ME539 Homework 5

July 19, 2022

1 Homework 5

1.1 References

- Lectures 17-20 (inclusive).

1.2 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.

[1]:
```
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

from matplotlib import rc
rc('text', usetex=True)

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)
```

[2]:
```
def perform_plot(
    x,
    y,
    color,
    box,
    title,
    img=None,
    X=None,
    Y=None,
    Z=None,
    levels=None,
    alpha = 0.5,
    s = 0.5
):
    """Perform plotting routine for visualizing Uber pickup data

    Arguments
    x              -- The x-coordinate of the provided location data
    y              -- The y-coordinate of the provided location data
    color          -- Color of points to use when plotting
    box            -- The bounding box used to specify bounds for x and y
    title          -- Title to use for generated plot

    Keyword Arguments
    img            -- The image to display on the plot, if provided
    X              -- The x-coordinate of a meshed grid of points
    Y              -- The y-coordinate of a meshed grid of points
    Z              -- The reshaped scored samples from a model object
    levels         -- The levels to use for contour plotting
    alpha          -- The alpha value for plotted data points, to override the␣
 ↪default
    s              -- The marker size for plotted data points, to override the␣
 ↪default
    """
    # generate plotting object
    fig, ax = plt.subplots(dpi=600)
    # if an X (assuming Y, Z, and levels are as well) is provided, then plot␣
 ↪the contours
    if X is not None:
```

```
        c = ax.contour(X, Y, Z, levels)
        plt.colorbar(c)
    # plot x and y location data points, with specified arguments
    ax.scatter(
        x, y,
        zorder=1,
        alpha=alpha,
        c=color,
        s=s
    )
    # set limits of ploting object
    ax.set_xlim(box[0],box[1])
    ax.set_ylim(box[2],box[3])
    # if an image is provided, display it
    if img is not None:
        ax.imshow(
            img,
            zorder=0,
            extent=box,
            aspect= 'equal'
        )
    # set label and title
    ax.set_xlabel('Longitude')
    ax.set_ylabel('Latitude')
    ax.set_title(title);
```

1.3 Student details

- **First Name:** Jack
- **Last Name:** Girard
- **Email:** girard2@purdue.edu

2 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:

[3]:
```
url = "https://github.com/PredictiveScienceLab/data-analytics-se/raw/master/
 ↪lecturebook/data/uber-raw-data-apr14.csv"
download(url)
```

And you can load it using pandas:

[4]:
```
import pandas as pd
p1_data = pd.read_csv('uber-raw-data-apr14.csv')
```

Here is a text view:

[5]:
```
print(p1_data)
```

```
                  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 x 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:

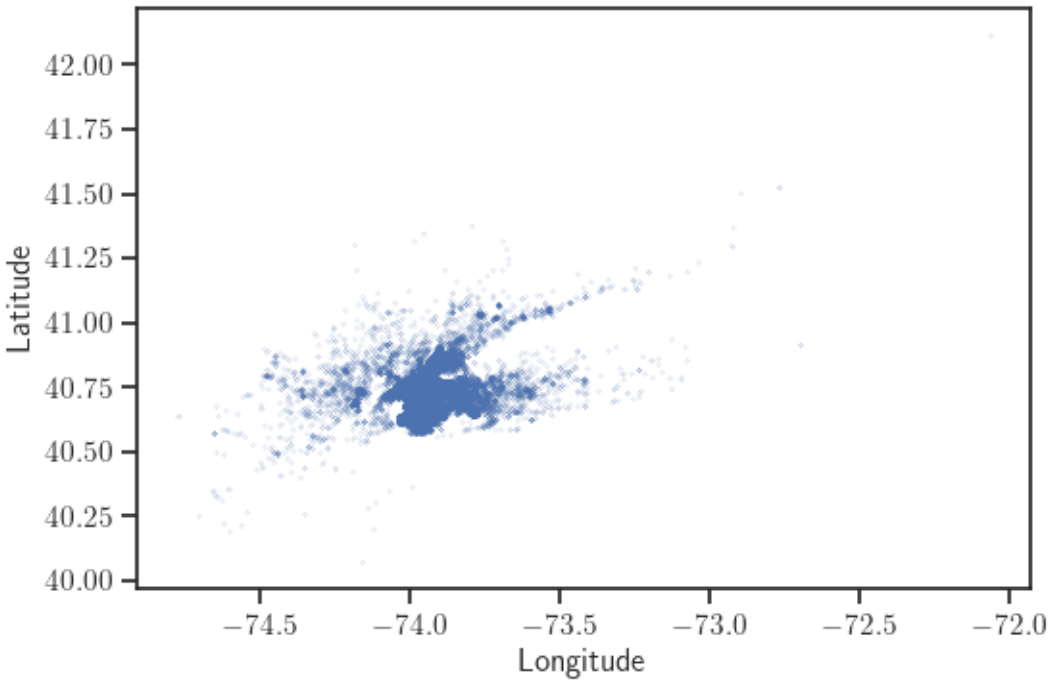
[6]:
```
# Just use the column names as indices.
# The two brackets are required because you are actually
# passing a list of columns
loc_data = p1_data[['Lon', 'Lat']]
print(loc_data)
```

```
            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 x 2 columns]
```

Let's also visualize these points:

[7]:
```
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*:

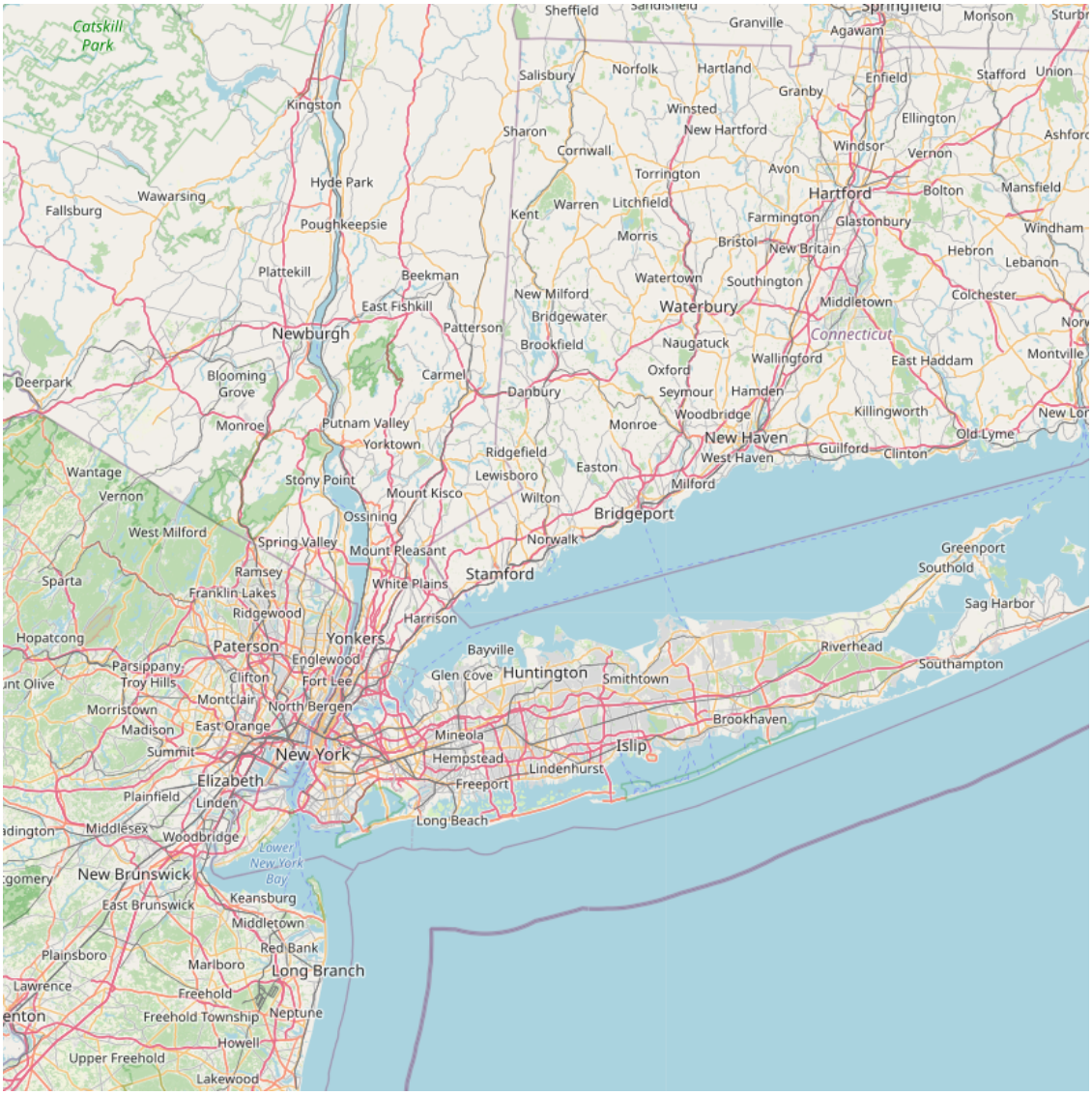
[8]:
```
box = ((loc_data.Lon.min(), loc_data.Lon.max(),
        loc_data.Lat.min(), loc_data.Lat.max()))
print(box)
```

```
(-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:

[9]:
```
url = "https://github.com/PredictiveScienceLab/data-analytics-se/raw/master/
  ↪lecturebook/images/ny_map.png"
download(url)
```

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



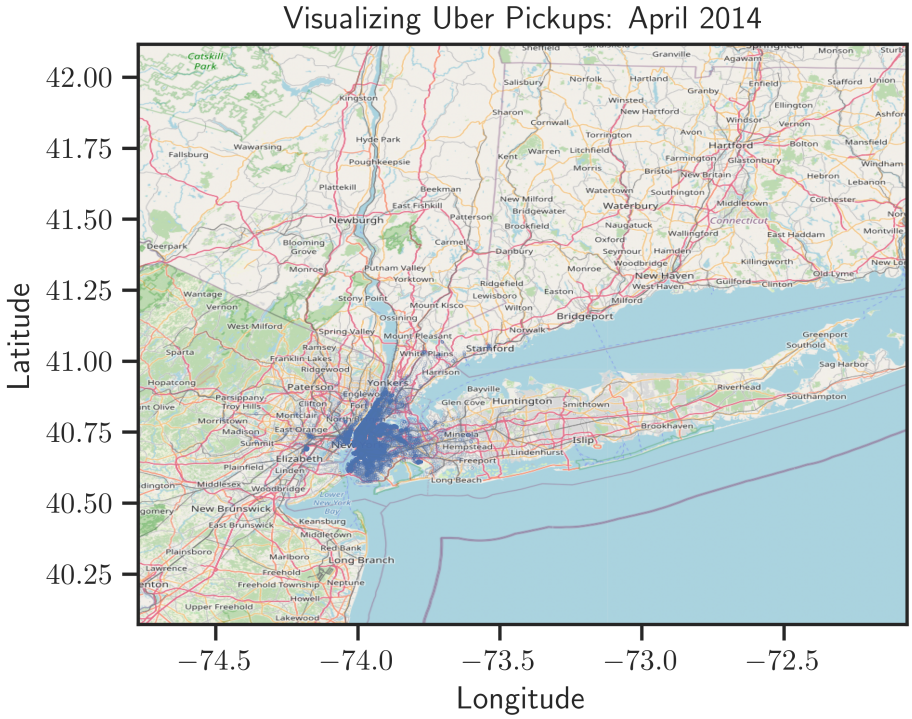
Now let's load the image as a matrix:

[10]:
```
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:

[11]:
```
# Perform plot using function
perform_plot(
    loc_data.Lon,
    loc_data.Lat,
    'b',
    box,
    "Visualizing Uber Pickups: April 2014",
    ny_map,
    alpha = 0.5,
```

```
    s = 0.001
)
```



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.

[12]:
```
# 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
```

2.1 Part A - Splitting New York City into Subregions

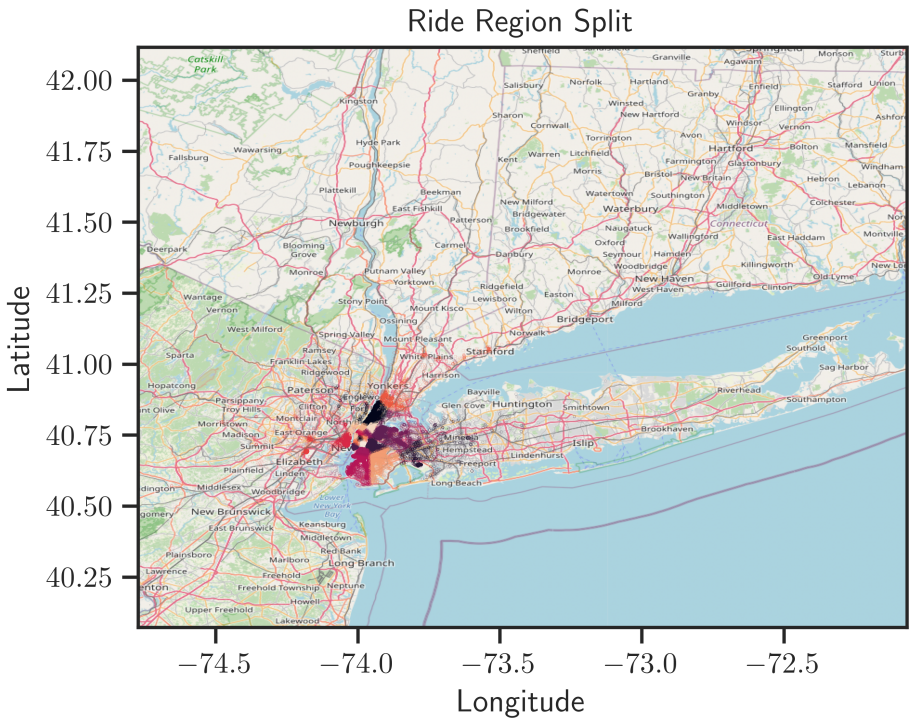
Suppose that you are assigned the task of splitting New York City into operating subregions with pretty much equal demand. When a pickup is requested in each subregion only the drivers in that region are called. Note that this can quickly become a very difficult problem very quickly. We are not looking 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

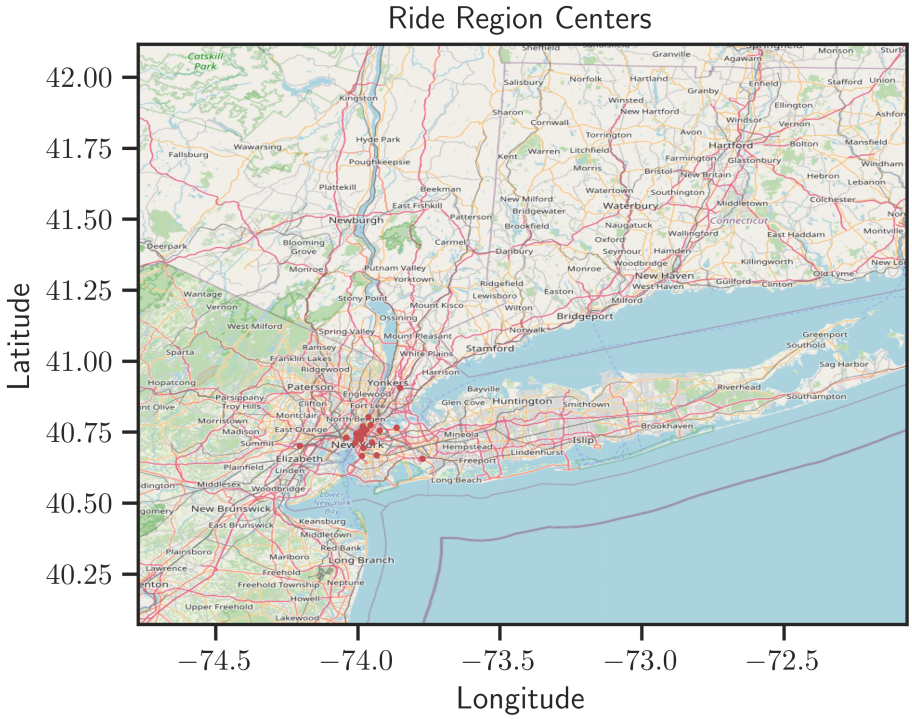
(even if it is possible the drivers may have to pay tolls to cross). If it is impossible to get perfect results simply by Kmeans, feel free to ignore a small number of outliers as they could be handled manually; + Use MiniBatchKMeans which is an much faster version of Kmeans suitable for large datasets (>10K observations);

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

[13]:
```
# read in necessary function from sklearn
from sklearn.cluster import MiniBatchKMeans
# define number of clusters to use
numClust = 21
# fit model
model = MiniBatchKMeans(n_clusters=numClust).fit(p1_train_data)
# make prediction using model
labels = model.predict(p1_train_data)
# plot results using function
perform_plot(
    p1_train_data.Lon,
    p1_train_data.Lat,
    labels,
    box,
    "Ride Region Split",
    ny_map,
    s = 0.001
)
# reference(s): hands-on activity 17.1
```



[14]:
```
# extract cluster centers
clustCenters = model.cluster_centers_
xCenters = [loc[0] for loc in clustCenters]
yCenters = [loc[1] for loc in clustCenters]
# plot results using function
perform_plot(
    xCenters,
    yCenters,
    'r',
    box,
    "Ride Region Centers",
    ny_map,
    alpha = 1,
    s = 2
)
# reference(s): hands-on activity 17.1
```



Part A Discussion:

The number of clusters was selected to be 21. This number appears to split the New York City metropolitan area into small operating sub-regions which allows for minimal tolls and more frequent ride pick-ups and drop-offs. By visual inspection, it appears that very few of the sub-regions cross

water, if any.

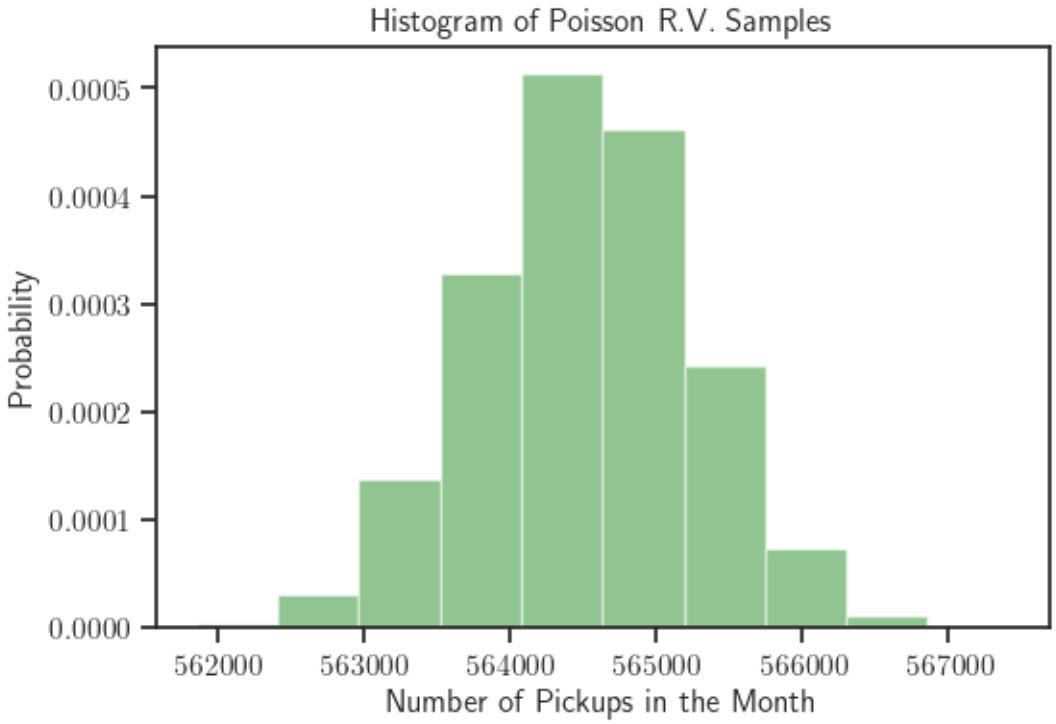
2.2 Part B - Create a Stochastic Model of Pickups

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.

2.2.1 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.

[15]:
```
# using the monthly pickup rate from April 2014 data
lambda_pickup = p1_train_data.shape[0]
# define Poisson random variable
numPickups = st.poisson(lambda_pickup)
# define number of samples to take
num_samples = 10000
# take samples from distribution
samples = numPickups.rvs(num_samples)
# plot histogram
fig, ax = plt.subplots()
ax.hist(samples, alpha=0.5, color='forestgreen', density=True)
ax.set_xlabel('Number of Pickups in the Month')
ax.set_ylabel('Probability')
ax.set_title('Histogram of Poisson R.V. Samples');
```



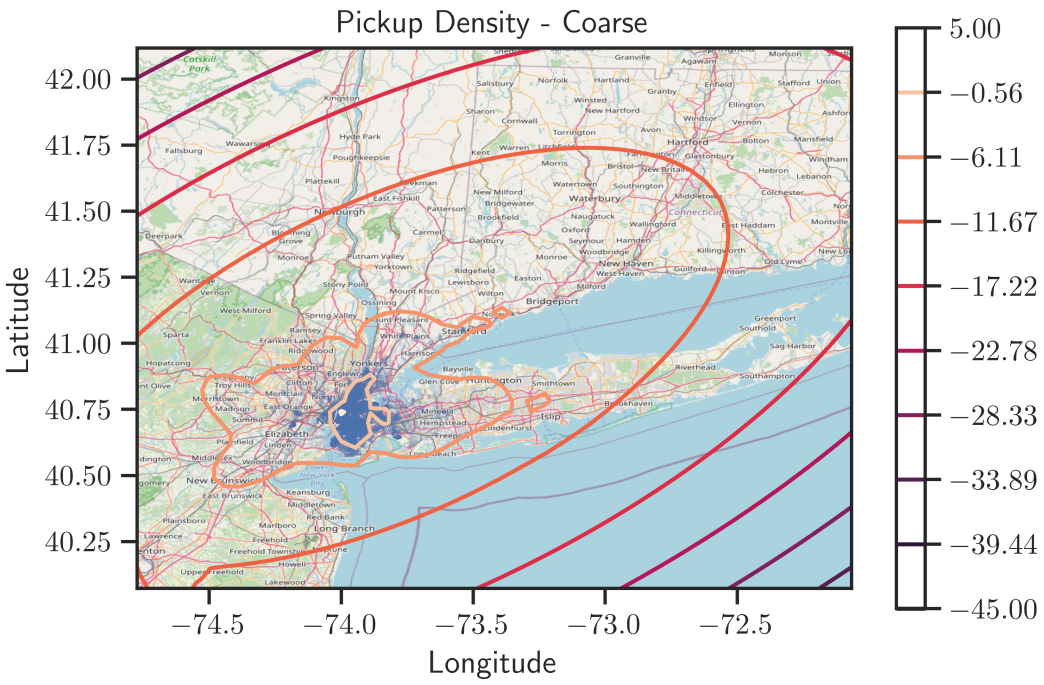
2.2.2 Subpart B.II - Estimate the spatial density of pickups

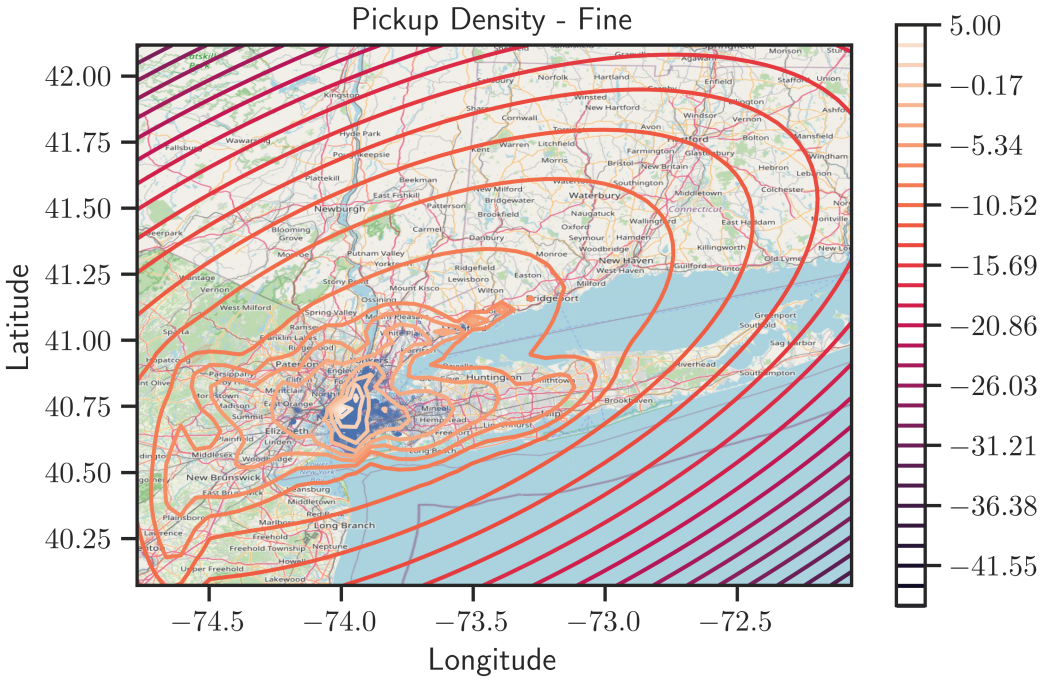
Fit a Gaussian Mixture model to the pickup data. **Do not use the Bayesian Information Criterion** to decide how many components to keep. This would take quite a bit of time for this problem. Simply use 40 mixture components. Plot the contour of the logarithm of the probability density on the New York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

[16]:
```
# read in necessary function from sklearn
from sklearn.mixture import GaussianMixture
# define number of mixture components
num_components = 40
# cast the training data DataFrame to numpy array, for fitting the model
p1_train_data_nparray = p1_train_data.to_numpy()
# fit model
model = GaussianMixture(n_components=num_components).fit(p1_train_data_nparray)
# reference(s): hands-on activity 17.2
```

[17]:
```
# make grid
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 = model.score_samples(XX)
```

```
Z = z.reshape(X.shape)
# number of levels for coarse contour result
levels=np.linspace(-45, 5, 10)
# plot results & coarse contours using function
perform_plot(
    p1_train_data.Lon,
    p1_train_data.Lat,
    'b',
    box,
    "Pickup Density - Coarse",
    ny_map,
    X, Y, Z, levels,
    s = 0.001
)
# number of levels for fine contour result
levels=np.linspace(-45, 5, 30)
# plot results & coarse contours using function
perform_plot(
    p1_train_data.Lon,
    p1_train_data.Lat,
    'b',
    box,
    "Pickup Density - Fine",
    ny_map,
    X, Y, Z, levels,
    s = 0.001
)
# reference(s): hands-on activity 17.2
```


Pickup Density - Coarse
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
5.00
−0.56
−6.11
−11.67
−17.22
−22.78
−28.33
−33.89
−39.44
−45.00

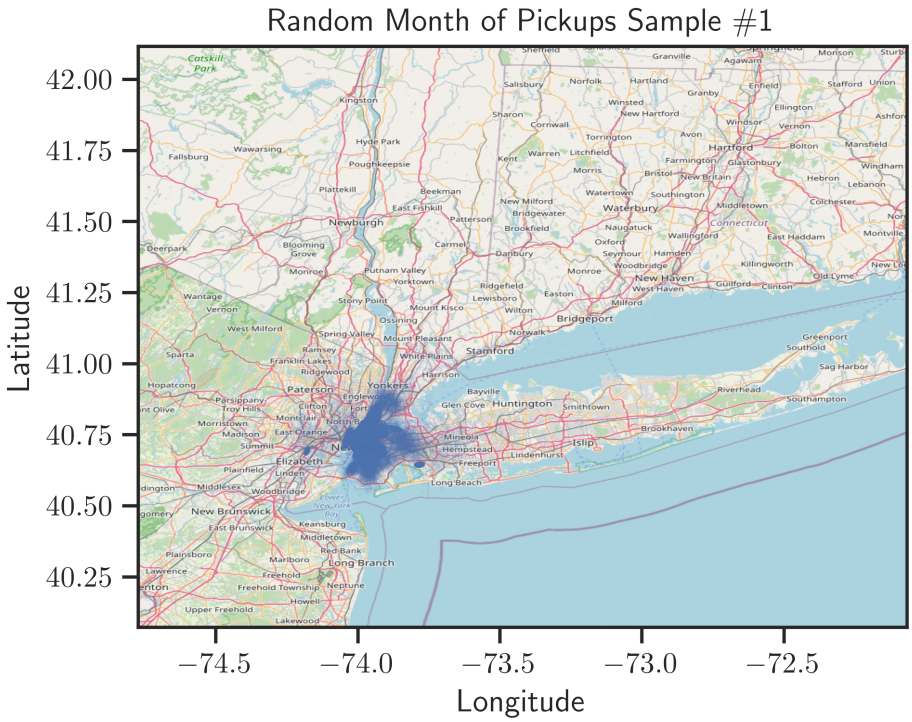

Pickup Density - Fine
Latitude
42.00
41.75
41.50
41.25
41.00
40.75
40.50
40.25
−74.5
−74.0
−73.5
−73.0
−72.5
Longitude
5.00
−0.17
−5.34
−10.52
−15.69
−20.86
−26.03
−31.21
−36.38
−41.55

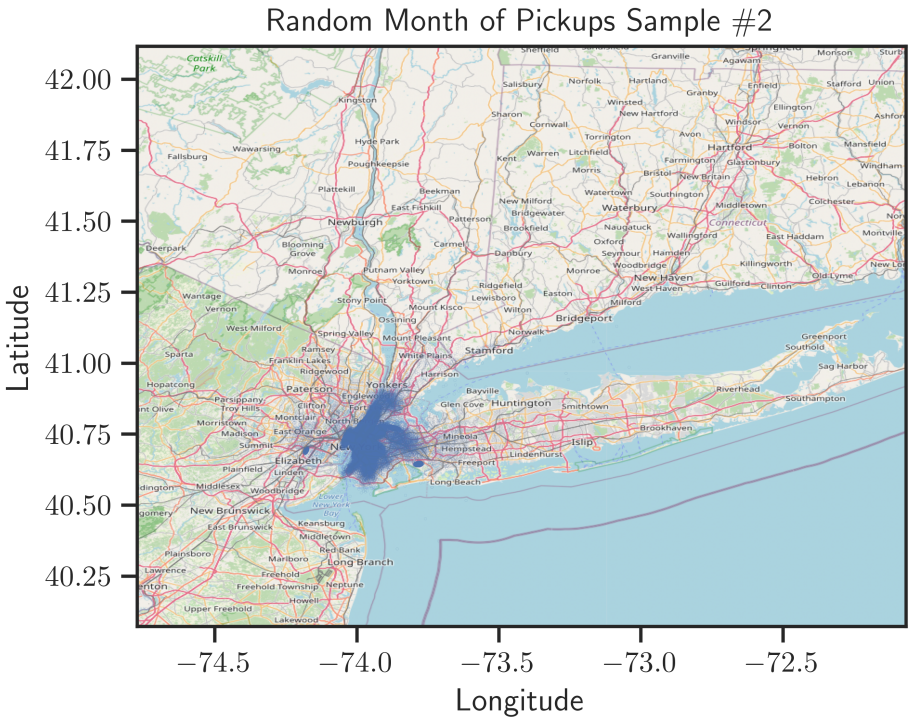
2.2.3 Subpart B.III - Sample some random months of pickups

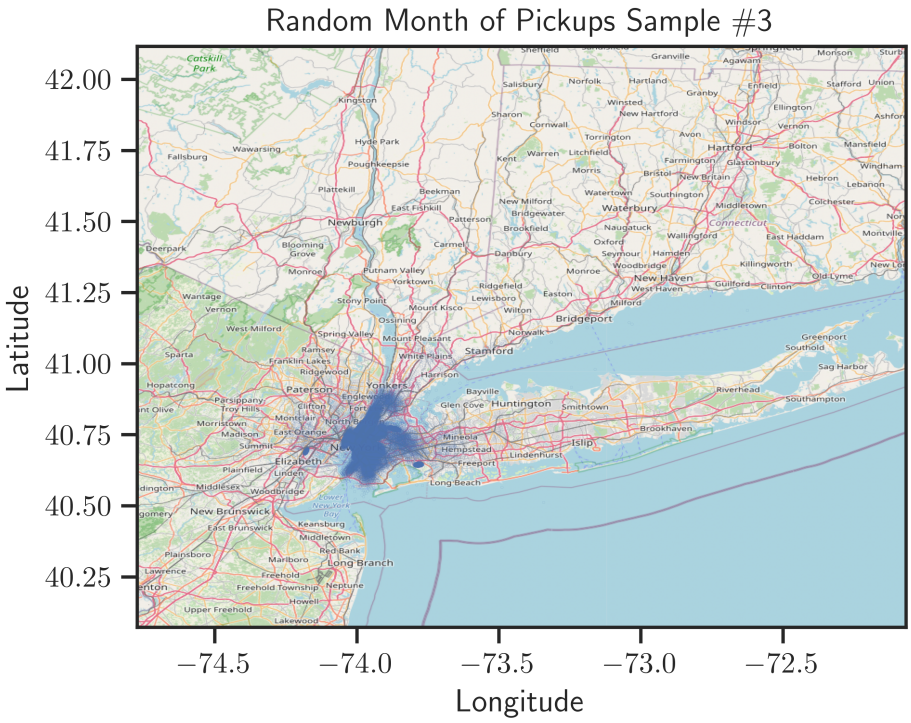
Now that you have a model that gives you the number of pickups and a model that allows you to sample a pickup location, sample five different datasets (number of pickups and location of each pick) from the combined model and visualize them on the New York map.

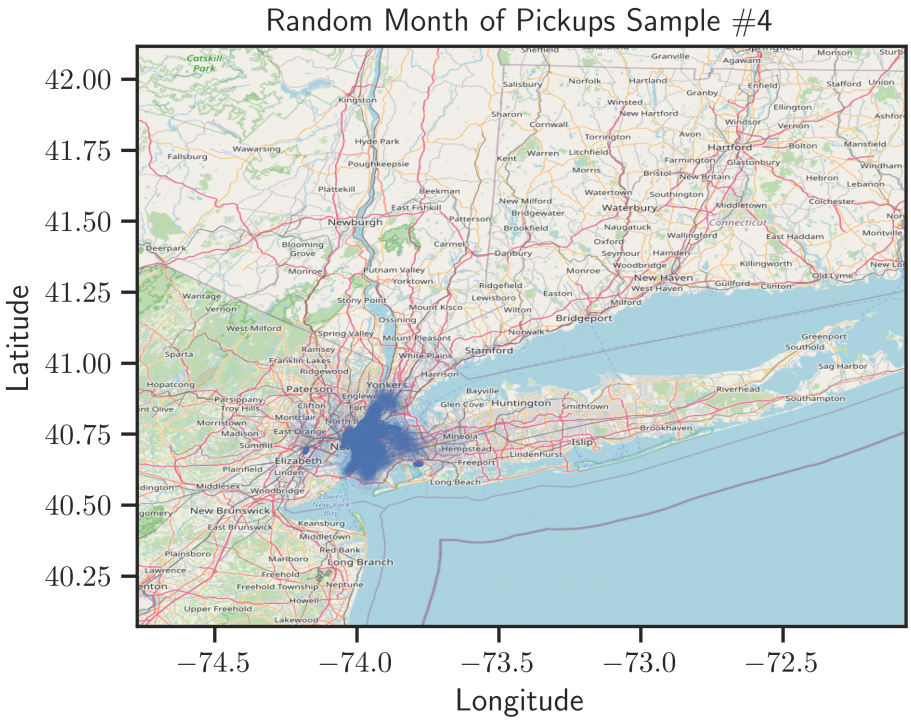
Hint: Don't get obsessed with making the model perfect. It's okay if a few of the pickups are on water...

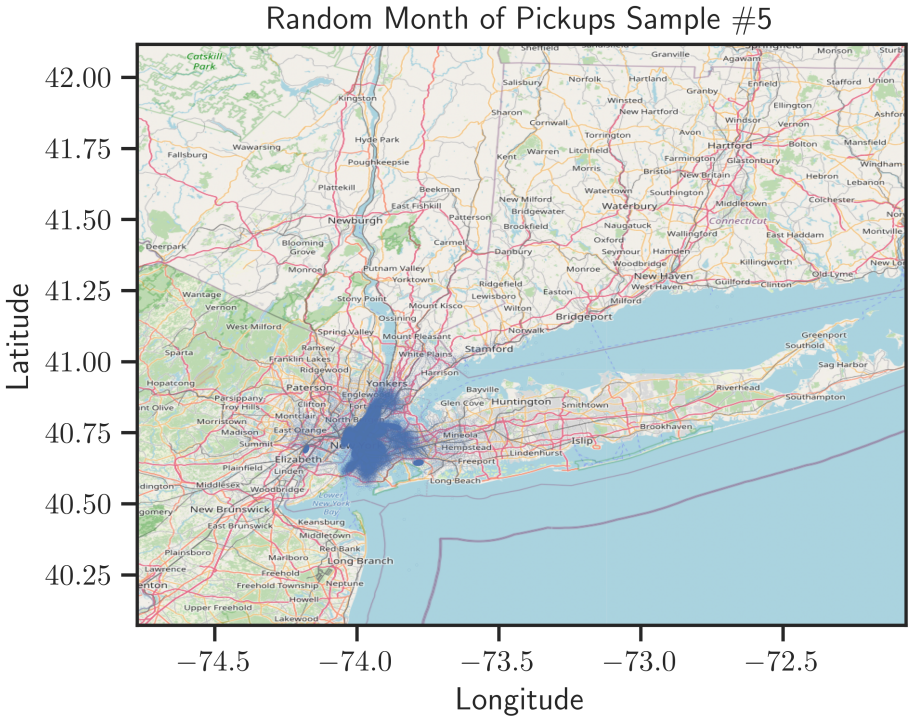
[18]:
```
# define the number of sample datasets to take
num_datasets = 5
# for each dataset:
for i in range(num_datasets):
    # sample r.v. above for number of pickups in the month
    num_pickups_sample = numPickups.rvs()
    # sample from the Gaussian Mixture model using the above r.v.
    s_data, labels = model.sample(num_pickups_sample)
    # plot results using function
    perform_plot(
        s_data[:, 0],
        s_data[:, 1],
        'b',
        box,
        "Random Month of Pickups Sample \#" + str(i+1),
        ny_map,
        s = 0.001
    )
```




Random Month 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


Random Month of Pickups Sample #3
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

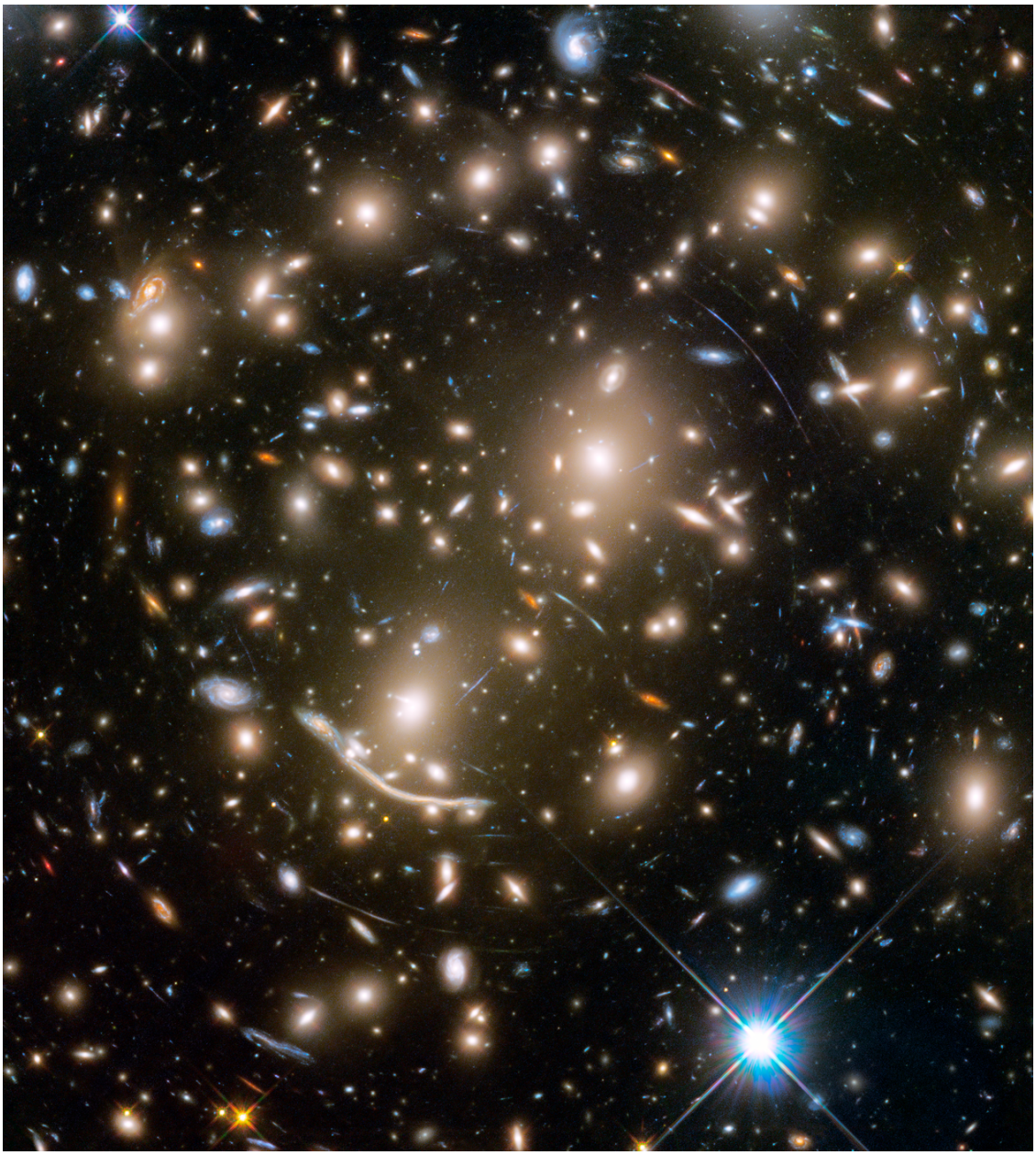

Random Month of Pickups Sample #4
Latitude
42.00
41.75
41.50
41.25
41.00
40.75
40.50
40.25
−74.5
−74.0
−73.5
−73.0
−72.5
Longitude


Random Month of Pickups Sample #5
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

3 Problem 2 - Counting Celestial Objects

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

[19]:
```
url = 'https://raw.githubusercontent.com/PredictiveScienceLab/data-analytics-se/
↪master/lecturebook/images/galaxies.png'
download(url)
```



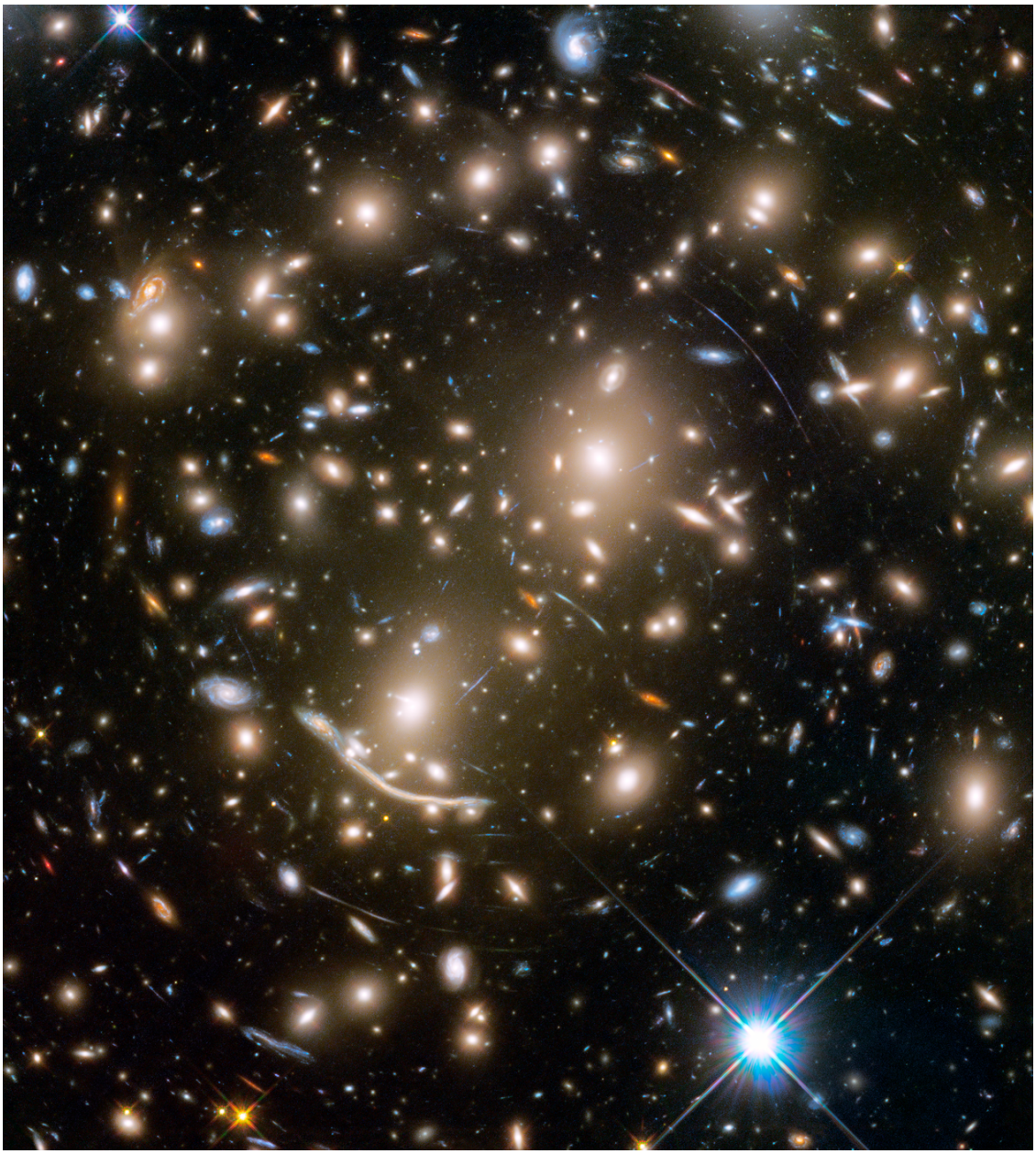
This picture includes many galaxies, but also some starts. We are going to create a machine learning model capable of counting the number of objects in such images. Our model is not going to be able to differentiate between the different types of objects, and it will not be very accurate, but it does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

[20]:
```
from PIL import Image
hubble_image = Image.open('galaxies.png')
# here is how to see the image
hubble_image
```

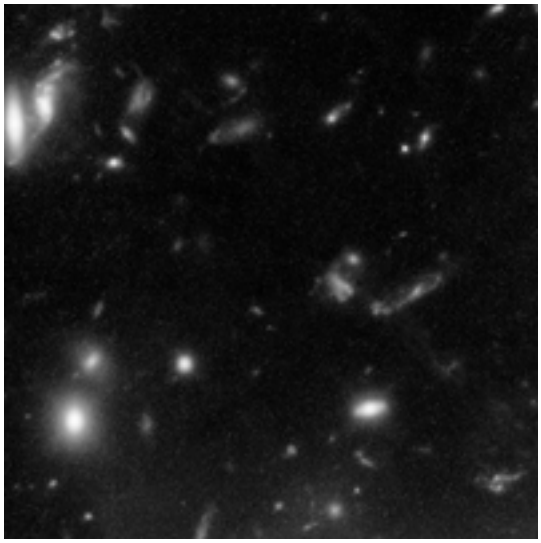
[20]:



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

[21]:
```
img = hubble_image.convert('L').crop((100, 100, 300, 300))
img
```

[21]:



Remember that black-and white images are matrices:

[22]:
```
img_ar = np.array(img)
print(img_ar)
```

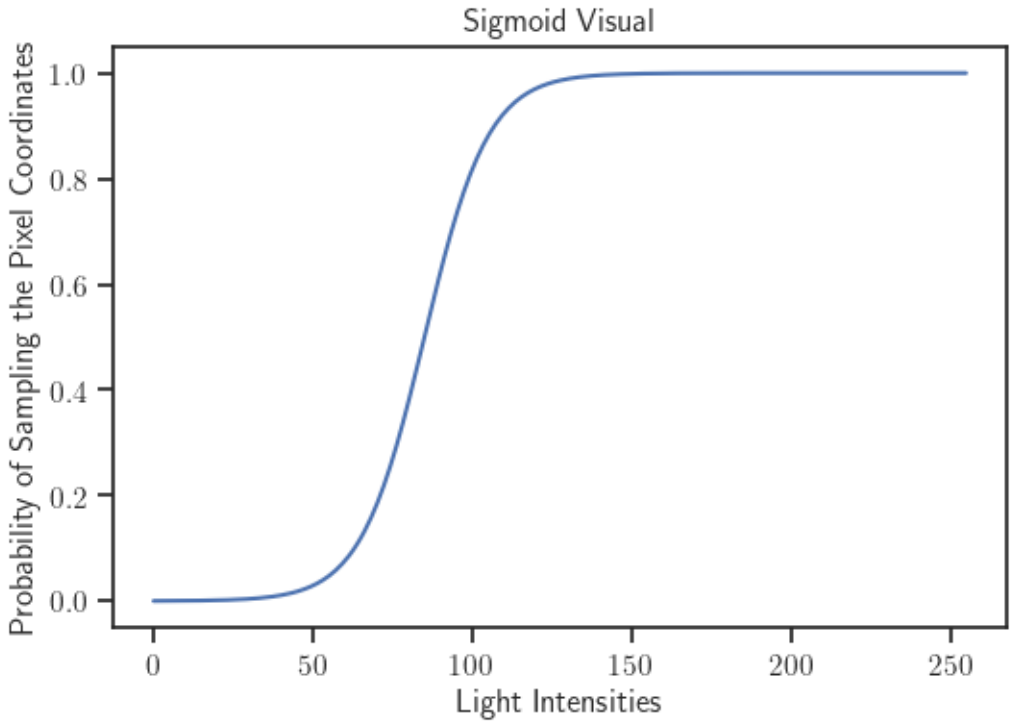
```
[[ 7 11 11 … 28 62 88]
 [12 12 11 … 29 47 86]
 [18 13 11 … 24 34 87]
 …
 [24 12 15 … 43 47 40]
 [23 12 19 … 48 49 40]
 [18 18 23 … 50 49 41]]
```

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:

[23]:
```
intensities = np.linspace(0, 255, 255)
fig, ax = plt.subplots()
alpha = 0.1
beta = 255 / 3
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')
ax.set_title('Sigmoid Visual');
```



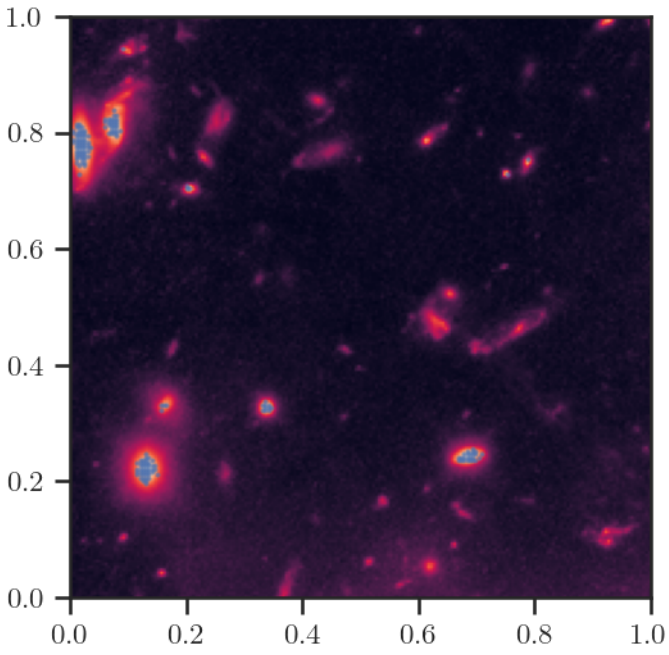
And here is the code that samples the pixel coordinates. I am organizing it into a function because we may want to use it with different pictures:

[24]:
```
def sample_pixel_coords(img, alpha, beta):
    """
    Samples pixel coordinates based on a probability defined as the sigmoid of␣
 ↪the intensity.

    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:

[25]:
```
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.

[26]:
```
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)
    # initialize variable arrays
    # note: the +1 in the bics initialization and following for loop are␣
 ↪necessary in order to include both the max and min number of components␣
 ↪inclusively)
    ncs = []
    models = []
    bics = np.ndarray((nc_max - nc_min + 1, ))
    # loop through number of components, generating a new model and appending␣
 ↪values to respective arrays each time
    for nc in range(nc_min, nc_max+1):
        m = GaussianMixture(n_components=nc).fit(locs)
        bics[nc - nc_min] = m.bic(locs)
        models.append(m)
        ncs.append(nc)
    # returning the number of components and model that correspont to the␣
 ↪minimum BIC score
    best_nc = ncs[np.argmin(bics)]
    best_model = models[np.argmin(bics)]
    # personal addition: plot bics
    fig, ax = plt.subplots()
    ax.bar(range(nc_min, nc_max+1), bics)
    ax.set_title("BIC Score Visual - Best \#: " + str(best_nc) + " Components")
    ax.set_ylabel('BIC Score')
    ax.set_xlabel('Number of Components');
    return best_nc, best_model, locs
```

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.**

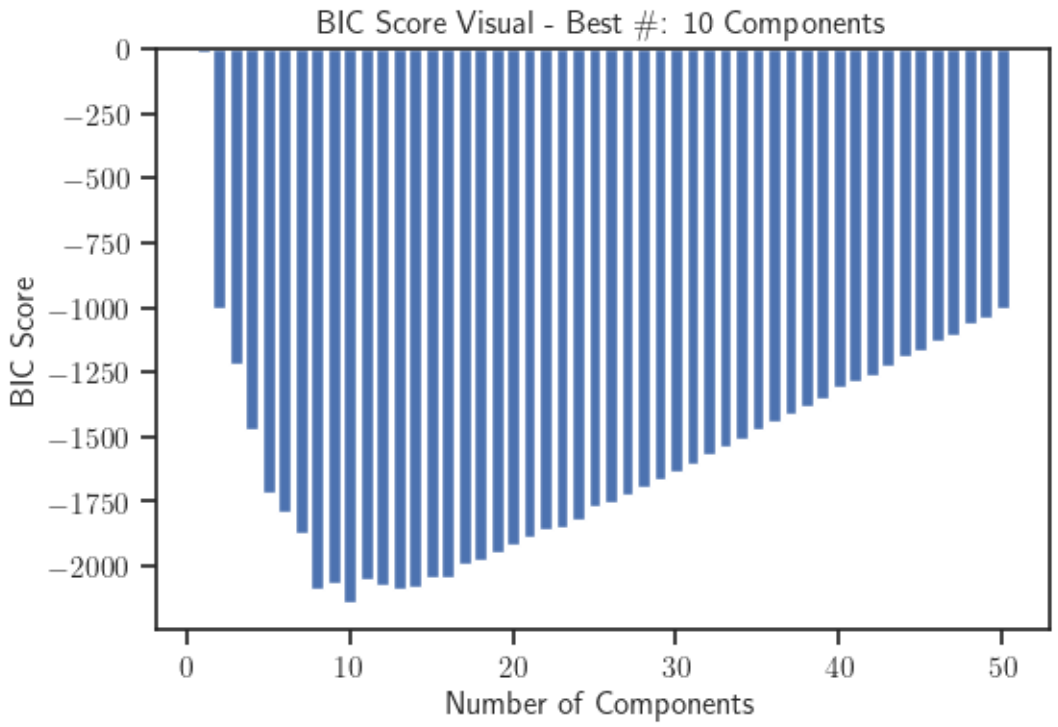
Student Note: the parameters α and β were adjusted slightly from the default values to improve the model marginally, but it is not perfect.

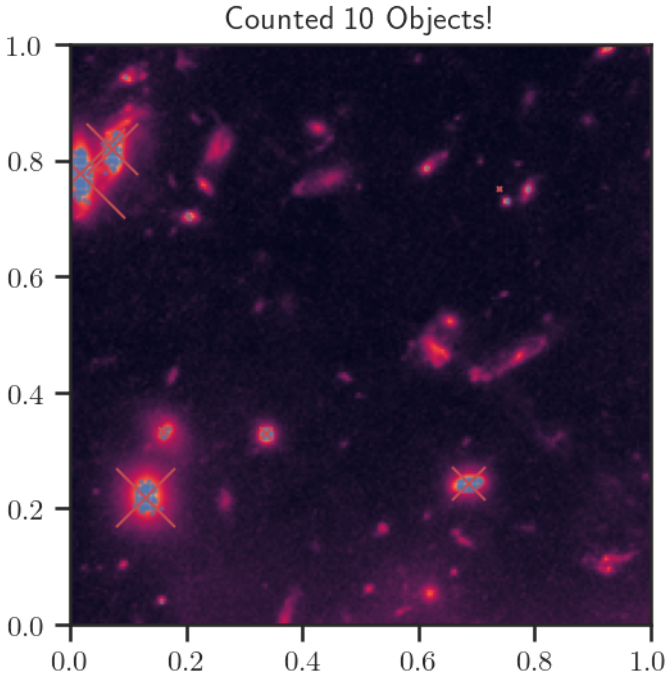
Here is a helper function for visualizing what you get:

[27]:
```
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:

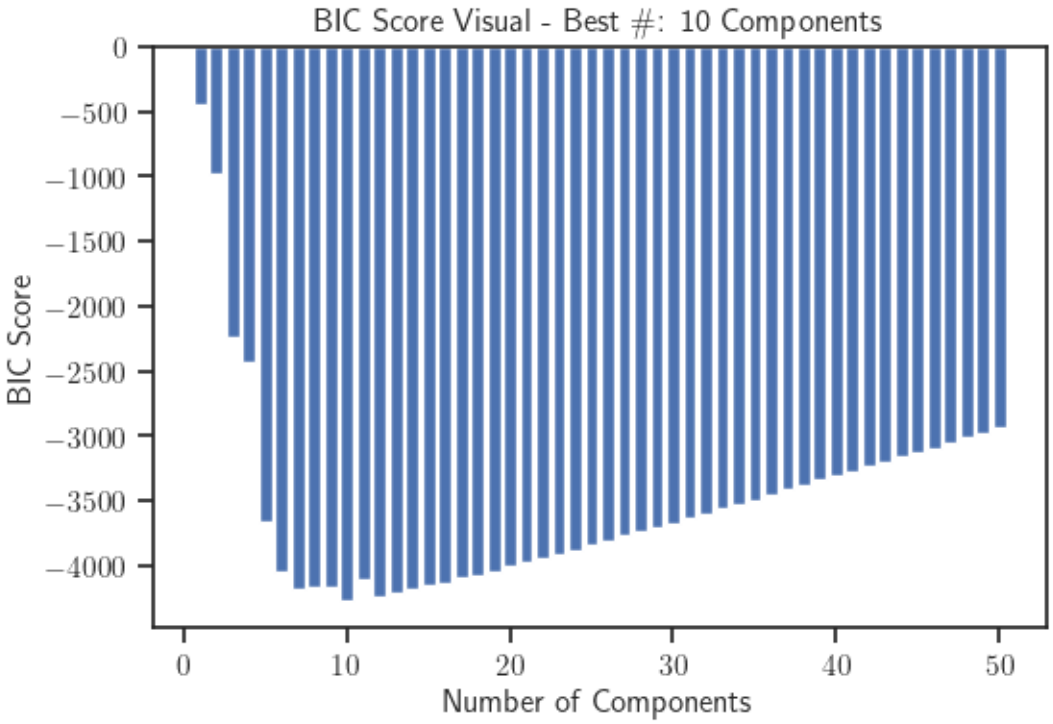
[28]:
```
objs, model, locs = count_objs(img, alpha=0.1, beta=180)
visualize_counts(img, objs, model, locs)
```

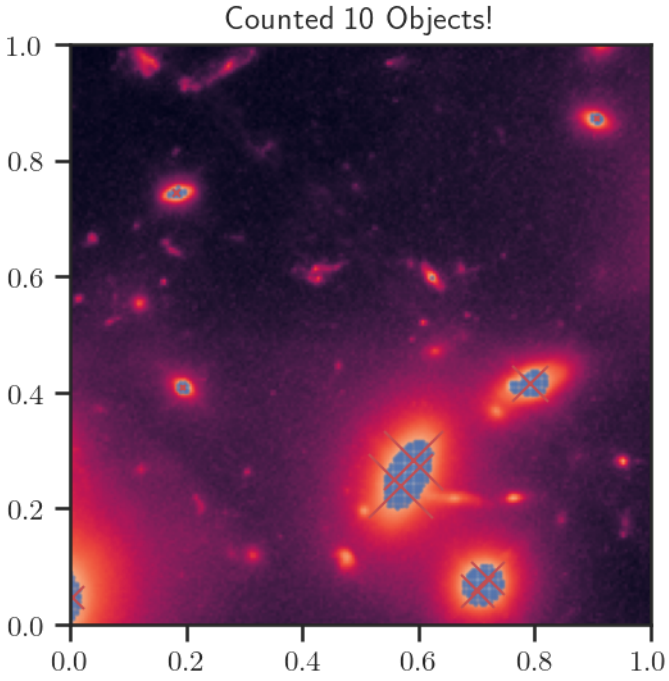

BIC Score Visual - Best #: 10 Components
BIC Score
0
−250
−500
−750
−1000
−1250
−1500
−1750
−2000
0
10
20
30
40
50
Number of Components


Counted 10 Objects!
1.0
0.8
0.6
0.4
0.2
0.0
0.0
0.2
0.4
0.6
0.8
1.0

Try this one:

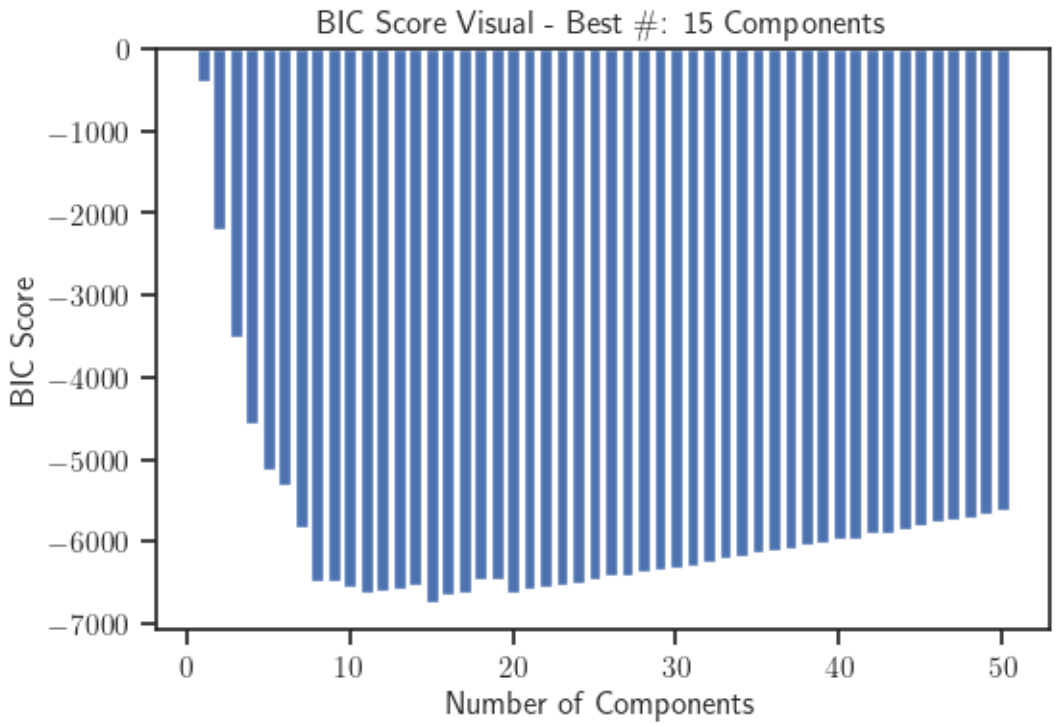
[29]:
```
img = hubble_image.convert('L').crop((200, 200, 400, 400))
objs, model, locs = count_objs(img, alpha=0.8, beta=210)
visualize_counts(img, objs, model, locs)
```

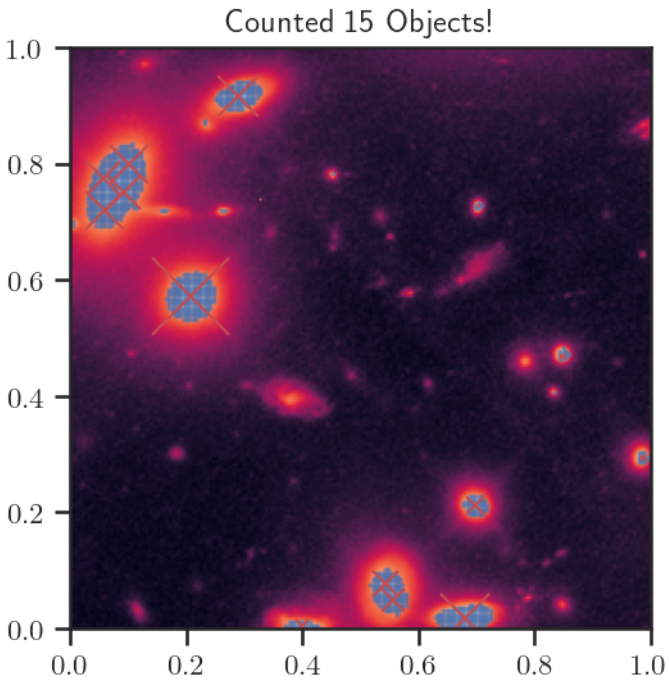




And try this one:

[30]:
```
img = hubble_image.convert('L').crop((300, 300, 500, 500))
objs, model, locs = count_objs(img, alpha=0.8, beta=190)
visualize_counts(img, objs, model, locs)
```





4 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:

[31]:
```
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:

[32]:
```
# 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))
```

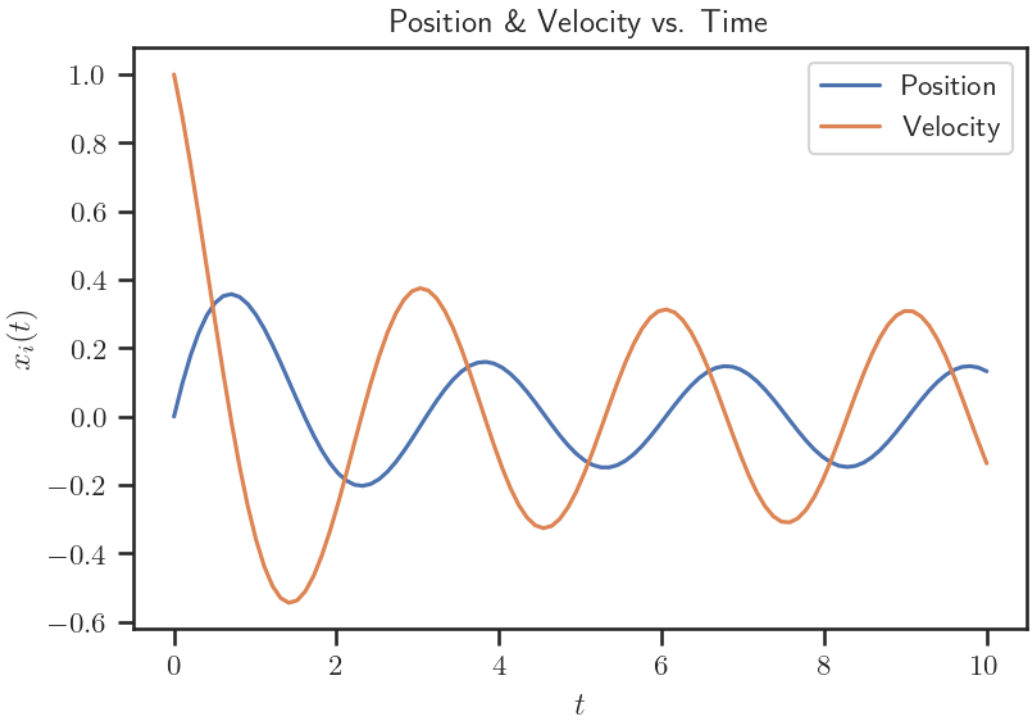
The solution is stored here:

[33]:
```
print(sol.y.shape)
```

```
(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:

[34]:
```
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')
ax.set_title("Position \& Velocity vs. Time");
```



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

[35]:
```
true_acc = np.array([rhs(t, x, omega0, zeta, u0, omega)[1] for (t, x) in␣
 ↪zip(t_eval, sol.y.T)])
```

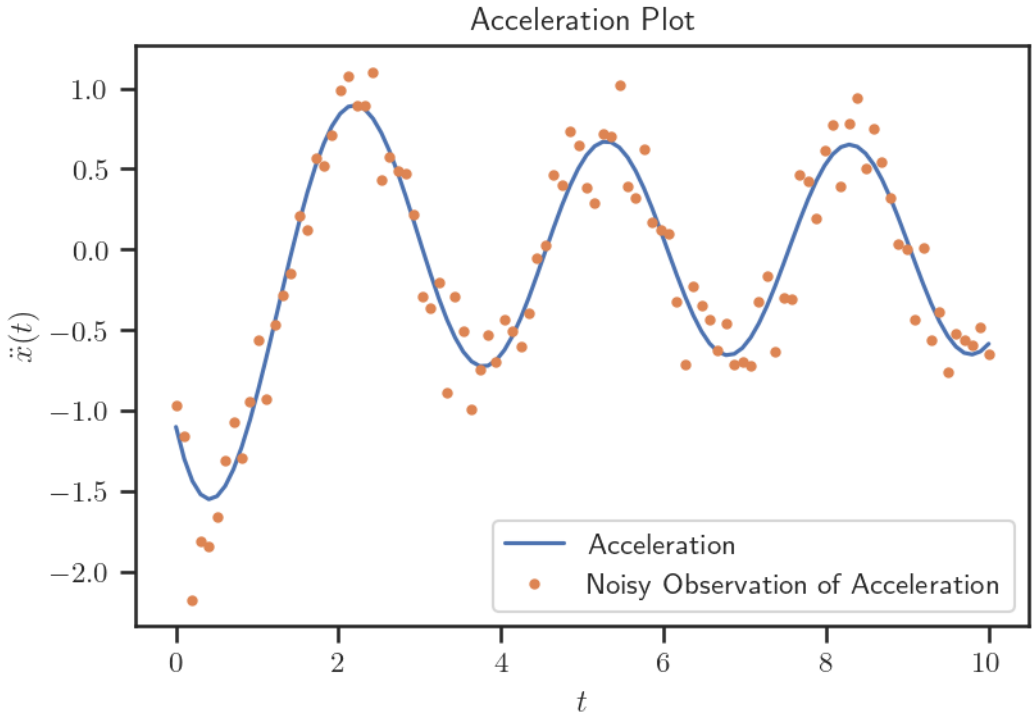
And now I am going to add some Gaussian noise to it:

[36]:
```
# 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:

[37]:
```
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')
ax.set_title("Acceleration Plot");
```



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.

4.1 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:

By converting the second order ODE into a system of first order ODE's, the two equations of interest are:

$$\frac{dx}{dt} = \dot{x}$$

$$\frac{d\dot{x}}{dt} = \ddot{x} = -2\zeta\omega_0\dot{x} - \omega_0^2 x + u_0\cos(\omega t)$$

Discretize in time to identify the transition matrix and control matrix:

$$\frac{x(t+\Delta t) - x(t)}{\Delta t} = \dot{x}(t) \rightarrow x(t+\Delta t) = x(t) + \Delta t\dot{x}(t)$$

$$\frac{\dot{x}(t+\Delta t) - \dot{x}(t)}{\Delta t} = \ddot{x}(t) \rightarrow \dot{x}(t+\Delta t) = \dot{x}(t) + \Delta t\ddot{x}(t)$$

$$\begin{aligned} &= \dot{x}(t) + \Delta t\left(-2\zeta\omega_0\dot{x}(t) - \omega_0^2 x(t) + u_0\cos(\omega t)\right) \\ &= -\omega_0^2\Delta t x(t) + \dot{x}(t) - 2\Delta t\zeta\omega_0\dot{x}(t) + u_0\Delta t\cos(\omega t) \end{aligned}$$

By separating terms in the above expressions, and knowing the relationship below:

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

the transition matrix and control matrix can be identified:

$$\mathbf{A} = \begin{bmatrix} 1 & \Delta t \\ -\omega_0^2\Delta t & 1 - 2\Delta t\zeta\omega_0 \end{bmatrix}$$

$$B = \begin{bmatrix} 0 \\ u_0\Delta t \end{bmatrix}$$

where:

$$\mathbf{x}_{j+1} = \begin{bmatrix} x((j+1)\Delta t) \\ \dot{x}((j+1)\Delta t) \end{bmatrix},$$

$$\mathbf{x}_j = \begin{bmatrix} x(j\Delta t) \\ \dot{x}(j\Delta t) \end{bmatrix},$$

$u_j = \cos(\omega(j\Delta t))$, and $\mathbf{z}_j \sim N(\mathbf{0}, \mathbf{Q})$.

Based on content from the lecture notes and hands-on activities from this course, process covariance matrix was selected to encapsulate the following information:

- Discretization error is present for both the position and velocity
- External forces not captured by u_j may be present
- The position and velocity are uncorrelated

As a result:

$$\mathbf{Q} = \begin{bmatrix} \epsilon & 0 \\ 0 & \sigma_q^2 \end{bmatrix}$$

where ϵ captures the discretization error of the Euler scheme, and σ_q^2 captures both the discretization error of the Euler scheme and any external forces to the system that are not captured by u_j. The values for ϵ and σ_q^2 were arbitrarily selected to mirror the values selected in the corresponding course hands-on activities.

[38]:
```
# define transition matrix (see work above)
A = np.array(
    [
        [1, dt],
        [-dt*omega0**2, 1-2*dt*zeta*omega0]
    ]
)
```

[39]:
```
# define control matrix (see work above)
B = np.array(
    [
        [0],
        [u0*dt]
    ]
)
```

[40]:
```
# define the variance for the process noise for position (using same value as␣
 ↪in the hands-on activity)
epsilon = 1e-6
# define the standard deviation for process noise for velocity (using same␣
 ↪value as in the hands-on activity)
sigma_q = 1e-2
# define process covariance (see work above)
Q = np.array(
    [
        [epsilon, 0.0],
        [0.0, sigma_q**2]
    ]
)
# reference(s): hands-on activities 19.1, 20.1
```

4.2 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:

$$\begin{aligned} y_j &= \ddot{x}(j\Delta t) - u_0 \cos(\omega t) + w_j \\ &= -2\zeta\omega_0 \dot{x}(j\Delta t) - \omega_0^2 x(j\Delta t) + u_0 \cos(\omega t) - u_0 \cos(\omega t) + w_j \\ &= -\omega_0^2 x(j\Delta t) - 2\zeta\omega_0 \dot{x}(j\Delta t) + w_j \end{aligned}$$

By separating terms in the above expression, and knowing the relationship below:

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

the emission matrix can be identified:

$$\mathbf{C} = \begin{bmatrix} -\omega_0^2 & -2\zeta\omega_0 \end{bmatrix}$$

where:

$$\mathbf{x}_j = \begin{bmatrix} x(j\Delta t) \\ \dot{x}(j\Delta t) \end{bmatrix},$$

$w_j \sim N(\mathbf{0}, R)$, and measurement covariance matrix $R = [\sigma_r^2]$. The value σ_r^2 captures the variance of the acceleration measurements. The same σ_r value is being used for both the definition of R and the generation of the synthetic noisy acceleration measurements.

[41]:
```
# define emission matrix (see work above)
C = np.array(
    [
        [-omega0**2, -2*zeta*omega0]
    ]
)
# define measurement covariance (see work above)
```

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

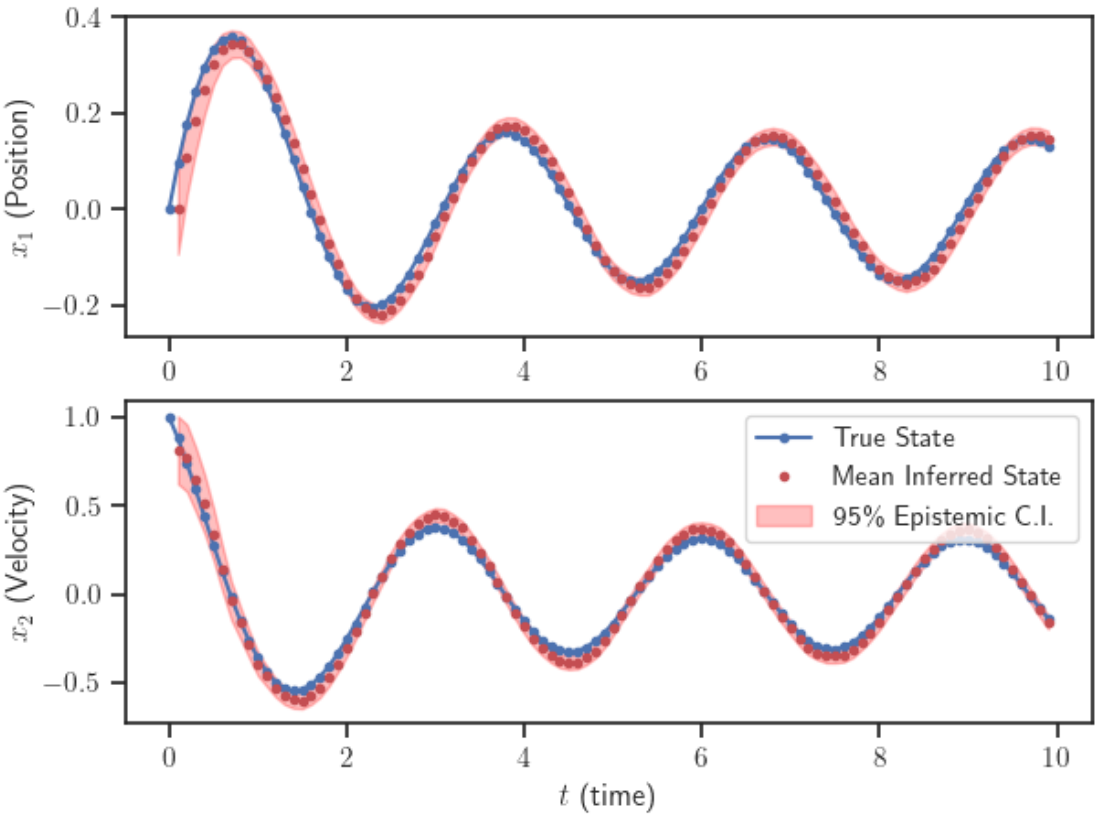
4.3 Part C - Apply the Kalman filter

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).

[42]:
```
# read in necessary function from filterpy
from filterpy.kalman import KalmanFilter
# define the dimensionality of the state space
kf = KalmanFilter(dim_x=2, dim_z=1)
# initial mean (with initial position = 0, initial velocity = 1)
mu0 = np.zeros((2,))
mu0[0] = 0
mu0[1] = 1
# initial covariance
V0 = np.array([0.1**2, 0.1**2]) * np.eye(2)
# define discretized force vector
us = np.cos(omega*t_eval).reshape(-1, 1)
# making the kf object aware of the various defined parameters
kf.x = mu0
kf.P = V0
kf.Q = Q
kf.R = R
kf.H = C
kf.F = A
kf.B = B
# run the batch filter
means, covs, _, _ = kf.batch_filter(observations, us=us)
# reference(s): hands-on activity 20.1
```

[43]:
```
# ** Plot estimates of the state with 95% credible interval **
y_labels = ['$x_1$ (Position)', '$x_2$ (Velocity)']

dpi = 150

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)

times = dt * np.arange(n_steps + 1)

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(times[0:n_steps], sol.y[j, :].T,
               'b.-', label="True State")
    ax[j].plot(times[1:n_steps], means[1:n_steps, j],
               'r.', label="Mean Inferred State")
    ax[j].fill_between(
        times[1:n_steps],
        (
            means[1:n_steps, j]
            - 2.0 * np.sqrt(covs[1:n_steps, j, j])
        ),
        (
            means[1:n_steps, j]
            + 2.0 * np.sqrt(covs[1:n_steps, j, j])
        ),
        color='red',
        label="95\% Epistemic C.I.",
        alpha=0.25
    )
plt.legend(loc='best');
# reference(s): hands-on activities 19.1, 20.1
```



4.4 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.

Answer:

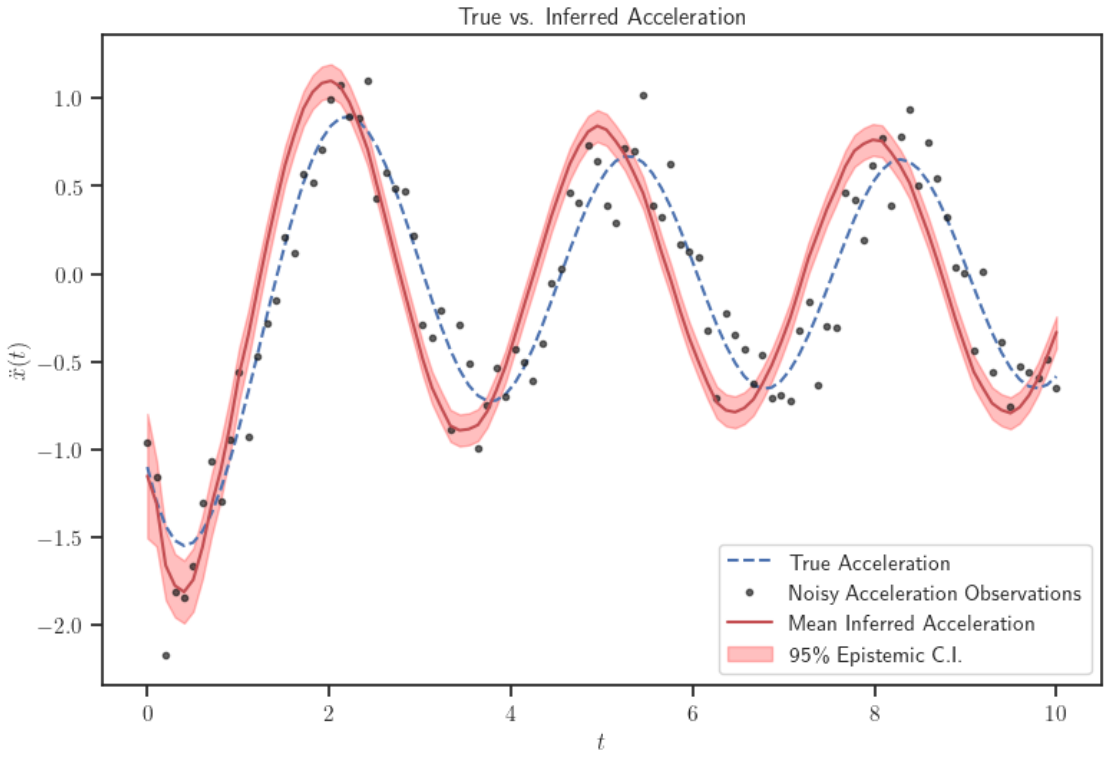
From the Kalman Filter batch run above, we have the covariances of the state at each time step. We want to find the variance in the output emission, y_j. Assuming that x_j and w_j are uncorrelated:

$$\begin{aligned} Var\left[y_j\right] &= Var\left[\mathbf{C}\mathbf{x}_j + w_j\right] \\ &= Var\left[\mathbf{C}\mathbf{x}_j\right] + Var\left[w_j\right] + 0 \\ &= \mathbf{C}Var\left[\mathbf{x}_j\right]\mathbf{C}^T + Var\left[w_j\right] \end{aligned}$$

where the first term in the expression, $\mathbf{C}Var\left[\mathbf{x}_j\right]\mathbf{C}^T$, corresponds to epistemic uncertainty of y_j and the second term in the expression, $Var\left[w_j\right]$, corresponds to aleatory uncertainty of y_j.

Together, the total uncertainty in y_j is known. Below, a 95% credible interval of only the epistemic uncertainty will be plotted.

[44]:
```
# calculate the mean inferred acceleration using the emission equation
y = C @ means.T
# determining the epistemic uncertainty in the mean inferred acceleration
# is is known that if A is a matrix: Var[Ax] = AVar[x]A.T
yVar = C @ covs @ C.T
# define upper and lower credible interval profiles (epistemic only)
upper_e = y.T + 2.0 * (np.sqrt(yVar[:, 0, 0]).reshape(-1,1))
lower_e = y.T - 2.0 * (np.sqrt(yVar[:, 0, 0]).reshape(-1,1))

# plotting results with 95% epistemic credible interval
fig, ax = plt.subplots(figsize=(9,6))
ax.plot(t_eval, true_acc, 'b--', label='True Acceleration')
ax.plot(
    t_eval, observations,
    'k.', label='Noisy Acceleration Observations', alpha=0.66
)
ax.plot(
    t_eval, y.T,
    'r-', label='Mean Inferred Acceleration'
)
ax.fill_between(t_eval, upper_e[:,0], lower_e[:,0],
                label="95\% Epistemic C.I.",
                color='red', alpha=0.25)
ax.set_xlabel('$t$')
ax.set_ylabel('$\ddot{x}(t)$')
ax.set_title("True vs. Inferred Acceleration")
plt.legend(loc='best');
# reference(s): matrix cookbook, hands-on activity 20.1, lecture 5 reading
```

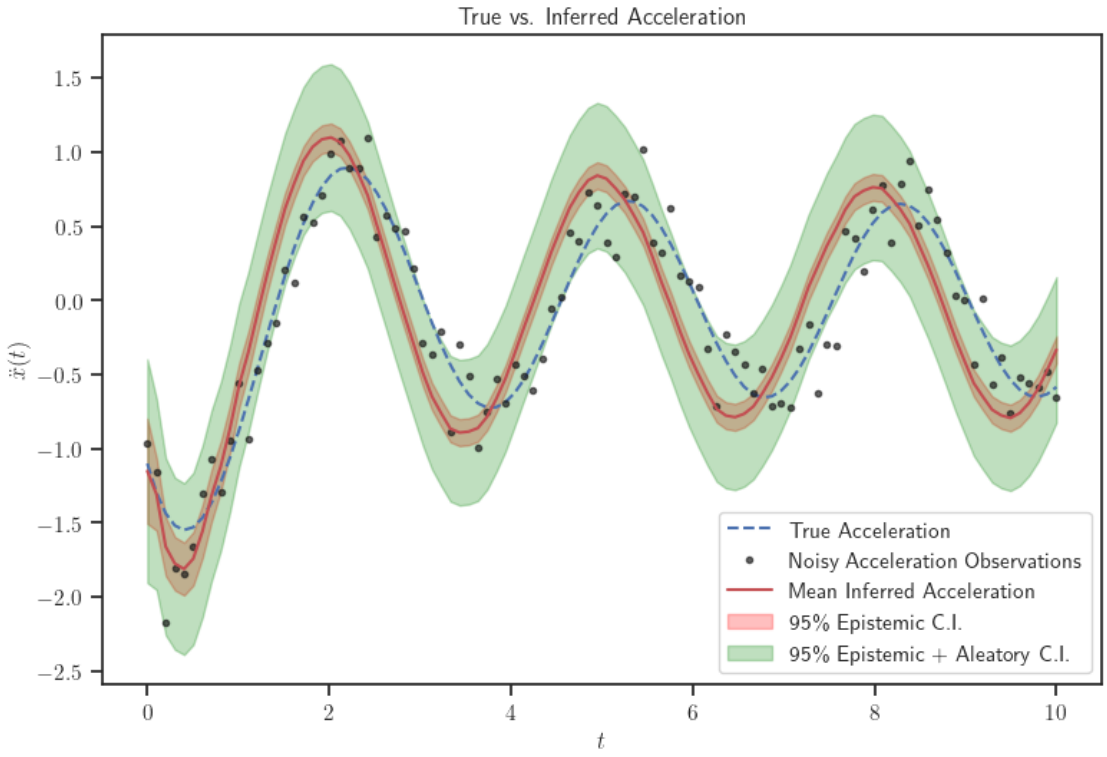


Optional Exercise:

The above plot serves as the answer submission for Problem 3 Part 3D. As an optional exercise (and for more thorough analysis) the above plot was repeaded with the *inclusion* of aleatory uncertainty. It is my hope that everything below this point has no effect on the homework score.

[45]:
```
# Optional exercise to plot epistemic + aleatory uncertainty credible interval␣
 ↪as well
# calculate standard deviation of measurement variance
meas_std = np.sqrt(R[0, 0])
# define upper and lower credible interval profiles (epistemic + aleatory)
upper_ae = y.T + 2.0 * (np.sqrt(yVar[:, 0, 0]).reshape(-1,1) + meas_std)
lower_ae = y.T - 2.0 * (np.sqrt(yVar[:, 0, 0]).reshape(-1,1) + meas_std)

# plotting results with 95% epistemic credible interval
fig, ax = plt.subplots(figsize=(9,6))
ax.plot(t_eval, true_acc, 'b--', label='True Acceleration')
ax.plot(
    t_eval, observations,
    'k.', label='Noisy Acceleration Observations', alpha=0.66
)
ax.plot(
    t_eval, y.T,
```

```
    'r-', label='Mean Inferred Acceleration'
)
ax.fill_between(t_eval, upper_e[:,0], lower_e[:,0],
                label="95\% Epistemic C.I.",
                color='red', alpha=0.25)
ax.fill_between(t_eval, upper_ae[:,0], lower_ae[:,0],
                label="95\% Epistemic + Aleatory C.I.",
                color='green', alpha=0.25)
ax.set_xlabel('$t$')
ax.set_ylabel('$\ddot{x}(t)$')
ax.set_title("True vs. Inferred Acceleration")
plt.legend(loc='best');
```



Part D Discussion:

The above plots indicate that a large portion of the uncertainty comes from the aleatory component in this situation. The epistemic + aleatory credible interval about the mean inferred acceleration does appear to encapsulate a majority of the noisy acceleration observations as well as the true unobserved acceleration.