# 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 [ ]:
```
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)
```

## Student details


- **First Name:** Kyle
- **Last Name:** Illenden
- **Email:** killende@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 [ ]:
```
url = "https://github.com/PredictiveScienceLab/data-analytics-se/raw/master/lecture
download(url)
```

And you can load it using pandas:

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

Here is how the data look like:

In [ ]:
```
p1_data
```

Out[ ]:

| | 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

If you have never played before with pandas, you can find a nice tutorial here.

We have half a million data points. Let's extract the latitude and longitude and put them in a numpy array:

In [ ]:
```
loc_data = p1_data[['Lon', 'Lat']]
loc_data
```

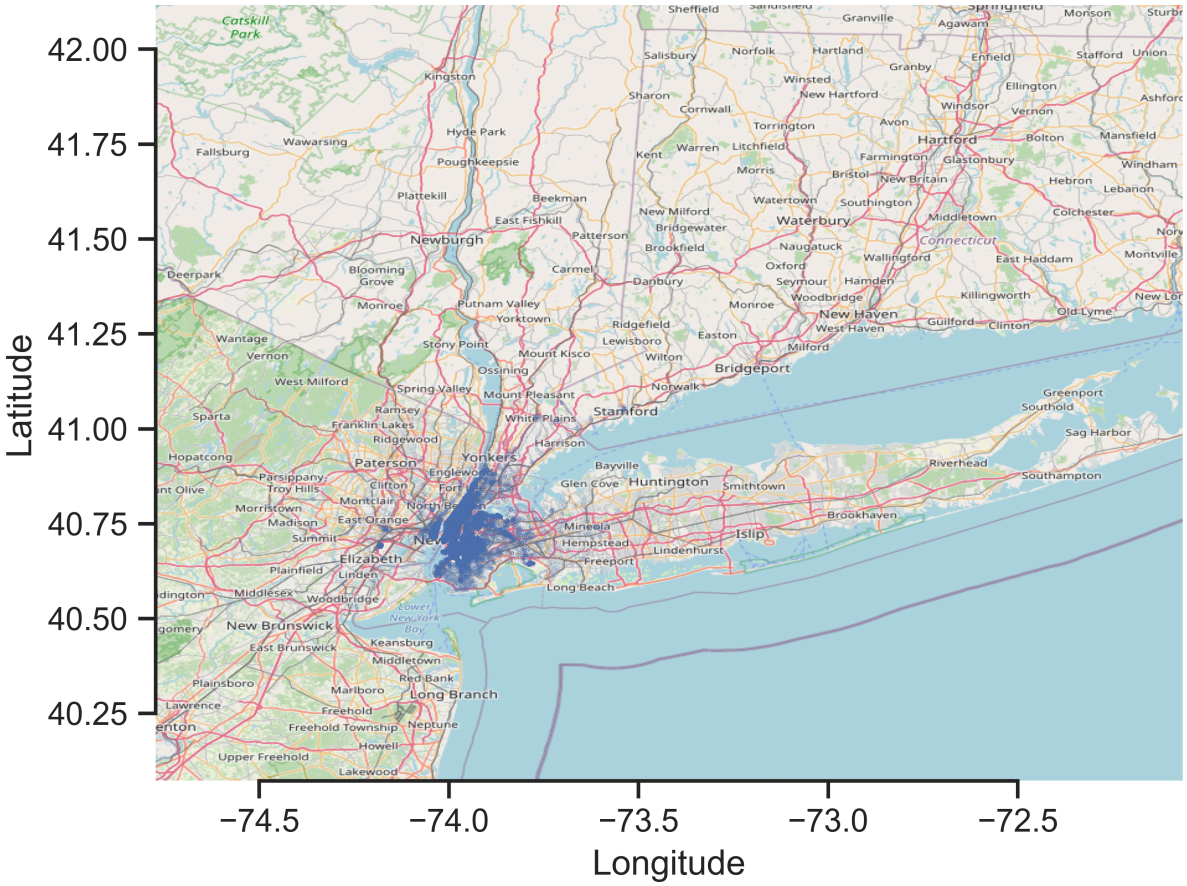
Out[ ]:

| | 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 visualize these points on the map of New York City:

In [ ]:
```
url = "https://github.com/PredictiveScienceLab/data-analytics-se/raw/master/lecture
download(url)
ny_map = plt.imread('ny_map.png')
box = ((loc_data.Lon.min(), loc_data.Lon.max(),
        loc_data.Lat.min(), loc_data.Lat.max()))
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')
sns.despine(trim=True);
```




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

In [ ]:
```
p1_train_data = loc_data[:50_000] # keeping 50,000 to follow piazza post (question
```

# Part A - Splitting New York City into Subregions

Suppose you are assigned to split New York City into operating subregions with equal demand. When a pickup is requested in each subregion, only the drivers in that region are called. Note that this can become a challenging problem very quickly. We are not looking for the best possible answer here. We are looking for a data-informed heuristic solution that is good enough.

Do (at least) the following:

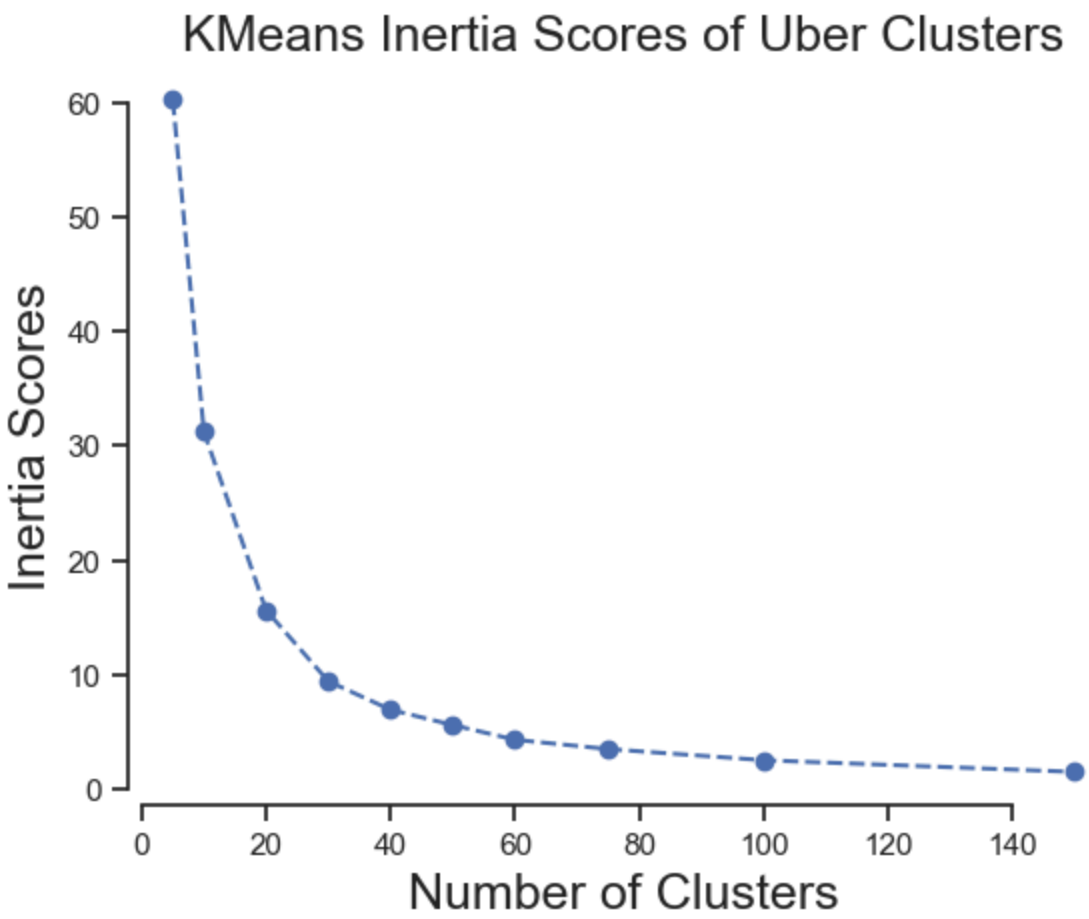
- Use Kmeans clustering on the pickup data with different numbers of clusters;
- Visualize the labels of the clusters on the map using different colors (see the hands-on activities);
- Visualize the centers of the discovered Kmeans clusters (in red color);
- Use common sense, e.g., ensure there are enough clusters so no region crosses the water. 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 a much faster version of Kmeans suitable for large datasets (>10K observations);

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

**Using KMeans**

In [ ]:
```
from sklearn.cluster import KMeans, MiniBatchKMeans

inertia = []
for i in [5,10,20,30,40,50,60,75,100,150]:
    kmeans = KMeans(n_clusters = i, n_init='auto')
    kmeans.fit(p1_train_data)
    inertia.append(kmeans.inertia_)

fig, ax = plt.subplots()
ax.plot([5,10,20,30,40,50,60,75,100,150], inertia, marker='o', linestyle='--')
ax.set_xlabel('Number of Clusters', fontsize=18)
ax.set_ylabel('Inertia Scores', fontsize=18)
ax.set_title('KMeans Inertia Scores of Uber Clusters', fontsize=18)
sns.despine(trim=True);
```

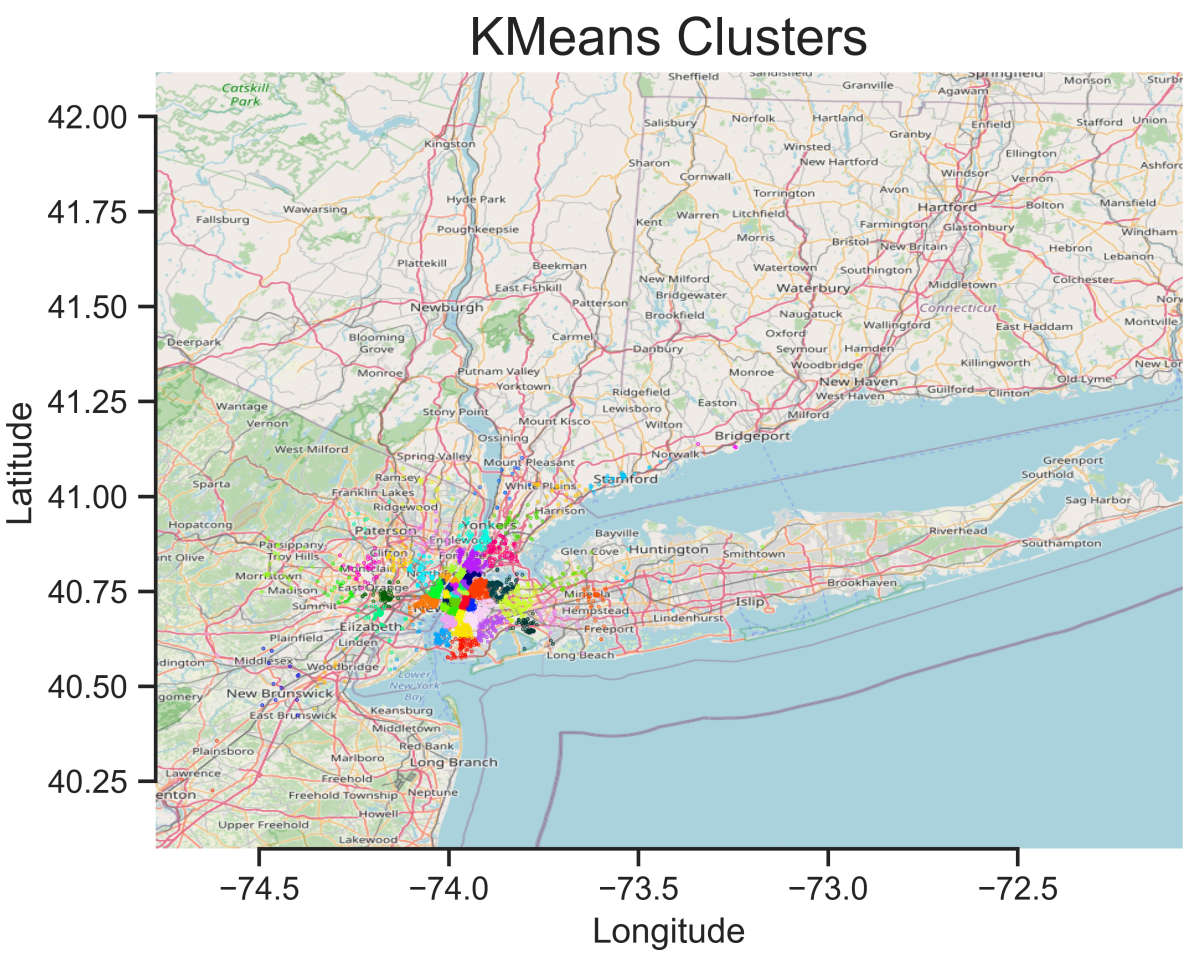



Using the inertia score vs number of clusters above, I am looking to minimize both values. This provides me with a faster execution using KMeans, while still creating enough valuable

clusters. For this I will be choosing 60 cluster, because we do not want any clusters to cross the water. This allows for more minimization of inertia score, with a slight efficiency drain.

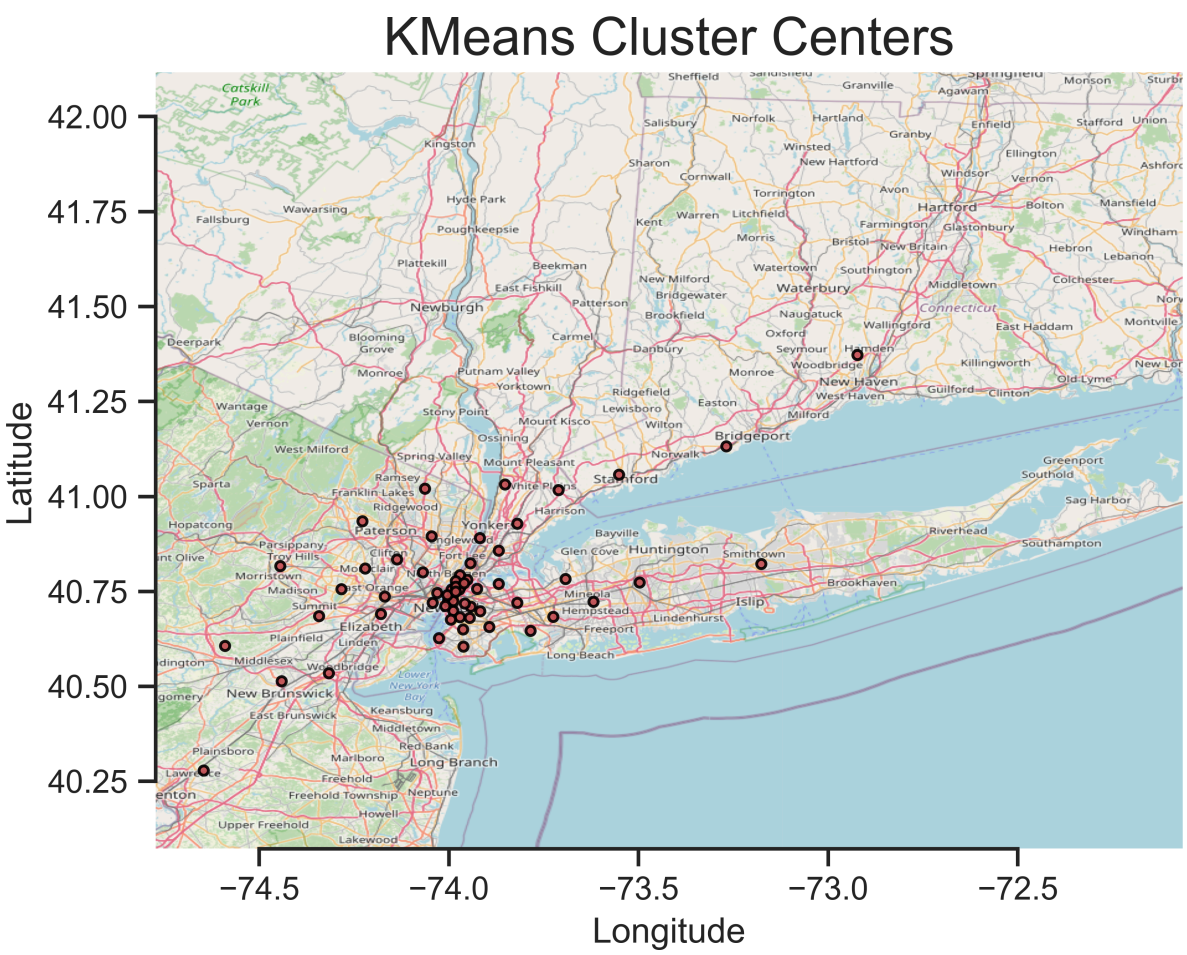
In [ ]:
```
no_of_clusters = 60
model = KMeans(n_clusters=no_of_clusters, n_init='auto').fit(p1_train_data)
```

In [ ]:
```
labels = model.predict(p1_train_data)

fig, ax = plt.subplots(dpi=600)
ax.scatter(
    p1_train_data.Lon,
    p1_train_data.Lat,
    zorder=1,
    alpha= 0.8,
    c=labels,
    cmap='gist_ncar',
    s=0.1,
)
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')
ax.set_title('KMeans Clusters', fontsize=18)
sns.despine(trim=True);
```




In [ ]:

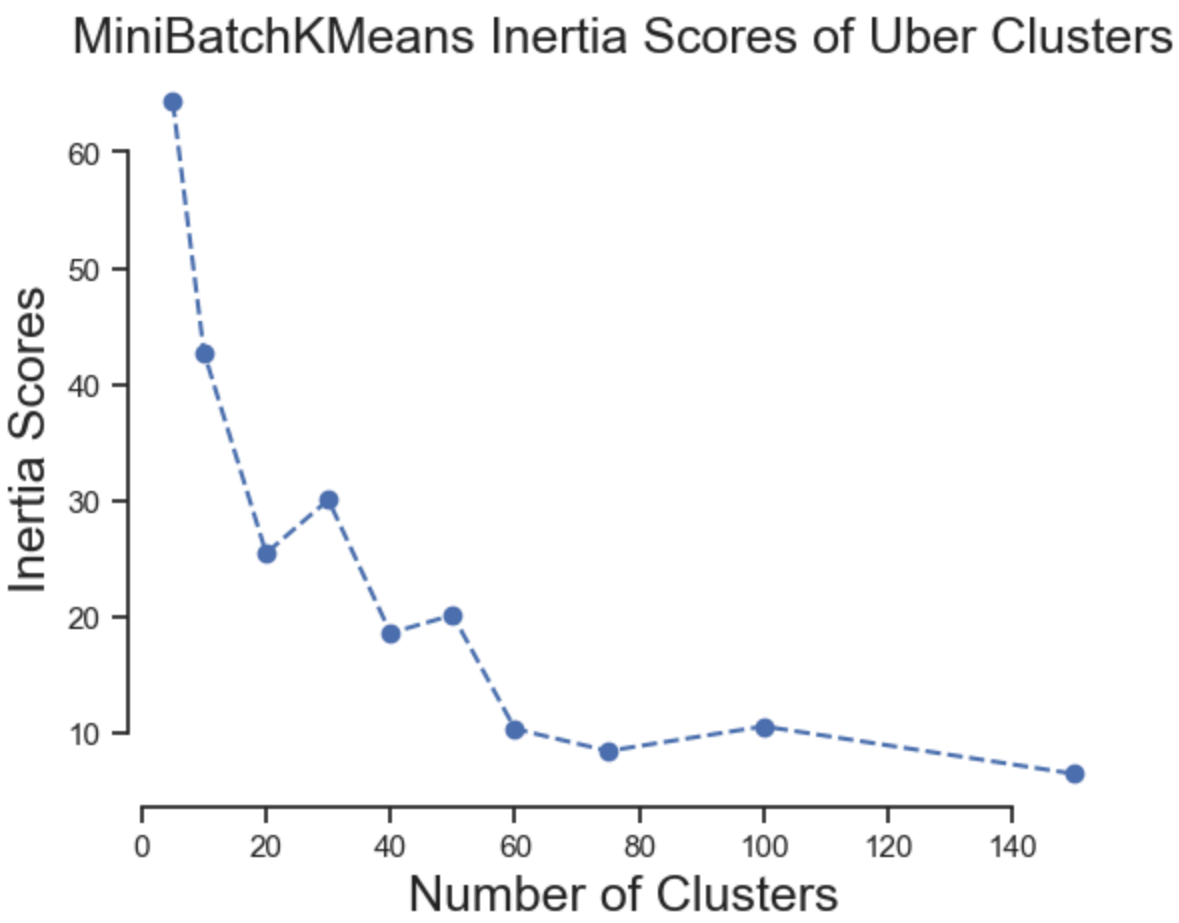
```
fig, ax = plt.subplots(dpi=600)
ax.scatter(
    model.cluster_centers_[:,0],
    model.cluster_centers_[:,1],
    zorder=1,
    alpha= 0.9,
    c='r',
    s=10,
    edgecolors='black'
)
ax.set_xlim(box[0],box[1])
ax.set_ylim(box[2],box[3])
ax.imshow(
    ny_map,
    zorder=0,
    extent=box,
    aspect= 'equal'
)
ax.set_xlabel('Longitude')
ax.set_ylabel('Latitude')
ax.set_title('KMeans Cluster Centers', fontsize=18)
sns.despine(trim=True);
```




**Now using MiniBatchKMeans**

In [ ]:

```
inertia = []
for i in [5,10,20,30,40,50,60,75,100,150]:
    kmeans = MiniBatchKMeans(n_clusters = i, n_init='auto')
    kmeans.fit(p1_train_data)
    inertia.append(kmeans.inertia_)

fig, ax = plt.subplots()
ax.plot([5,10,20,30,40,50,60,75,100,150], inertia, marker='o', linestyle='--')
ax.set_xlabel('Number of Clusters', fontsize=18)
ax.set_ylabel('Inertia Scores', fontsize=18)
ax.set_title('MiniBatchKMeans Inertia Scores of Uber Clusters', fontsize=18)
sns.despine(trim=True);
```

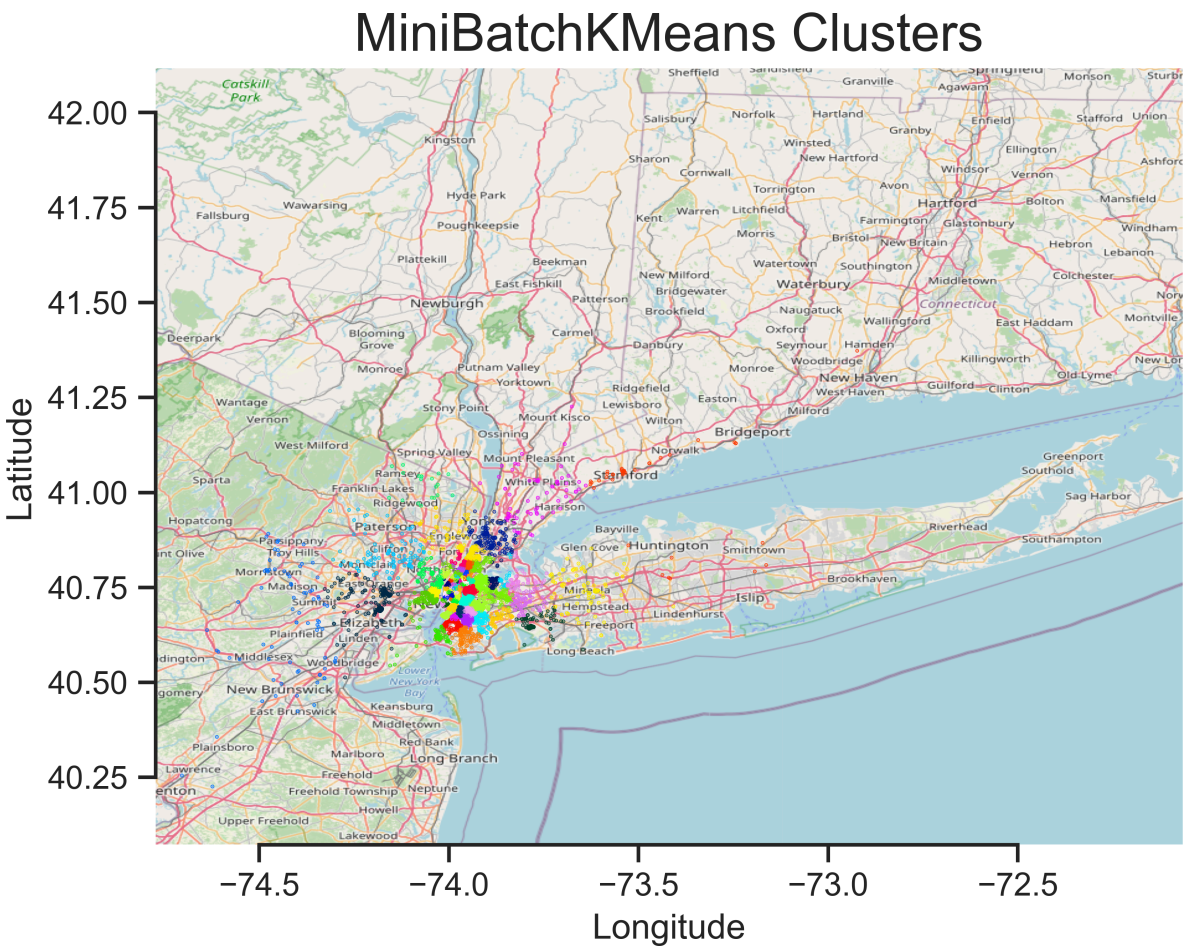



Using the inertia score vs number of clusters above, I am looking to minimize both values. This provides me with a faster execution using KMeans, while still creating enough valuable clusters. For this I will be choosing 80 cluster, because we do not want any clusters to cross the water. This allows for more minimization of inertia score, with a slight efficiency drain.

In [ ]:
```
no_of_clusters = 80
model = MiniBatchKMeans(n_clusters=no_of_clusters, n_init='auto').fit(p1_train_data
```

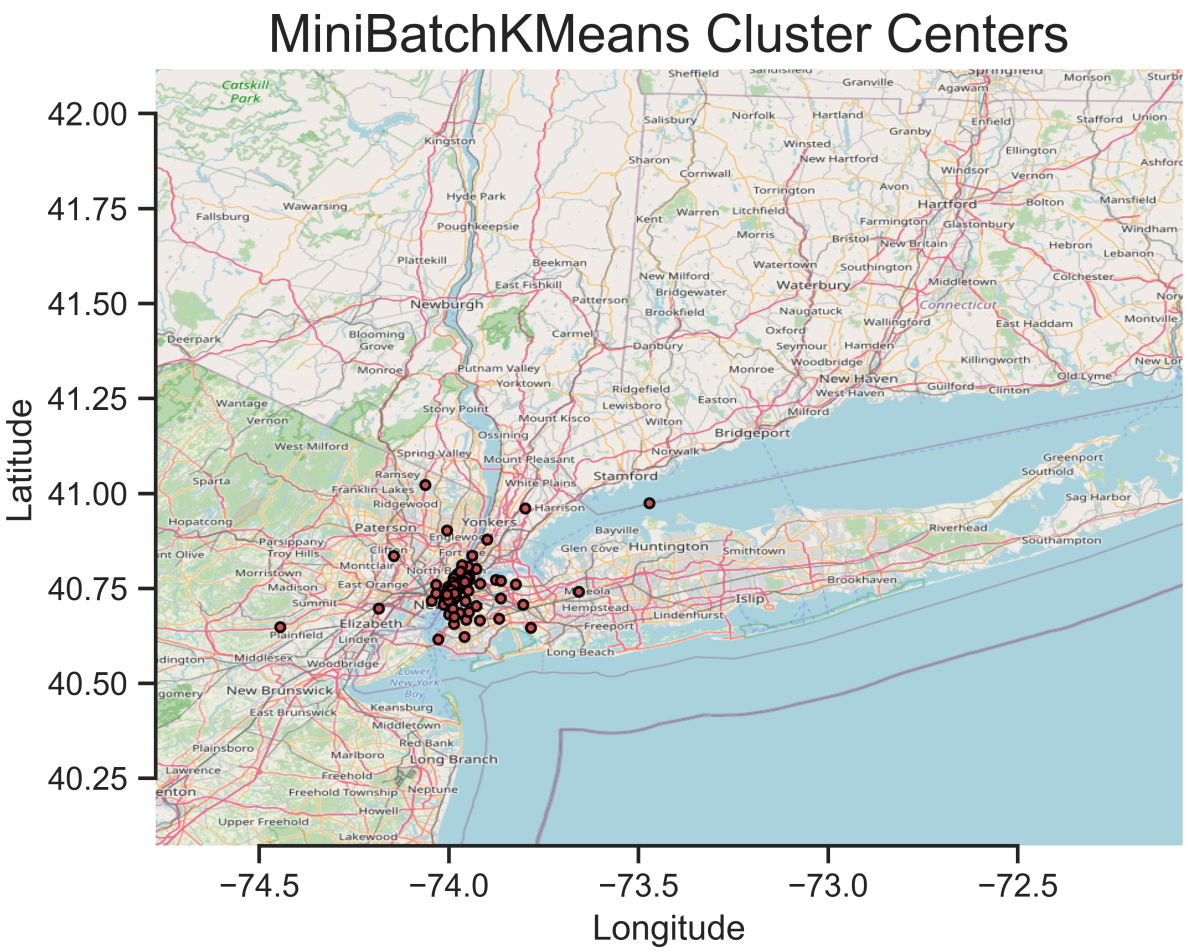
In [ ]:
```
labels = model.predict(p1_train_data)

fig, ax = plt.subplots(dpi=600)
ax.scatter(
    p1_train_data.Lon,
    p1_train_data.Lat,
    zorder=1,
    alpha= 0.8,
    c=labels,
    cmap='gist_ncar',
    s=0.1,
)
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')
ax.set_title('MiniBatchKMeans Clusters', fontsize=18)
sns.despine(trim=True);
```




In [ ]:
```
fig, ax = plt.subplots(dpi=600)
ax.scatter(
    model.cluster_centers_[:,0],
    model.cluster_centers_[:,1],
    zorder=1,
    alpha= 0.9,
    c='r',
    s=10,
    edgecolors='black'
)
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')
```

```
ax.set_title('MiniBatchKMeans Cluster Centers', fontsize=18)
sns.despine(trim=True);
```




# Part B - Create a Stochastic Model of Pickups

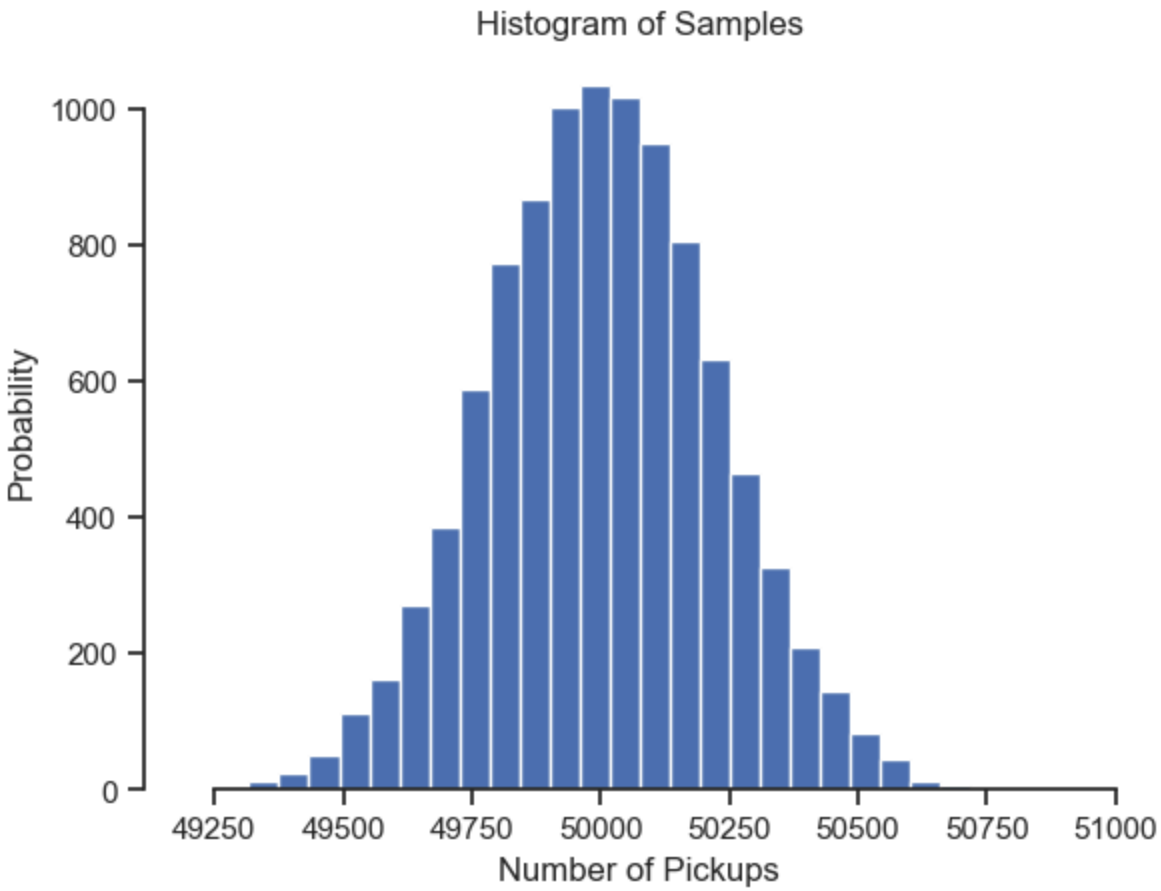
One of the key ingredients for a more sophisticated approach to optimizing the operations of Uber is the construction of a stochastic model of the demand for pickups. The ideal model for this problem is the Poisson Point Process. However, we will do something more straightforward, using the Gaussian 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 the 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 [ ]:

```
# Your code here
from scipy.stats import poisson

mean = len(p1_train_data)

samples = 10000

RV = poisson(mu=mean)

pickup_samples = RV.rvs(size=samples)

plt.hist(pickup_samples, bins=30)
plt.xlabel('Number of Pickups')
plt.ylabel('Probability')
plt.title('Histogram of Samples')
sns.despine(trim=True);
```



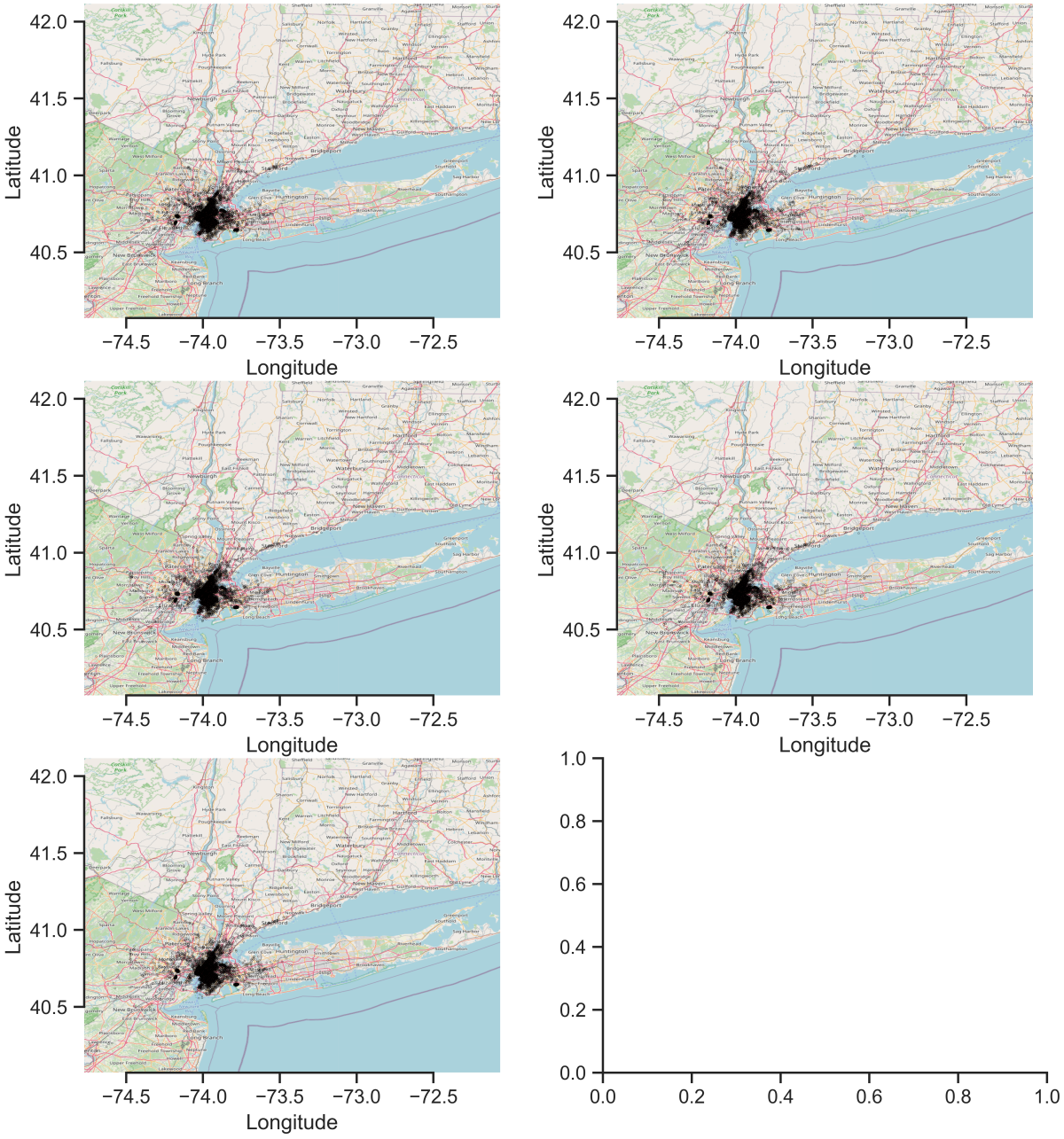
## Subpart B.II - Sample some random monthly pickup numbers

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 [ ]:
```
from sklearn.mixture import GaussianMixture

model = GaussianMixture(n_components=no_of_clusters).fit(p1_train_data)
```

In [ ]:
```
cols = 2
rows = 3

fig, axs = plt.subplots(rows,cols,figsize=(10,11),dpi=600)

num_plots = 5
plot_counter = 0
skip_next = False

for row_id in range(rows):
    for col_id in range(cols):
        if plot_counter < num_plots:
            pickup_samples = RV.rvs()
            location_samples = model.sample(pickup_samples)[0]

            axs[row_id, col_id].scatter(
                location_samples[:,0],
                location_samples[:,1],
                zorder=1,
                alpha= 0.9,
                c='r',
                s=0.01,
                edgecolors='black'
            )
            axs[row_id, col_id].set_xlim(box[0],box[1])
            axs[row_id, col_id].set_ylim(box[2],box[3])
            axs[row_id, col_id].imshow(
                ny_map,
                zorder=0,
                extent=box,
                aspect= 'equal'
            )
            axs[row_id, col_id].set_xlabel('Longitude')
            axs[row_id, col_id].set_ylabel('Latitude')
            plot_counter += 1

sns.despine(trim=True);
```




# Problem 2 - Counting Celestial Objects

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

Let's download it so that you have it here:

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

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

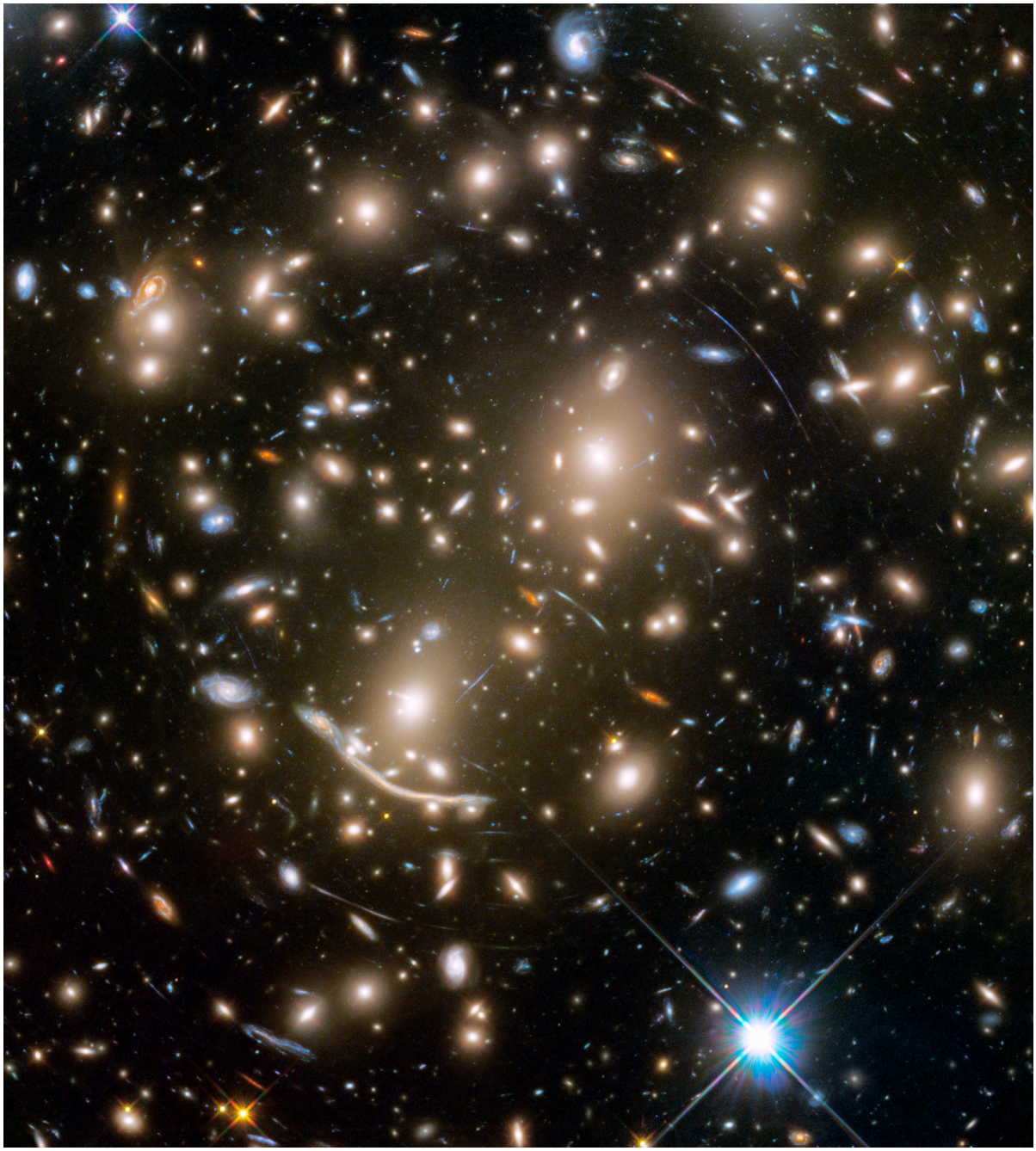
- Convert the picture to points sampled according to the intensity of light.
- Apply Gaussian mixture on the resulting points.
- Use the Bayesian Information Criterion to identify the number of components in the picture.
- Associate the number of components with the actual number of celestial objects.

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

Out[ ]:

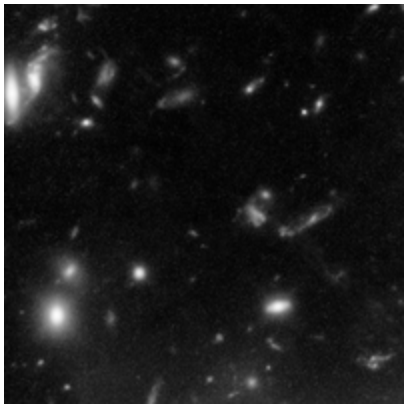


Now, we are going to convert it to grayscale and crop it to make the problem a little bit easier:

In [ ]:

```
img = hubble_image.convert('L').crop((100, 100, 300, 300))
img
```

Out[ ]:



Remember that black-and white images are matrices:

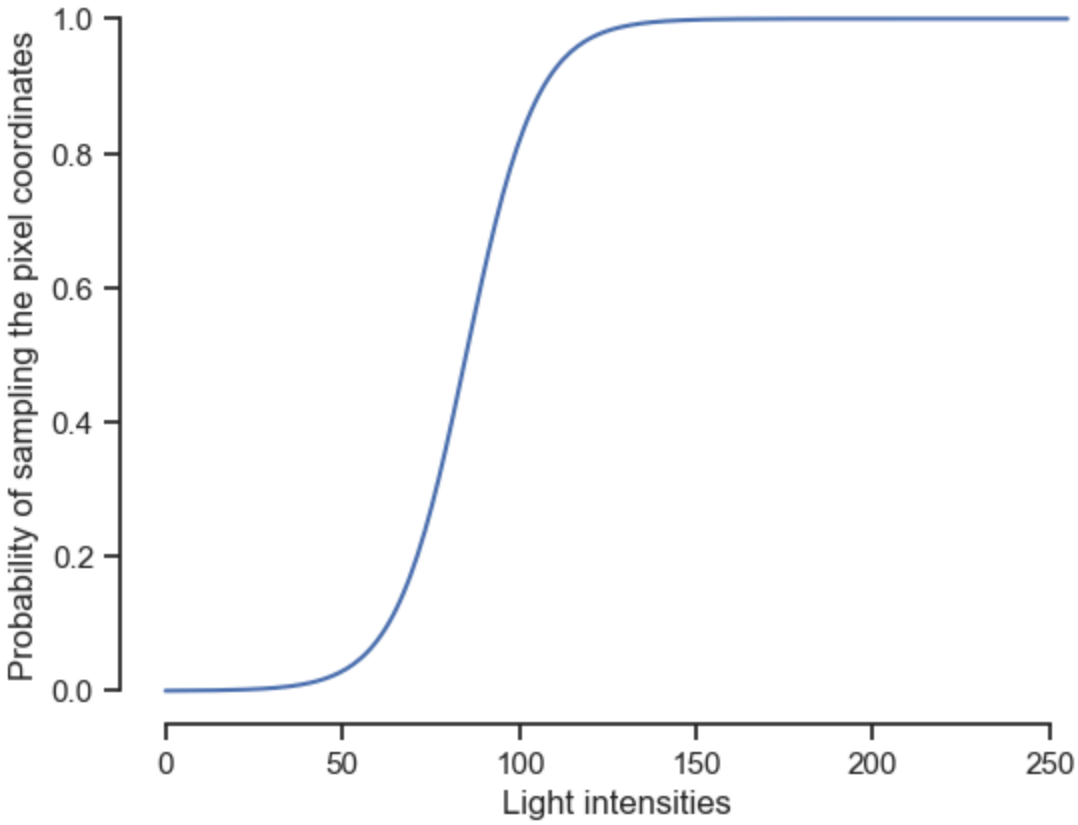
In [ ]:
```
img_ar = np.array(img)
img_ar
```

Out[ ]:
```
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 is $0$, corresponding to black, and the maximum is 255, corresponding to white. Anything in between is some shade of gray.

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

In [ ]:
```
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')
sns.despine(trim=True);
```



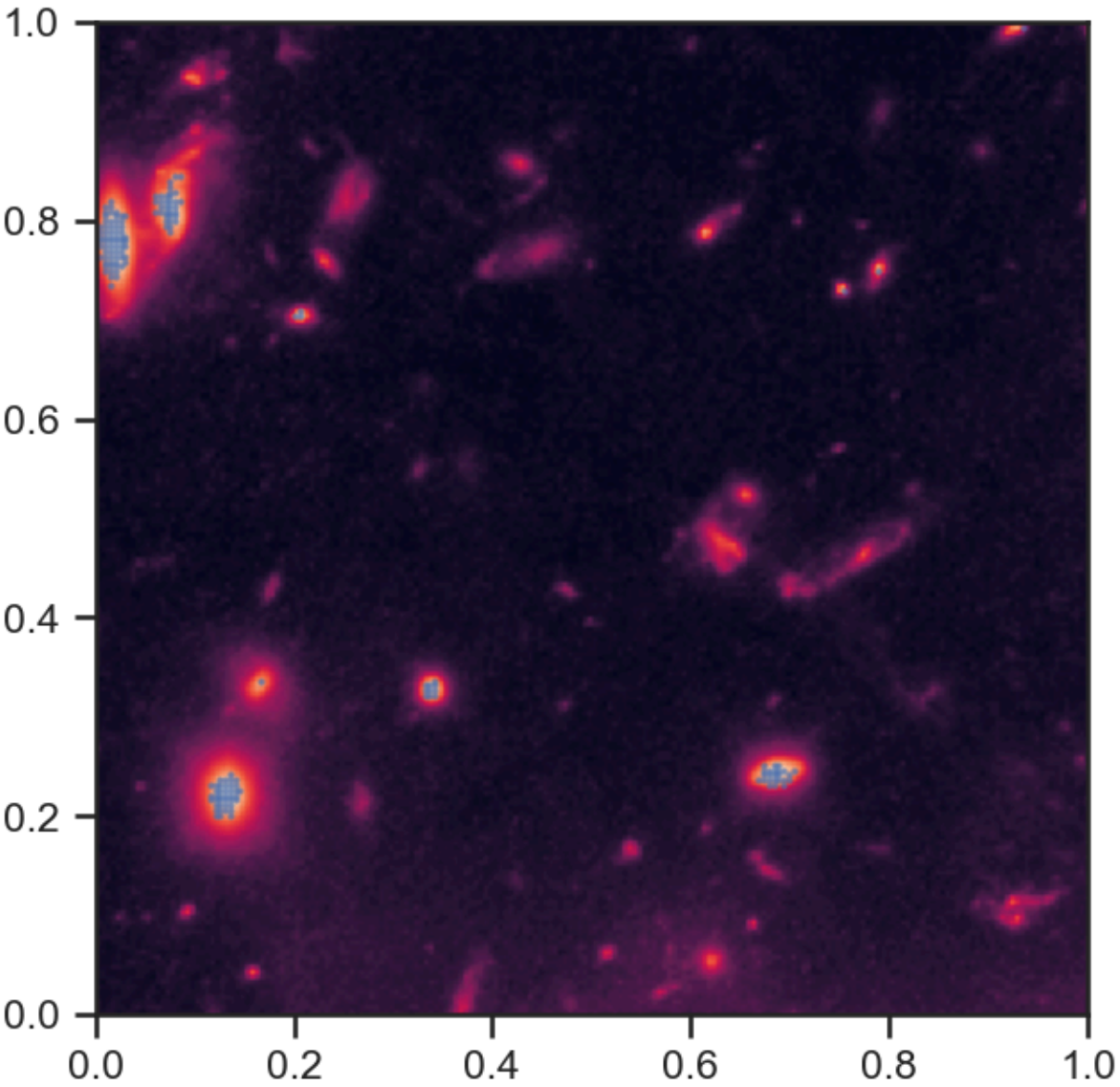

And here is the code that samples the pixel coordinates. I am organizing it into a function because we may want to use it with different pictures:

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

    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 it:

In [ ]:
```
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 playing with $\alpha$ and $\beta$ makes the whole thing more or less sensitive to the light intensity.

Complete the following function:

In [ ]:
```
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 - nc_min] = m.bic(locs)
        models.append(m)

    best_idx = np.argmin(bics)
    best_nc = best_idx + nc_min
    best_model = models[best_idx]

    return best_nc, best_model, locs
```

Once you have completed the code, try out the following images. Feel free to play with $\alpha$ and $\beta$ to improve the performance. **Do not try to make a perfect model. We would have to go beyond the Gaussian mixture model to do so. This is just a homework problem.**

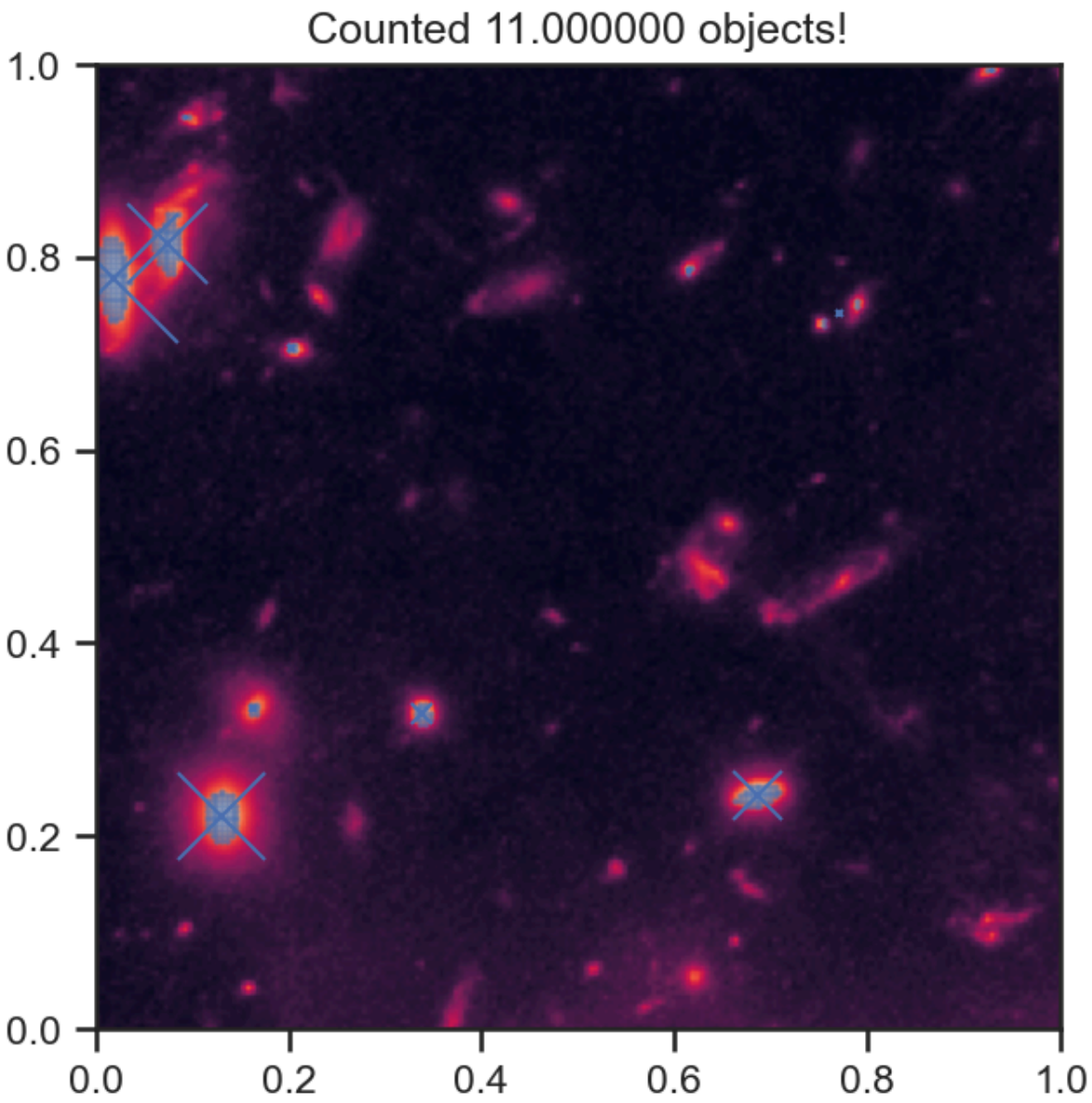
Here is a helpful function that you can use to visualize the results:

In [ ]:
```
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],
            'bx',
            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:f} objects!'.format(objs));
```

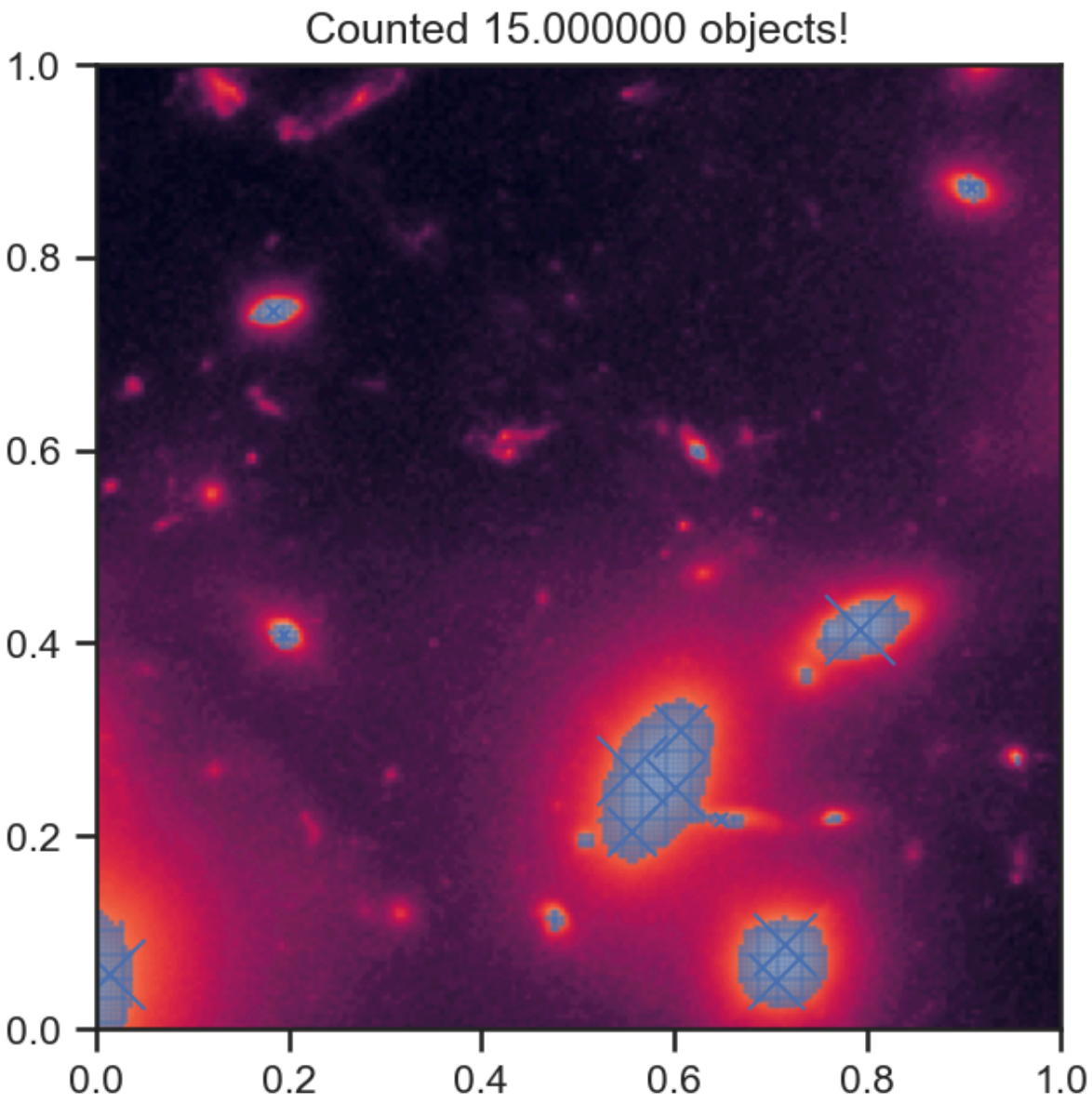
Here is how to use it:

In [ ]:
```
img = hubble_image.convert('L').crop((100, 100, 300, 300))
objs, model, locs = count_objs(img, alpha=1.0, beta=175)
visualize_counts(img, objs, model, locs)
```



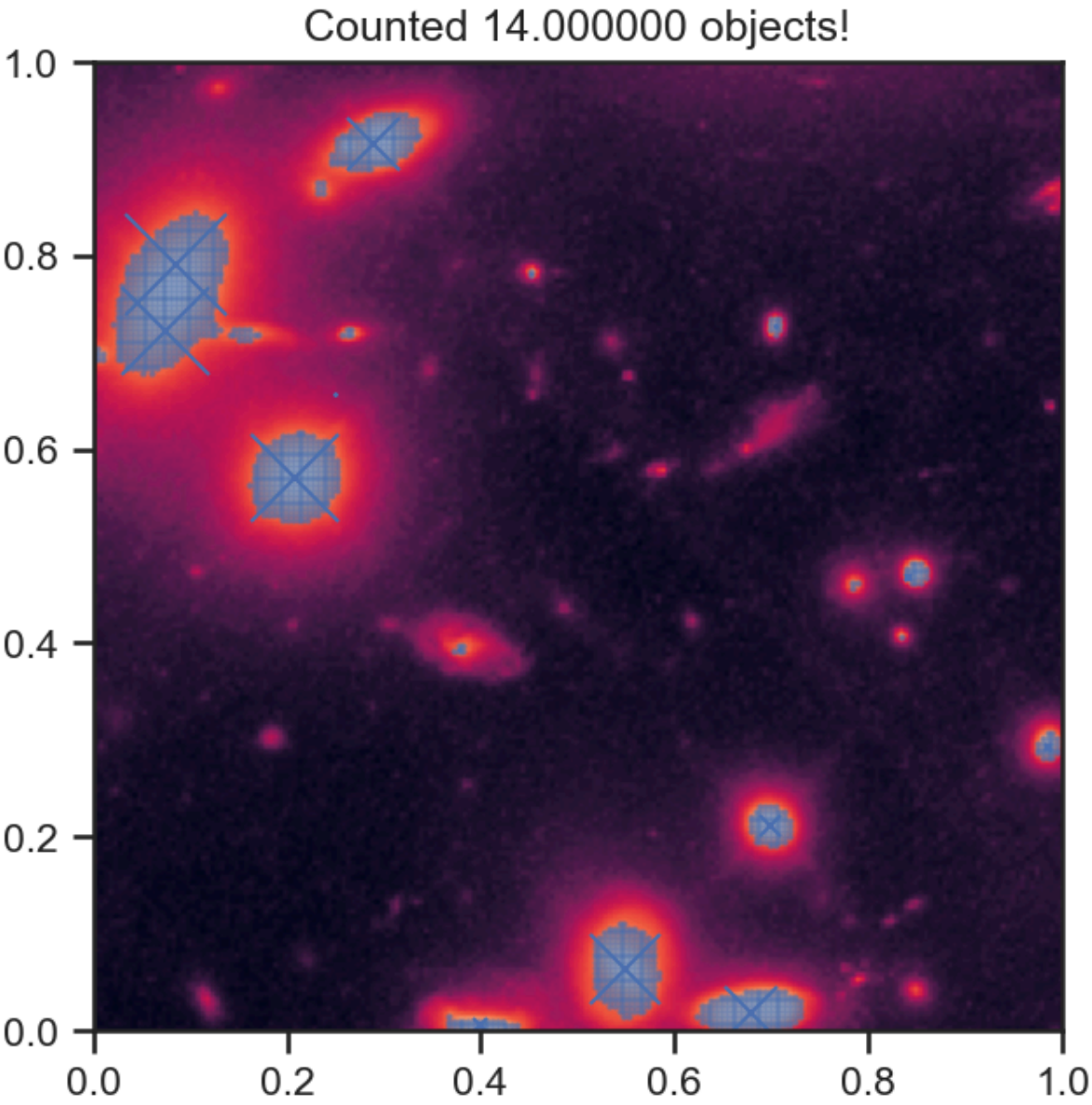

Try this image:

In [ ]:
```
img = hubble_image.convert('L').crop((200, 200, 400, 400))
objs, model, locs = count_objs(img, alpha=1.0, beta=175)
visualize_counts(img, objs, model, locs)
```




And this one:

In [ ]:
```
img = hubble_image.convert('L').crop((300, 300, 500, 500))
objs, model, locs = count_objs(img, alpha=1.0, beta=175)
visualize_counts(img, objs, model, locs)
```




# 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 $\zeta$, $\omega_0$, $u_0$, and $\omega$ 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 $\Delta t$ (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 $\sigma^2$ 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 that $\sigma_x^2$ and $\sigma_v^2$ are specific numbers we will specify below.

Before we go over the questions, let's write code that generates the actual trajectory of the system at some random initial conditions and 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. 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 [ ]:
```
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 *
    return res
```

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

In [ ]:
```
# 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, om
```

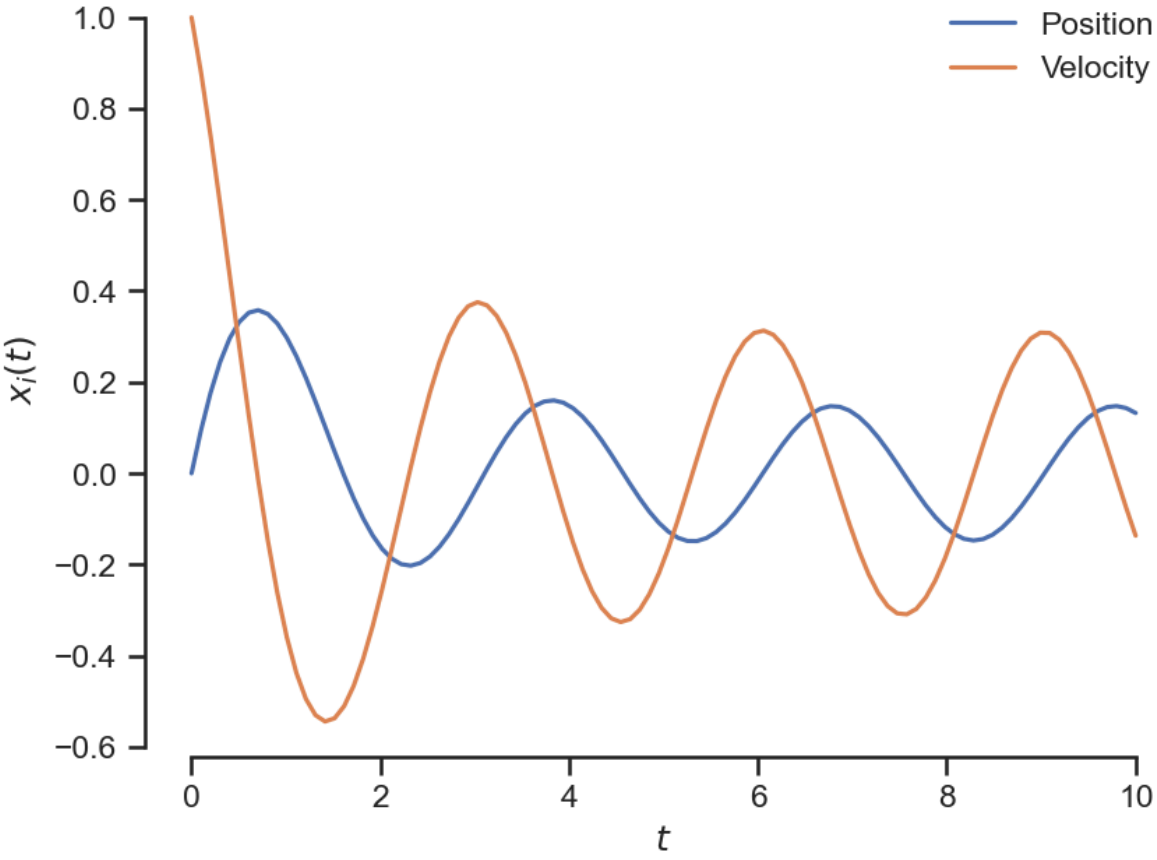
The solution is stored in the `sol` variable:

In [ ]:
```
sol.y.shape
```

Out[ ]:
```
(2, 100)
```

The shape of `sol.y` is (2, 100), which means that we have 100 timesteps and two variables (position and velocity). Let's plot the position and the velocity:

In [ ]:
```
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', frameon=False)
sns.despine(trim=True);
```

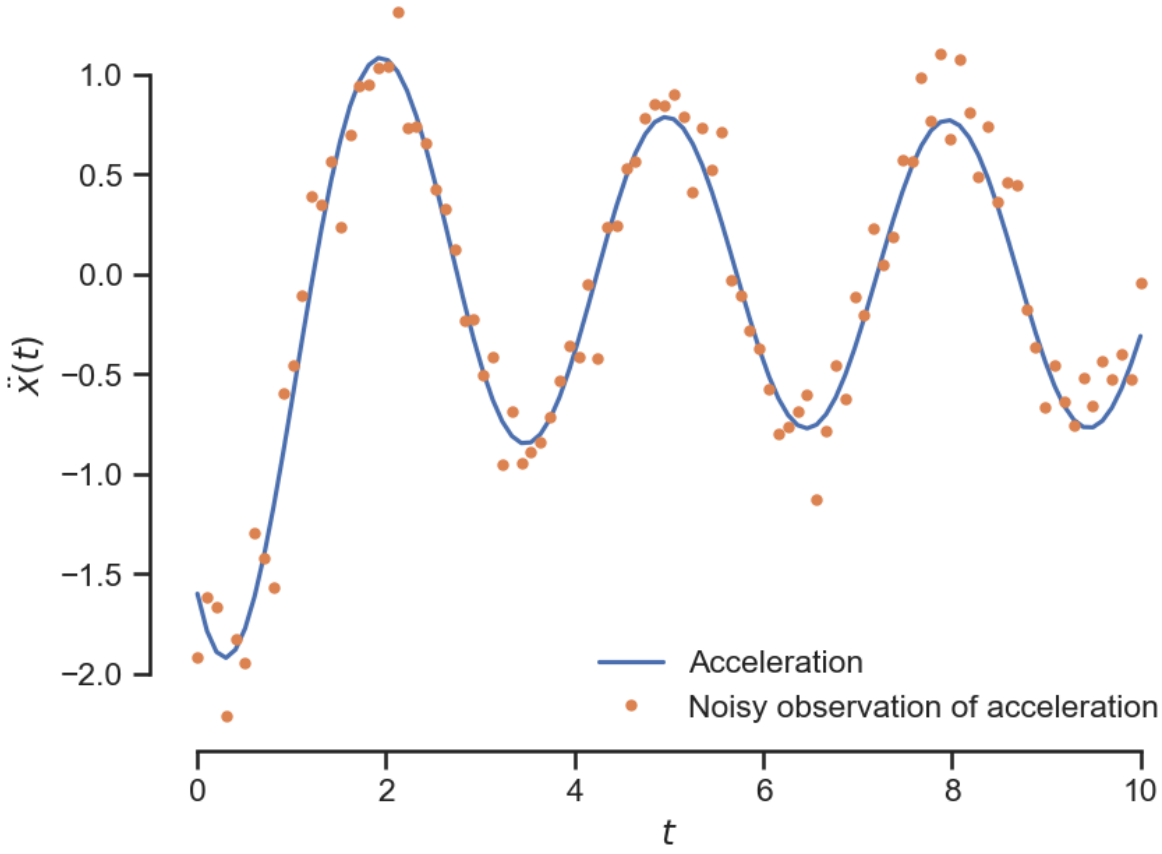



Let's now generate some synthetic observations of the acceleration with some given Gaussian noise. To get the acceleration, you can do this:

In [ ]:
```
# Compute external excitation.
us = u0 * np.cos(omega * t_eval)
# Subtract us from \ddot{x}
true_acc = np.array([rhs(t, x, omega0, zeta, u0, omega)[1] for (t, x) in zip(t_eval
```

Let's add some noise:

In [ ]:
```
sigma_r = 0.2
observations = true_acc + sigma_r * np.random.randn(true_acc.shape[0])

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(r'$\ddot{x}(t)$')
plt.legend(loc='best', frameon=False)
sns.despine(trim=True);
```




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 expressions for the following:

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

**Answer:**

Knowing:

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

and

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

from above, we can make this become:

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

We know (for continuous time):

$$\begin{bmatrix} x_2 \\ -2\zeta\omega_0 x_2 - \omega_0^2 x_1 + u_0\cos(\omega t) \end{bmatrix} \Rightarrow \begin{bmatrix} 0 & 1 \\ -\omega_0^2 & -2\zeta\omega_0 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} + \begin{bmatrix} 0 \\ 1 \end{bmatrix} u_0\cos(\omega t)$$

$$A_c = \begin{bmatrix} 0 & 1 \\ -\omega_0^2 & -2\zeta\omega_0 \end{bmatrix}$$

and

$$B_c = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$$

Now we can find the discretization equations:

$$A = I_n + \Delta t A_c$$

$$A = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} + \Delta t \begin{bmatrix} 0 & 1 \\ -\omega_0^2 & -2\zeta\omega_0 \end{bmatrix} = \begin{bmatrix} 1 & \Delta t \\ -\omega_0^2\Delta t & 1 - 2\zeta\omega_0\Delta t \end{bmatrix}$$

and

$$B = \Delta t B_c$$

$$B = \Delta t \begin{bmatrix} 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 0 \\ \Delta t \end{bmatrix}$$

Then choosing Q, I went with:

$$ Q =

$$\begin{bmatrix} 0.0001 & 0 \\ 0 & 0.0001 \end{bmatrix}$$

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

B = np.array(
    [
        [0],
        [dt]
    ]
)

Q = np.array(
    [
        [1e-4,  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 $1 \times 2$ emission matrix $\mathbf{C}$.
- The $1 \times 1$ covariance "matrix" $R$ of the measurement noise.

**Answer:**

Knowing,

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

and,

$$x_j = \begin{bmatrix} x(j\Delta t) \\ \dot{x}(j\Delta t) \end{bmatrix}$$

Then we can solve for $\ddot{x}$,

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

Then we can substitute the state vector,

$$\ddot{x}(j\Delta t) = u_0 \cos(\omega(j\Delta t)) - 2\zeta\omega_0\dot{x}(j\Delta t) - \omega_0^2 x(j\Delta t)$$

Then we can substitute into $y_j$,

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

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

$$\Rightarrow y_j = \begin{bmatrix} -\omega_0^2 & -2\zeta\omega_0 \end{bmatrix} \begin{bmatrix} x(j\Delta t) \\ \dot{x}(j\Delta t) \end{bmatrix} + w_j,$$

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

Since we know the variance, the covariance matrix is

$$\Rightarrow R = \sigma^2$$

In [ ]:
```
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 [ ]:
```
# The number of steps in the trajectory

def plot_kf_estimates(means, covs):
    """Plot estimates of the state with 95% credible intervals."""
    y_labels = ['$x(t)$', '$\dot{x}(t)$']

    dpi = 150

    res_x = 1024
```

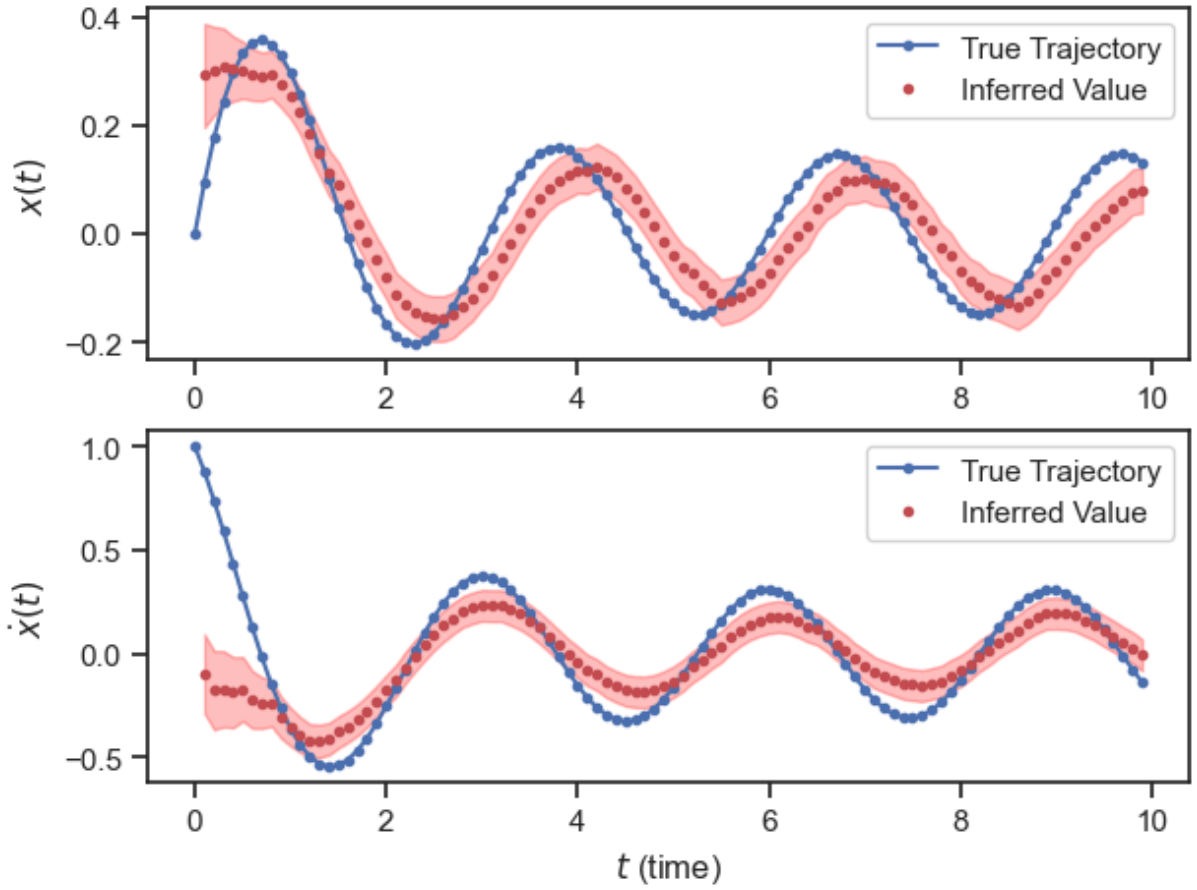
```
    res_y = 768

    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],
            true_trajectory[0:n_steps, j],
            'b.-'
        )
        ax[j].plot(
            times[1:n_steps],
            means[1:n_steps, j],
            'r.'
        )
        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',
            alpha=0.25
        )
        ax[j].legend(['True Trajectory', 'Inferred Value'])
```

In [ ]:

```
# Your answer here (as many code and text blocks as you want)

from filterpy.kalman import KalmanFilter

kf = KalmanFilter(dim_x=2, dim_z=1)

kf.x = np.array([0.0,0.0])
kf.P = 0.1**2
kf.Q = Q
kf.R = R
kf.H = C
kf.F = A
kf.B = B

kalman_filter_states = []
kalman_filter_covar = []
```

```
for i in range(n_steps):
    kf.predict() # updates state and covariance
    measurement = observations[i] + u0 * np.cos(omega * t_eval[i]) # calculates ext
    kf.update(np.array([[measurement]])) # updates based on measurement
    kalman_filter_states.append(kf.x) # saves state
    kalman_filter_covar.append(kf.P) # saves covariance

kalman_filter_states = np.array(kalman_filter_states)
kalman_filter_covar = np.array(kalman_filter_covar)

true_trajectory = np.vstack((sol.y[0], sol.y[1])).T # formats true trajectory into
```

In [ ]:
```
plot_kf_estimates(kalman_filter_states, kalman_filter_covar)
```

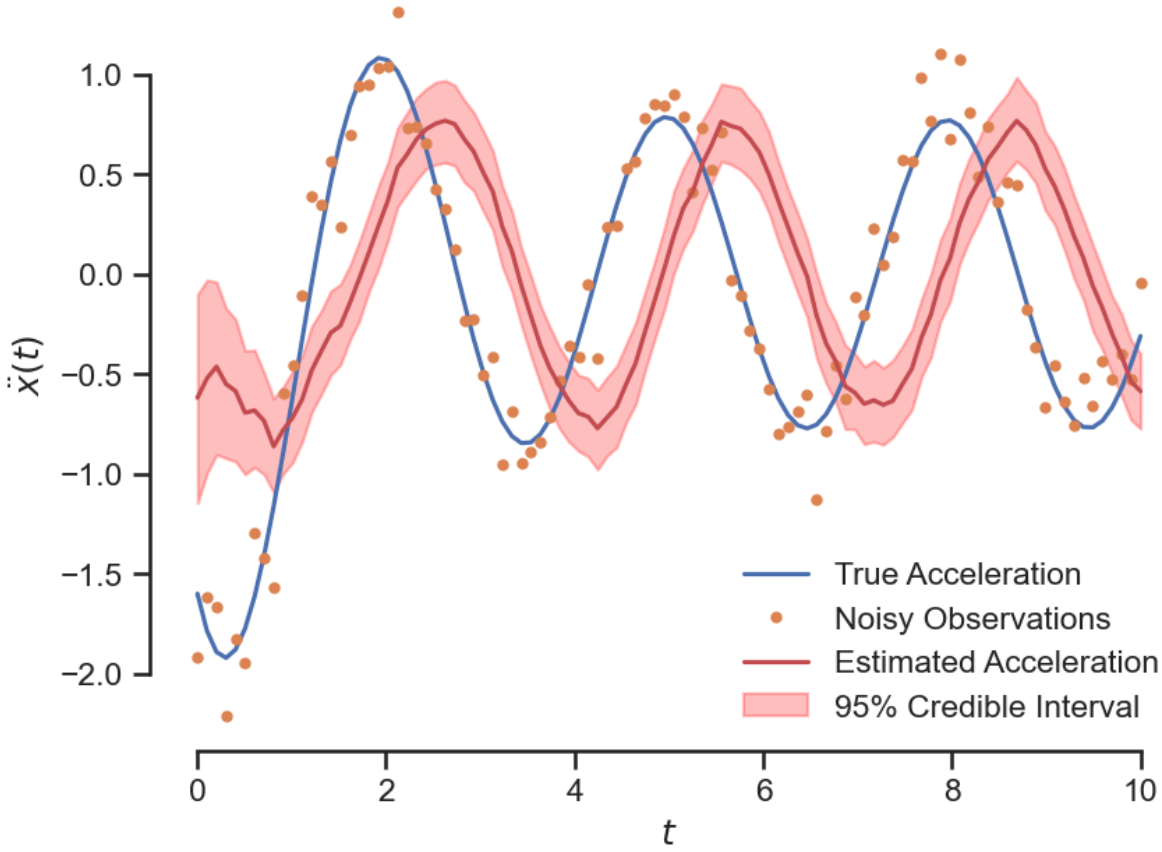



## Part D - Quantify and visualize your uncertainty about the actual acceleration value

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), the true unobserved values, and the noisy measurements.

In [ ]:

```
# Your answer here (as many code and text blocks as you want)

filtered_pos = kalman_filter_states[:, 0]
filtered_vel = kalman_filter_states[:, 1]


filtered_pos_cov = kalman_filter_covar[:, 0, 0]
filtered_vel_cov = kalman_filter_covar[:, 1, 1]



num_samples = 1000
acc_samples = np.zeros((n_steps, num_samples))

for i in range(n_steps):
    pos_samples = np.random.normal(filtered_pos[i], np.sqrt(filtered_pos_cov[i]), n
    vel_samples = np.random.normal(filtered_vel[i], np.sqrt(filtered_vel_cov[i]), n

    acc_samples[i] = -2.0* zeta * omega0 * vel_samples - omega0 ** 2 * pos_samples

mean_acc = np.mean(acc_samples, axis=1)
lower_acc = np.percentile(acc_samples, 2.5, axis=1) # for 95% CI
upper_acc = np.percentile(acc_samples, 97.5, axis=1) # for 95% CI

fig, ax = plt.subplots(dpi=150)
ax.plot(t_eval, true_acc, label='True Acceleration')
ax.plot(t_eval, observations, '.', label='Noisy Observations')
ax.plot(t_eval, mean_acc, 'r-', label='Estimated Acceleration')
ax.fill_between(t_eval, lower_acc, upper_acc, color='red', alpha=0.25, label='95% C
ax.set_xlabel('$t$')
ax.set_ylabel(r'$\ddot{x}(t)$')
plt.legend(loc='best', frameon=False)
sns.despine(trim=True)
```


1.0
0.5
0.0
−0.5
−1.0
−1.5
−2.0
$\ddot{x}(t)$
0
2
4
6
8
10
$t$
True Acceleration
Noisy Observations
Estimated Acceleration
95% Credible Interval