turn img_to_locs into a numpy array
    img_to_locs = np.array(img_to_locs)
    return img_to_locs
```

Let's test the code:

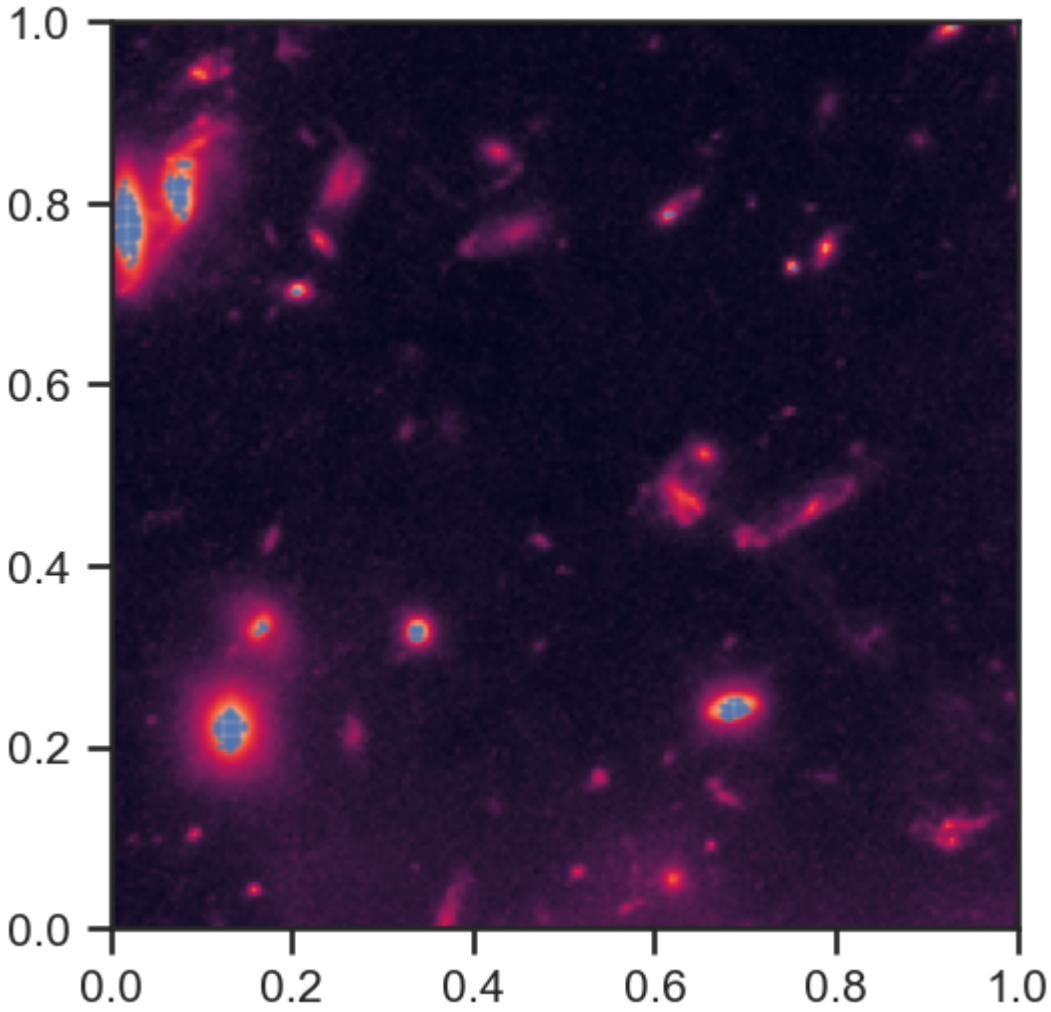
In [80]:
```
locs = sample_pixel_coords(img, alpha=0.1, beta=200)
fig, ax = plt.subplots(dpi=150)
ax.imshow(img, extent=((0, 1, 0, 1)), zorder=0)
ax.scatter(
    locs[:, 0],
    locs[:, 1],
    zorder=1,
    alpha=0.5,
    c='b',
    s=1
);
```



Note that by playing with α and β you can make the whole thing more or less sensitive to the light intensity.

In [81]:
```
locs = sample_pixel_coords(img, alpha=0.1, beta=200)
fig, ax = plt.subplots(dpi=150)
ax.imshow(img, extent=((0, 1, 0, 1)), zorder=0)
ax.scatter(
    locs[:, 0],
    locs[:, 1],
    zorder=1,
```

```
    alpha=0.5,
    c='b',
    s=1
);
```



Once you have completed the code, try it out the following images. Feel free to play with α and β to improve the performance. **Do not try to make a perfect model. To do so, we would have to go beyond the Gaussian mixture model. This is just a homework problem.**

Here is a helper function for visualizing what you get:

In [82]:

```
def visualize_counts(img, objs, model, locs):
    """Visualize the counts.

    Arguments
    img    --  The image.
    objs   --  Returned by count_objs()
    model  --  Returned by count_objs()
    locs   --  Returned by count_objs()
    """
    fig, ax = plt.subplots(dpi=150)
    ax.imshow(img, extent=((0, 1, 0, 1)))
    for i in range(model.means_.shape[0]):
        ax.plot(
            model.means_[i, 0],
```

```
            model.means_[i, 1],
            'rx',
            markersize=(
                10.0 * model.weights_.shape[0]
                * model.weights_[i]
            )
        )
    ax.scatter(
        locs[:, 0],
        locs[:, 1],
        zorder=1,
        alpha=0.5,
        c='b',
        s=1
    )
    ax.set_title('Counted {0:d} objects!'.format(objs));
```

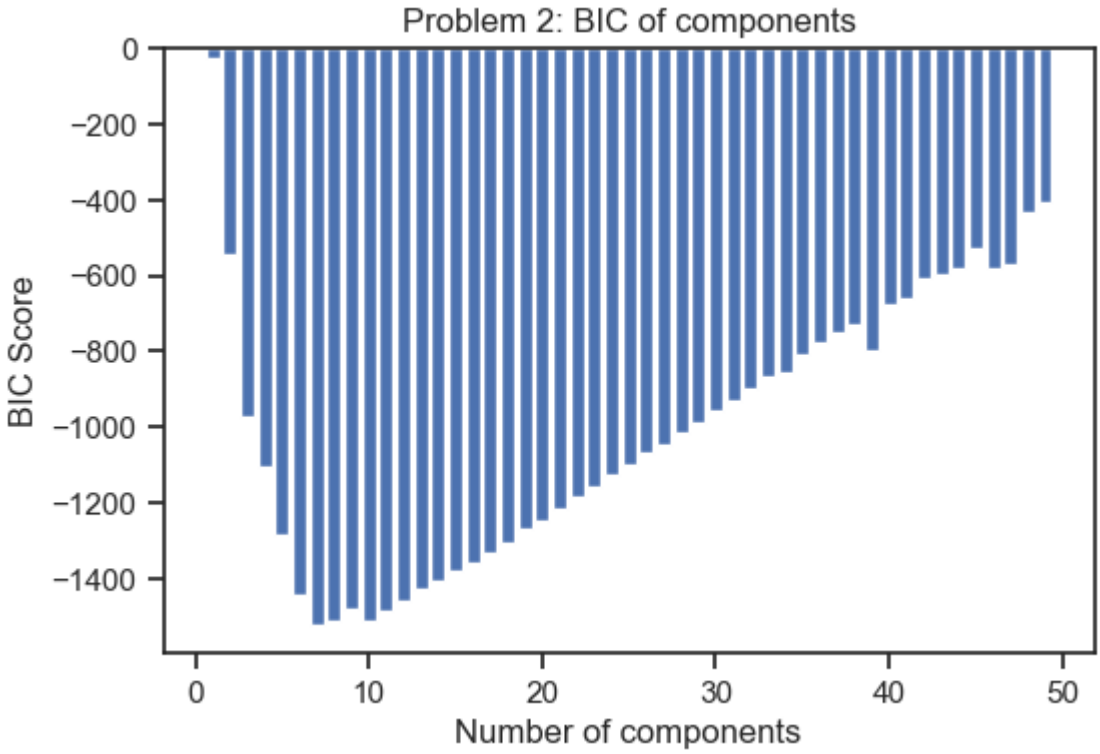
Here is how to use the code:

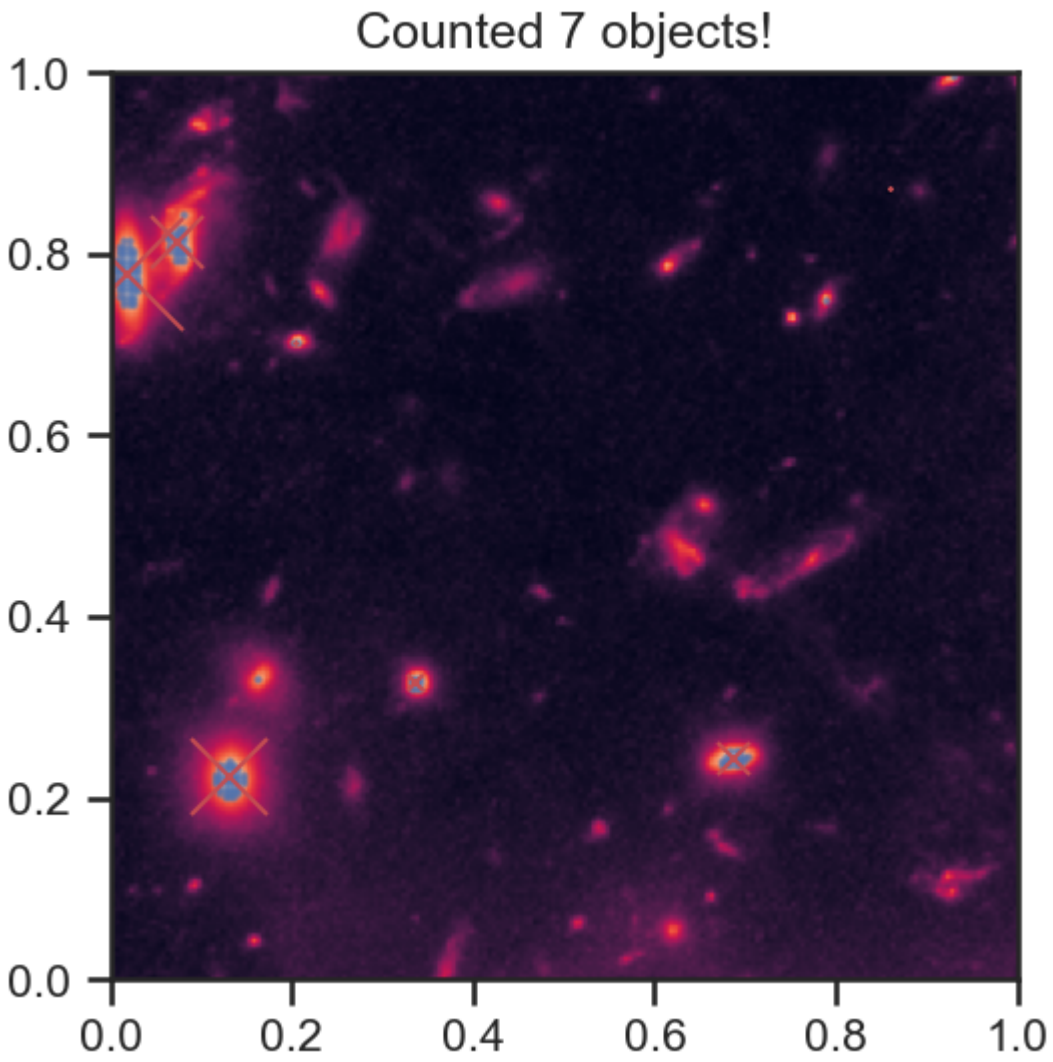
In [83]:

```
#problem 2
from sklearn.mixture import GaussianMixture

def count_objs(img, alpha, beta, nc_min=1, nc_max=50):
    """Count objects in image.

    Arguments:
        img       -     The image
        alpha     -     The scale of the sigmoid
        beta      -     The offset of the sigmoid
        nc_min    -     The minimum number of components to consider
        nc_max    -     The maximum number of components to consider
    """
    locs = sample_pixel_coords(img, alpha, beta)
    # **** YOUR CODE HERE ****
    # Use BIC to search for the best GaussianMixture model
    # with components between nc_min and nc_max
    # YOU CAN PULL THIS OFF BY COPY-PASTING MATERIAL FROM
    # LECTURE 17
    # Set the following variables

    bics = np.ndarray(nc_max - 1,)
    models = []


    for nc in range(nc_min, nc_max):
        m = GaussianMixture(n_components=nc).fit(locs)
        bics[nc - 1] = m.bic(locs)
        models.append(m)

    min_bics = min(bics)

    best_nc = np.where(bics == min_bics) #('Set this equal to the number of components
    best_nc = best_nc[0]
    best_nc = best_nc[0] + 1 #converting to int

    print(f'Number of Components are: {best_nc}')
    best_model = GaussianMixture(best_nc).fit(locs)# NotImplemented('Set this equatl to
    fig, ax = plt.subplots()
    ax.bar(range(1, nc_max), bics)
```

```
    ax.set_ylabel('BIC Score')
    ax.set_xlabel('Number of components');
    plt.title('Problem 2: BIC of components')

    return best_nc, best_model, locs

objs, model, locs = count_objs(img, alpha=1.0, beta=200)
visualize_counts(img, objs, model, locs)
```

```
D:\School\Programing\Anaconda\envs\BME301\lib\site-packages\sklearn\cluster\_kmeans.py:881: UserWarning: KMeans is known to have a memory leak on Windows with MKL, when there are less chunks than available threads. You can avoid it by setting the environment variable OMP_NUM_THREADS=1.
  warnings.warn(
Number of Components are: 7
```

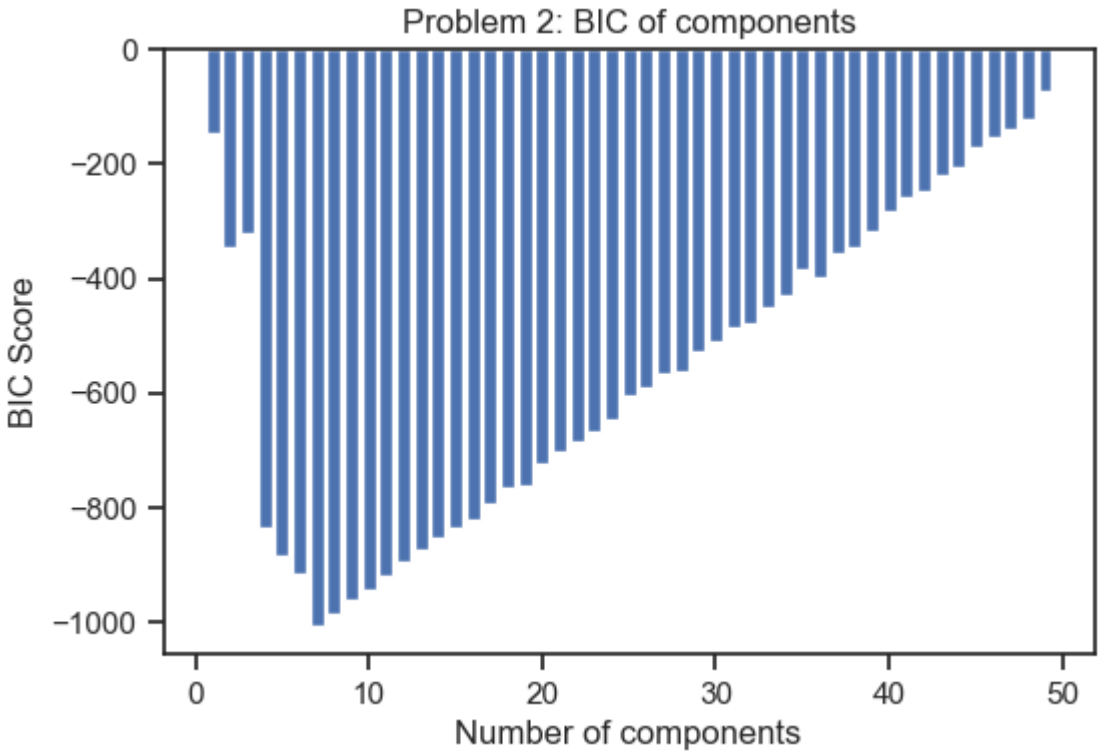


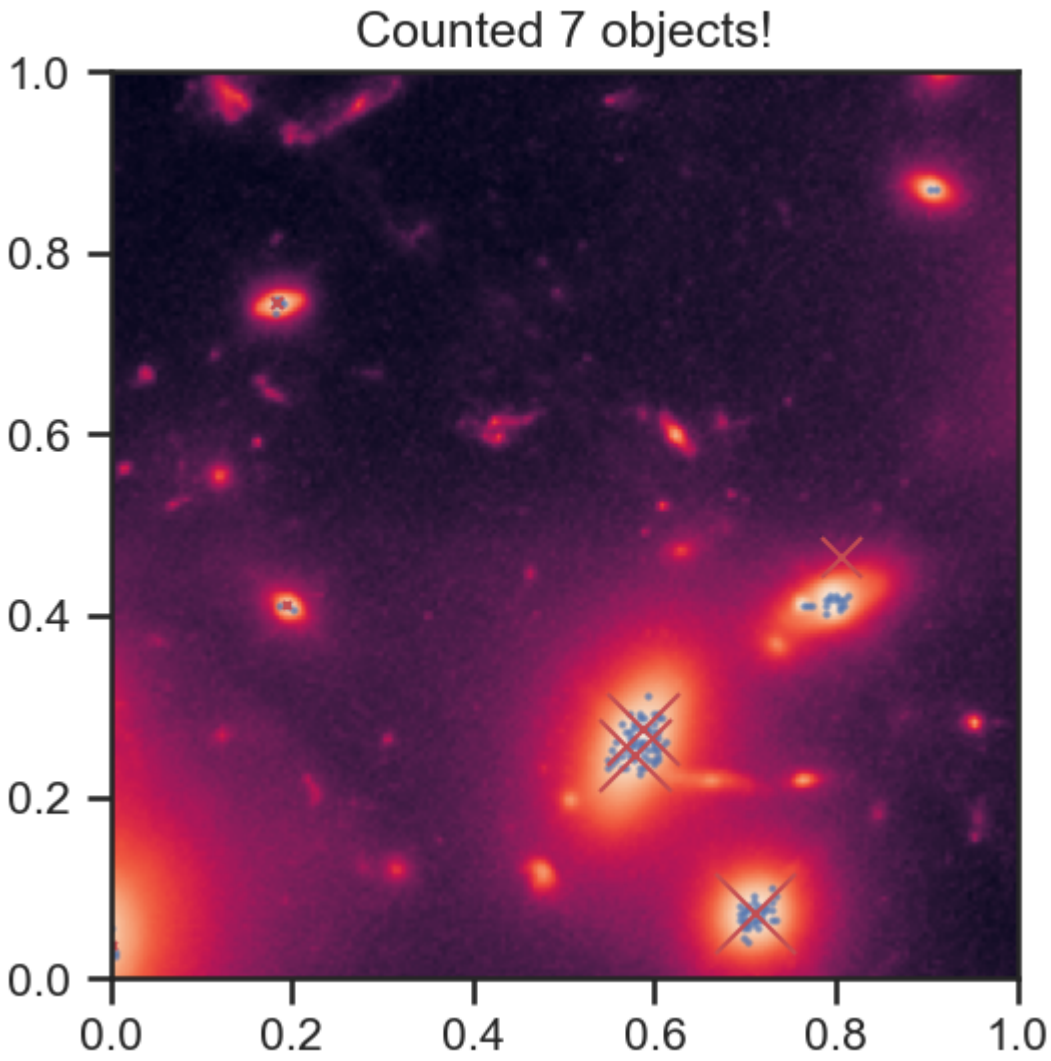


Try this one:

In [96]:
```
img = hubble_image.convert('L').crop((200, 200, 400, 400))
objs, model, locs = count_objs(img, alpha=.1, beta=250)
visualize_counts(img, objs, model, locs)
```

```
D:\School\Programing\Anaconda\envs\BME301\lib\site-packages\sklearn\cluster\_kmeans.py:8
81: UserWarning: KMeans is known to have a memory leak on Windows with MKL, when there a
re less chunks than available threads. You can avoid it by setting the environment varia
ble OMP_NUM_THREADS=1.
  warnings.warn(
Number of Components are: 7
```

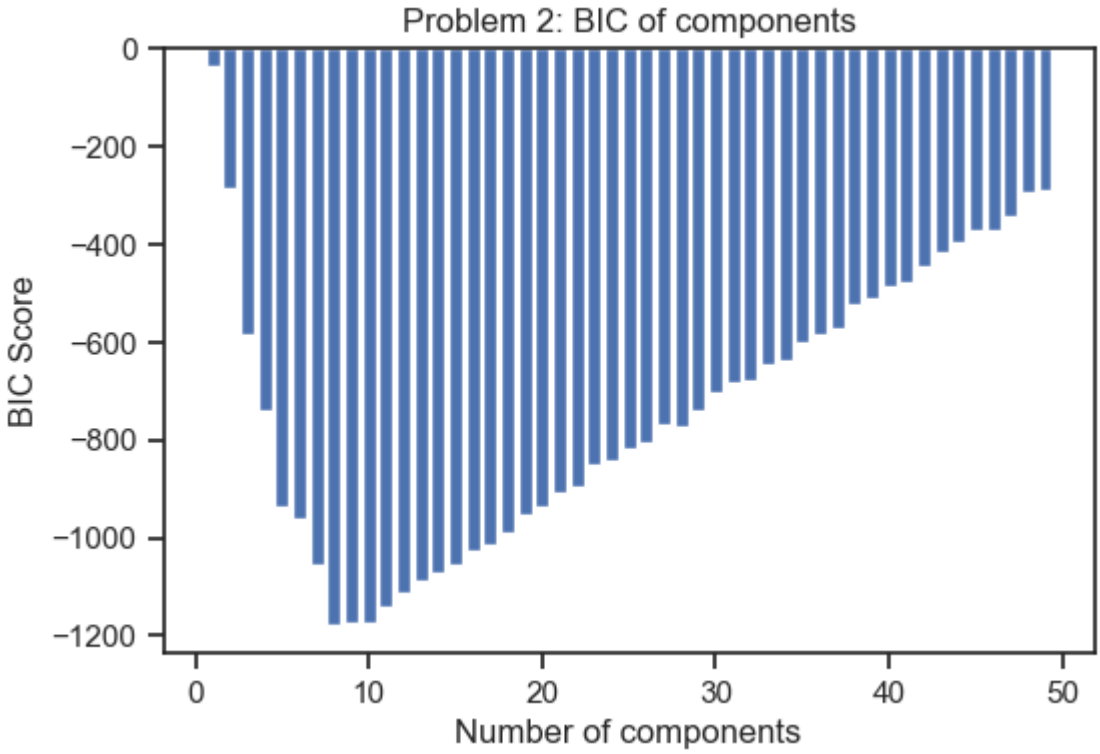


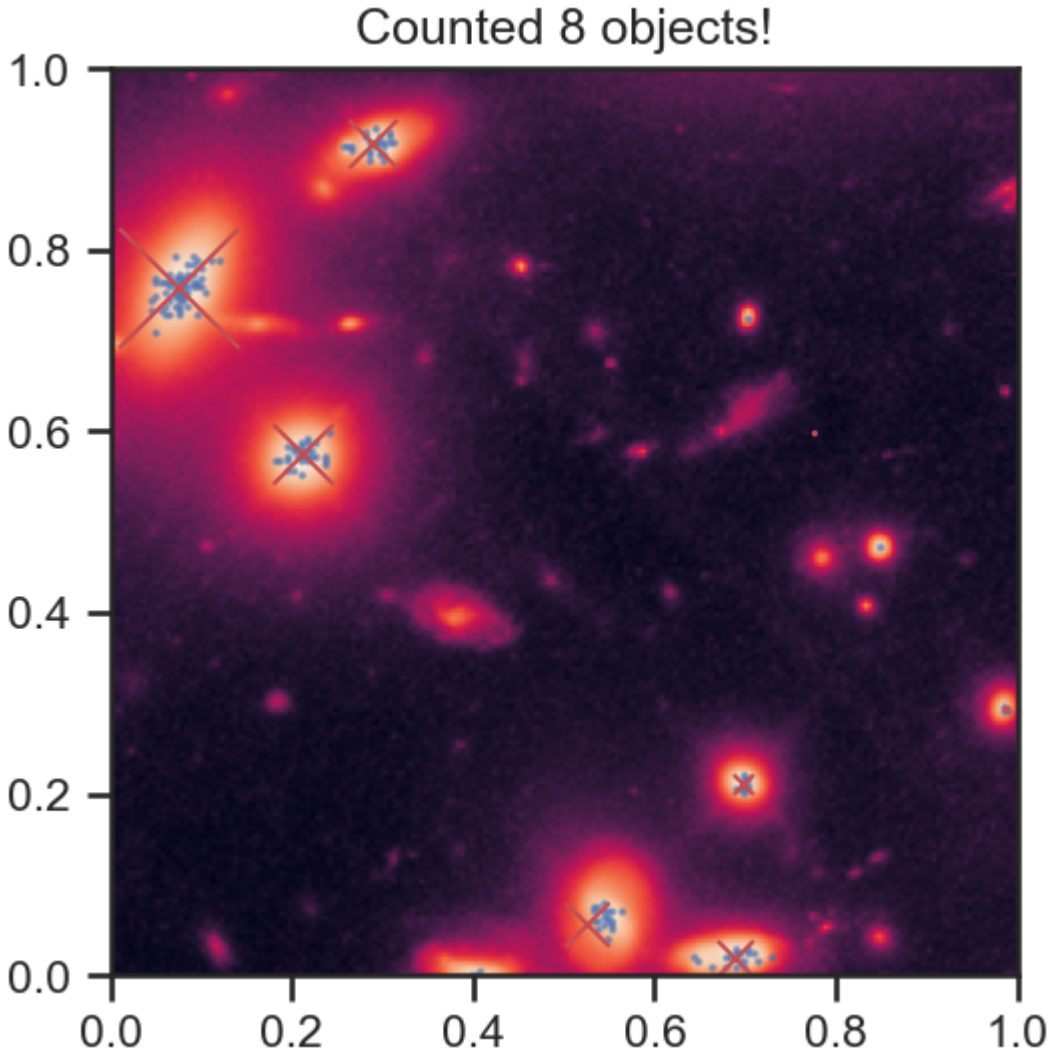


And try this one:

In [91]:
```
img = hubble_image.convert('L').crop((300, 300, 500, 500))
objs, model, locs = count_objs(img, alpha=.1, beta=250)
visualize_counts(img, objs, model, locs)
```

```
D:\School\Programing\Anaconda\envs\BME301\lib\site-packages\sklearn\cluster\_kmeans.py:881: UserWarning: KMeans is known to have a memory leak on Windows with MKL, when there are less chunks than available threads. You can avoid it by setting the environment variable OMP_NUM_THREADS=1.
  warnings.warn(
Number of Components are: 8
```





Problem 3 - Filtering of an Oscillator with Damping

Assume that you are dealing with a one-degree-of-freedom system which follows the equation:

$$\ddot{x} + 2\zeta\omega_0\dot{x} + \omega_0^2 x = u_0 \cos(\omega t),$$

where $x = x(t)$ is the generalized coordinate of the oscillator at time t, and the parameters ζ, ω_0, u_0, and ω are known to you (we will give them specific values later). Furthermore, assume that you are making noisy observations of the *absolute acceleration* at discrete timesteps Δt (also known):

$$y_j = \ddot{x}(j\Delta t) - u_0 \cos(\omega t) + w_j,$$

for $j = 1, \ldots, n$, where $w_j \sim N(0, \sigma^2)$ with σ^2 also known. Finally, assume that the initial conditions for the position and the velocity (you need both to get a unique solution) are given by:

$$x_0 = x(0) \sim N(0, \sigma_x^2),$$

and

$$v_0 = \dot{x} \sim N(0, \sigma_v^2).$$

Of course assume taht σ_x^2 and σ_v^2 are specific numbers that we are going to specify below.

Before I we go over the questions, let's write code that generates the true trajectory of the system at some random initial conditions as well as some observations. We will use the code to generate a synthetic dataset with known ground truth which you will use in your filtering analysis.

The first step we need to do, is to turn the problem into a first order differential equation. This is trivial. We set:

$$\mathbf{x} = \begin{bmatrix} x \\ \dot{x} \end{bmatrix}.$$

Assuming $\mathbf{x} = (x_1, x_2)$, then the dynamics are described by:

$$\dot{\mathbf{x}} = \begin{bmatrix} \dot{x} \\ \ddot{x} \end{bmatrix} = \begin{bmatrix} x_2 \\ -2\zeta\omega_0\dot{x} - \omega_0^2 x + u_0\cos(\omega t) \end{bmatrix} = \begin{bmatrix} x_2 \\ -2\zeta\omega_0 x_2 - \omega_0^2 x_1 + u_0\cos(\omega t) \end{bmatrix}$$

The initial conditions are of course just:

$$\mathbf{x}_0 = \begin{bmatrix} x_0 \\ v_0 \end{bmatrix}.$$

This first order system can solved using scipy.integrate.solve_ivp. Here is how:

In [98]:
```
from scipy.integrate import solve_ivp

# You need to define the right hand side of the equation
def rhs(t, x, omega0, zeta, u0, omega):
    """Return the right hand side of the dynamical system.

    Arguments
    t       -    Time
    x       -    The state
    omega0  -    Natural frequency
    zeta    -    Dumping factor (0<=zeta)
    u0      -    External force amplitude
    omega   -    Excitation frequency
    """
    res = np.ndarray((2,))
    res[0] = x[1]
    res[1] = -2.0 * zeta * omega0 * x[1] - omega0 ** 2 * x[0] + u0 * np.cos(omega * t)
    return res
```

And here is how you solve it for given initial conditions and parameters:

In [99]:
```
# Initial conditions
x0 = np.array([0.0, 1.0])
# Natural frequency
omega0 = 2.0
# Dumping factor
zeta = 0.4
# External forcing amplitude
```

```
u0 = 0.5
# Excitation frequency
omega = 2.1
# Timestep
dt = 0.1
# The final time
final_time = 10.0
# The number of timesteps to get the final time
n_steps = int(final_time / dt)
# The times on which you want the solution
t_eval = np.linspace(0, final_time, n_steps)
# The solution
sol = solve_ivp(rhs, (0, final_time), x0, t_eval=t_eval, args=(omega0, zeta, u0, omega)
```

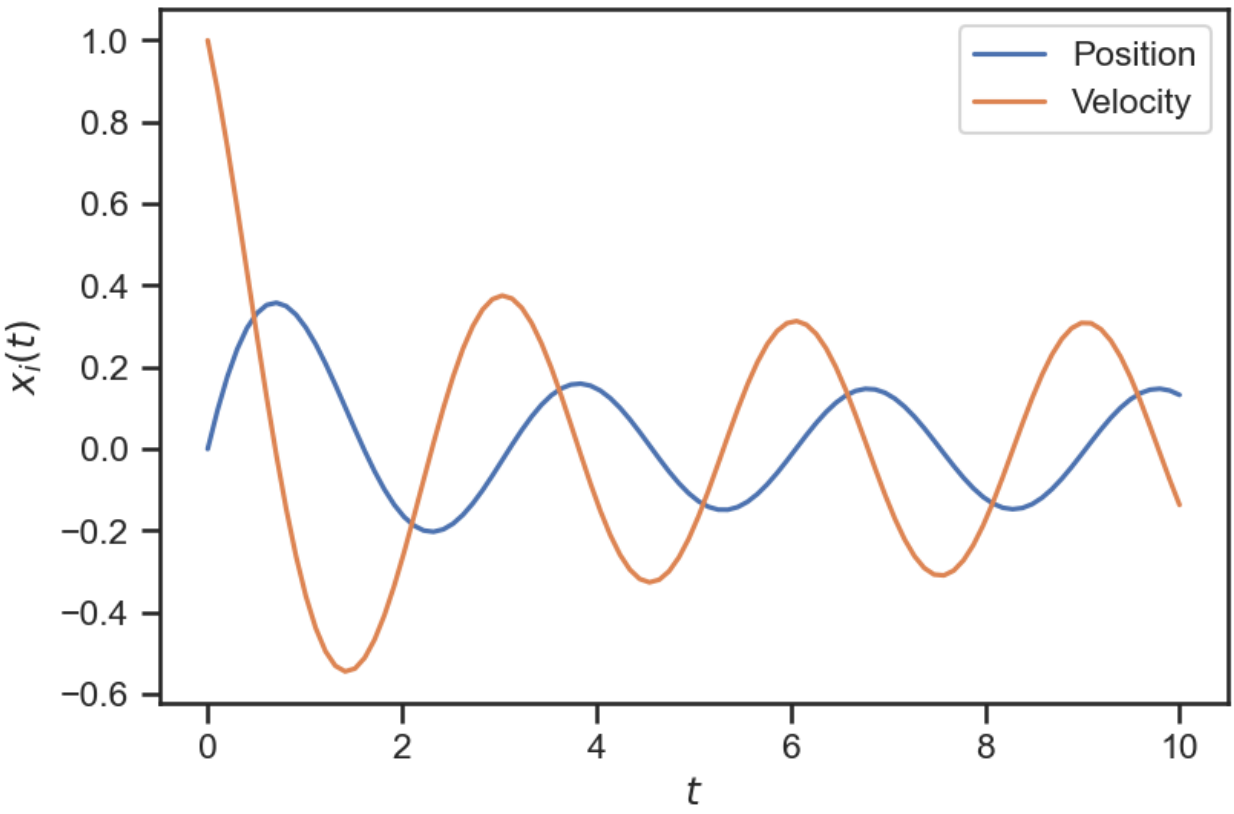
The solution is stored here:

In [100…
```
sol.y.shape
```

Out[100…
```
(2, 100)
```

You see that the shape is `number of states x number of time steps` . Let's visualize the trajectory separating position and velocity:

In [101…
```
fig, ax = plt.subplots(dpi=150)
ax.plot(t_eval, sol.y[0, :], label='Position')
ax.plot(t_eval, sol.y[1, :], label='Velocity')
ax.set_xlabel('$t$')
ax.set_ylabel('$x_i(t)$')
plt.legend(loc='best');
```



Let's now generate some synthetic observations of the acceleration with some given Gaussian noise. To get the true acceleration you can do this:

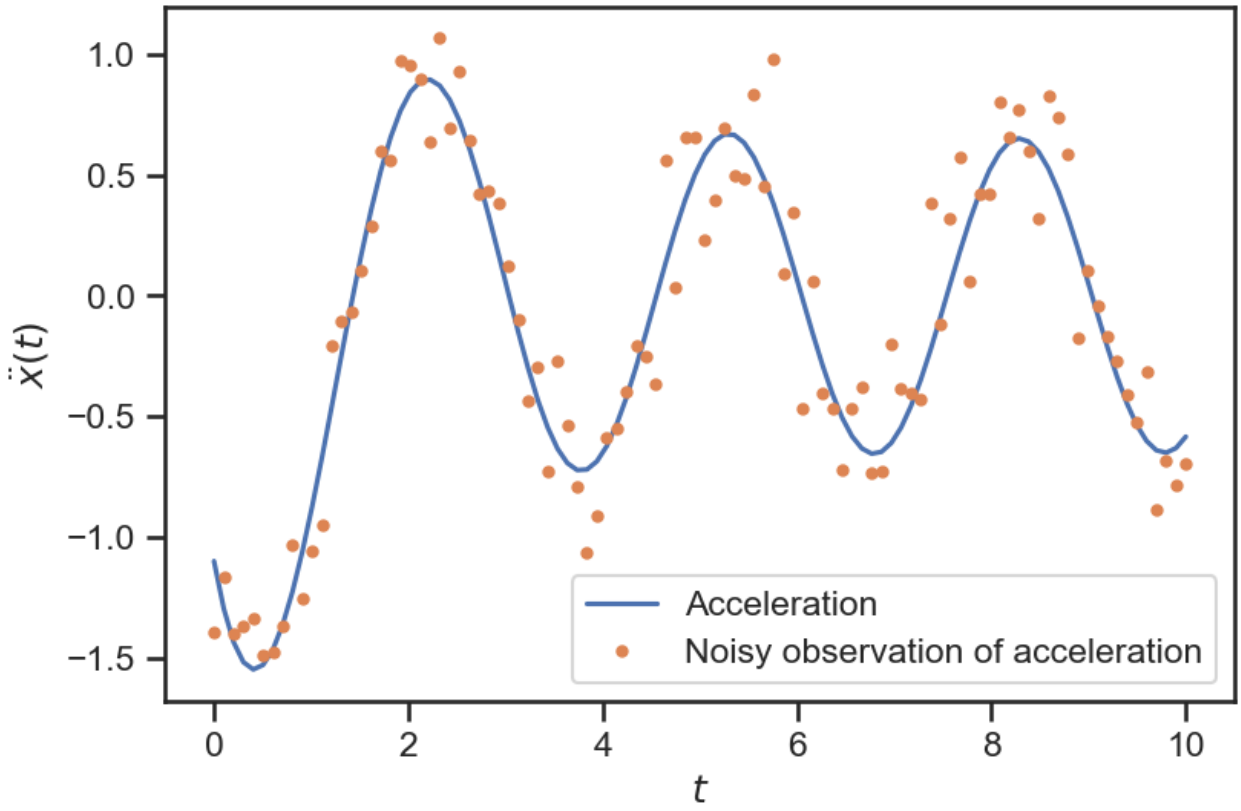
In [102…
```
true_acc = np.array([rhs(t, x, omega0, zeta, u0, omega)[1] for (t, x) in zip(t_eval, so
```

And now I am going to add some Gaussian noise to it:

In [103…
```
# The measurement standard deviation
sigma_r = 0.2
observations = true_acc + sigma_r * np.random.randn(true_acc.shape[0])
```

And here is how the noisy observations of the acceleration look like compared to the true acceleration value:

In [104…
```
fig, ax = plt.subplots(dpi=150)
ax.plot(t_eval, true_acc, label='Acceleration')
ax.plot(
    t_eval,
    observations,
    '.',
    label='Noisy observation of acceleration'
)
ax.set_xlabel('$t$')
ax.set_ylabel('$\ddot{x}(t)$')
plt.legend(loc='best');
```



Okay. Now imagine that you only see the noisy observations of the acceleration. The filtering goal is to recover the state of the underlying system (as well as its acceleration). I am going to guide you

through the steps you need to follow.

Part A - Discretize time (Transitions)

Use the Euler time discretization scheme to turn the continuous dynamical system into a discrete time dynamical system like this:

$$\mathbf{x}_{j+1} = \mathbf{A}\mathbf{x}_j + Bu_j + \mathbf{z}_j,$$

where

$$\mathbf{x}_j = \mathbf{x}(j\Delta t),$$
$$u_j = u(j\Delta t),$$

and $\mathbf{z}_j$ is properly chosen process noise term. You should derive and provide mathematical exprssions for the following:

- The 2×2 transition matrix $\mathbf{A}$.
- The 2×1 control "matrix" B.
- The process covariance $\mathbf{Q}$. For the process covariance, you may choose your own values by hand.

Answer:

$$\mathbf{x}_{j+1} = \begin{bmatrix} \mathbf{x}(n\Delta t) + \dot{x}\Delta t(j\Delta t) \\ \dot{\mathbf{x}}(n\Delta t) + \ddot{x}\Delta t(j\Delta t) \end{bmatrix} = \begin{bmatrix} \mathbf{x}(n\Delta t) + \dot{x}\Delta t(j\Delta t) \\ \dot{x}\Delta t(j\Delta t) + \Delta t(\zeta\omega_0\dot{x}(j\Delta t) - \omega_0^2 x(j\Delta t) + u_0 cos(\omega t) \end{bmatrix}$$

$$A = \begin{bmatrix} 1 & \Delta t \\ -\Delta t\omega_0^2 & 1 - 2\Delta t\zeta\omega_0 \end{bmatrix} * \begin{bmatrix} \dot{x}(j\Delta t) \\ \dot{x}(j\Delta t) \end{bmatrix}$$

$$B = \begin{bmatrix} 0 \\ u_0\Delta t \end{bmatrix} * [u(j\Delta t)]$$

with

$$u(j\Delta t) = cos(\omega t)$$

In [105…

```
# You should be using the parameters dt, omega0, zeta, etc.
# from above
A = np.array(
    [
        [1,dt],
        [-(omega**2) * dt,  1 + dt*zeta * -2 * omega0]
    ]
)
```

In [106…

```
B = np.array(
    [
        [0],
```

```
        [u0*dt]
    ]
)
```

In [107…

```
Q = np.array(
    [
        [1E-6, 0.0],
        [0.0, 1E-4]
    ]
)
```

Part B - Discretize time (Emissions)

Establish the map that takes you from the states to the accelerations at each timestep. That is, specify:

$$y_j = \mathbf{C}\mathbf{x}_j + w_j,$$

where

$$y_j = \ddot{x}(j\Delta t) - u_0 \cos(\omega t) + w_j,$$

and w_j is a measurement noise. You should derive and provide mathematical expressions for the following:

- The 1×2 emission matrix $\mathbf{C}$.
- The 1×1 covariance "matrix" R of the measurement noise.

Answer:

$$y_j = -2\zeta\omega_0\dot{x}(j\Delta t) + u_0 cos(\omega t) - u_0 cos(\omega t) + \omega_j,$$

$$y_j = -2\zeta\omega_0\dot{x}(j\Delta t) + \omega_j,$$

$$C_{xj} = \begin{bmatrix} -\omega_0^2 & -2\zeta\omega_0 \end{bmatrix} * \begin{bmatrix} x(j\Delta t) \\ \dot{x}(j\Delta t) \end{bmatrix}$$

$$R = x_j = |\sigma_r|$$

In [108…

```
#part B
C = np.array(
    [
        [-(omega0 ** 2),  -2* zeta  * omega0]
    ]
)

R = np.array(
    [
        [sigma_r**2]
    ]
)
```

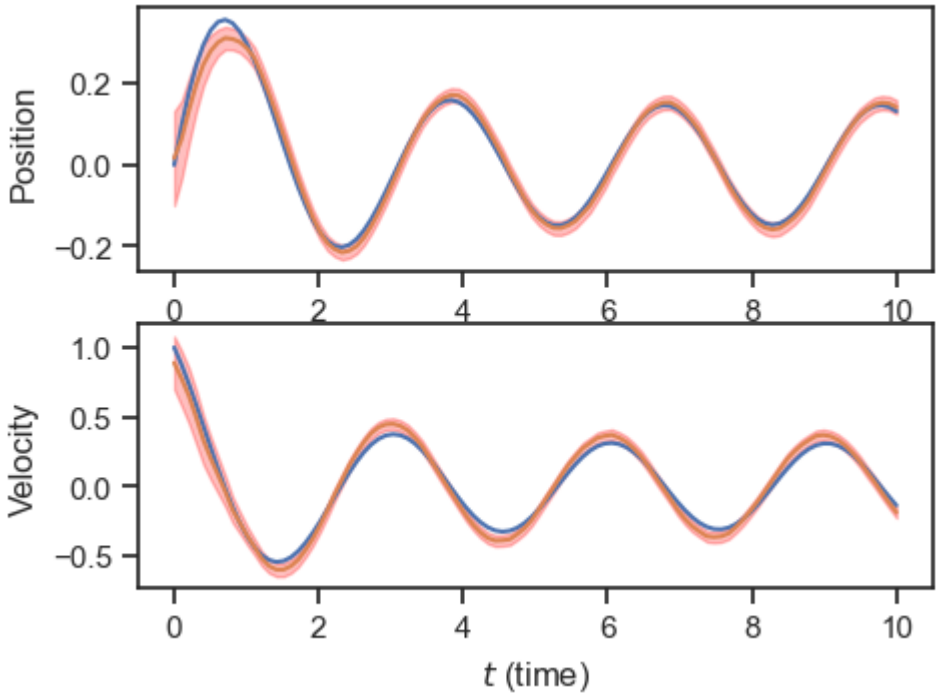
Part C - Apply the Kalman filter

Use `FilterPy` (see the hands-on activity of Lecture 20) to infer the unobserved states given the noisy observations of the accelerations. Plot time-evolving 95% credible intervals for the position and the velocity along with the true unobserved values of these quantities (in two separate plots).

In [109…

```
from filterpy.kalman import KalmanFilter
mu0 = x0 #just for my own confusion with the hands on and stack overflow documents :(
kf = KalmanFilter(dim_x=2, dim_z=1) #Forgot to change dim_z from hands on activity and
kf.x = mu0
V0 = np.array([0.1**2, 0.1**2]) * np.eye(2) # cov
kf.P = V0 #Unsure
kf.Q = Q
kf.R = R
kf.H = C
kf.F = A
kf.B = B


def plot_kf_estimates(means, covs):
    """Plot estimates of the state with 95% credible intervals."""
    y_labels = ['Position','Velocity']

    dpi = 200
    res_x = 1024
    res_y = 768
    w_in = res_x / dpi
    h_in = res_y / dpi

    fig, ax = plt.subplots(2, 1)
    fig.set_size_inches(w_in, h_in)


    for j in range(2):
        ax[j].set_ylabel(y_labels[j])
    ax[-1].set_xlabel('$t$ (time)')

    for j in range(2):
        ax[j].plot(
            t_eval,
            sol.y[j,:], # may need to be n,
            label = 'True Solution'
            #'b.-'
        )
        ax[j].plot(
            t_eval,
            means[:,j], # may need to be n,
            label = 'Mean'
            #'r.-'
        )
        ax[j].fill_between(
            t_eval,
            (
                means[:, j]
                - 2.0 * np.sqrt(covs[:, j, j])
            ),
            (
```

```
                means[:, j]
                + 2.0 * np.sqrt(covs[:, j, j])
            ),
            label = "95% Credible Region",
            color='red',
            alpha=0.25
        )


us = np.cos(omega*t_eval)[:,None]
kf.x = x0
kf.P = V0

means, cov, _, _ = kf.batch_filter(observations, us = us)

plot_kf_estimates(means, cov)
```



Part D - Quantify and visualize your uncertainty about the true acceleration value

Use standard uncertainty propagation techniques to quantify your epistemic uncertainty about the true acceleration value. You will have to use the inferred states of the system and the dynamical model. This can be done either analytically or by Monte Carlo. It's your choice. In any case, plot time-evolving 95% credible intervals for the acceleration (epistemic only) along with the true unobserved values and the noisy measurements.

In [110…

```
#part 3.D

guessVar = np.reshape((C @ cov[:] @ C.T), np.shape(t_eval))
guessMean = np.reshape((C @ means[:].T), np.shape(t_eval))

fig, ax = plt.subplots(dpi=200)
```

```
ax.plot(t_eval, true_acc, label='True')
ax.scatter(t_eval, observations, label='Noisy Observations', color = "orange", s =5)
ax.plot(t_eval,guessMean, label='Inferred States')
ax.fill_between(
    t_eval,
    (
        guessMean
        - 2.0 * np.sqrt(guessVar)
    ),
    (
        guessMean
        + 2.0 * np.sqrt(guessVar)
    ),
    label = "95% Credible Region",
    color='red',
    alpha=0.25
)
ax.set_xlabel('Time')
ax.set_ylabel('Acceleration')
plt.title('Problem 3.d: Uncertainty about True Acceleration')
plt.legend(loc='best');

print('There appears to be a time shift between the inferred and true. This should occu
```

```
There appears to be a time shift between the inferred and true. This should occur in Kal
man filtering of data that is not time independent correct?
```