DS-GA 3001.009 Modeling Time Series Data

Week 3 Kalman Filter

In [1]:

```
# Install PyKalman
# pip install pykalman
import numpy as np
import matplotlib.pyplot as plt
from pykalman import KalmanFilter
from scipy.stats import multivariate_normal

# Data Visualiztion
def plot_kalman(x,y,nx,ny,kx=None,ky=None, plot_type="r-", label=None):
    """
    Plot the trajectory
    """
    fig = plt.figure()
    if kx is not None and ky is not None:
        plt.plot(x,y,'g-',nx,ny,'b.',kx,ky, plot_type)
        plt.plot(kx[0], ky[0], 'or')
        plt.plot(kx[-1], ky[-1], 'xr')
    else:
        plt.plot(x,y,'g-',nx,ny,'b.')

    plt.xlabel('X position')
    plt.ylabel('Y position')
    plt.title('Parabola')

    if kx is not None and ky is not None and label is not None:
        plt.legend(('true','measured', label))
    else:
        plt.legend(('true','measured'))

    return fig

def visualize_line_plot(data, xlabel, ylabel, title):
    """
    Function that visualizes a line plot
    """
    plt.plot(data)
    plt.xlabel(xlabel)
    plt.ylabel(ylabel)
    plt.title(title)
    plt.show()

def print_parameters(kf_model, need_params=None):
    """
    Function that prints out the parameters for a Kalman Filter
```

```
    @param - kf_model : the model object
    @param - need_params : a list of string
    """
    if need_params is None:
        need_params = ['transition_matrices', 'observation_matrices', 'transi
                      'observation_offsets', 'transition_covariance',
                      'observation_covariance', 'initial_state_mean', 'initial_st
    for param in need_params:
        print("{0} = {1}, shape = {2}\n".format(param, getattr(kf_model, para
```

Data

We will use a common physics problem with a twist. This example will involve firing a ball from a cannon at a 45-degree angle at a velocity of 100 units/sec. We have a camera that will record the ball's position (pos_x, pos_y) from the side every second. The positions measured from the camera ($\hat{pos}_x, \hat{pos}_y$) have significant measurement error.

Latent Variable $z = [pos_x, pos_y, V_x, V_y]$

Observed Variable $x = [\hat{pos}_x, \hat{pos}_y, \hat{V}_x, \hat{V}_y]$

Reference: http://greg.czerniak.info/guides/kalman1/

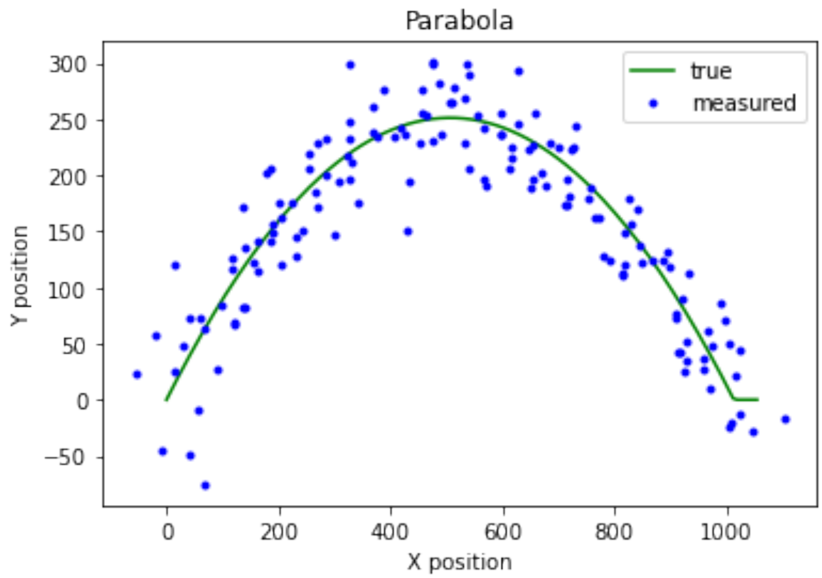
In [2]:

```
# true (latent) trajectory
x = [0, 7.0710678118654755, 14.142135623730951, 21.213203435596427, 28.284271
y = [0, 6.972967811865475, 13.847835623730951, 20.624603435596427, 27.3032712
# observed (noisy) trajectory
nx = [-55.891836789860065, -8.619869715037396, 42.294527931003934, -19.282331
ny = [23.580712916615695, -45.62854499965875, -48.454167220387774, 57.6368259
data = np.array([nx,ny]).T
```

In [3]:

```
print(data.shape)
_ = plot_kalman(x,y,nx,ny);
```

```
(150, 2)
```



Review on Gaussian marginal and conditional distributions

Assume

$$z=[x^Ty^T]^T$$$$z=\begin{bmatrix}x \\y\end{bmatrix}\sim N\left(\begin{bmatrix}a \\b\end{bmatrix}, \begin{bmatrix}A & C \\C^T & B\end{bmatrix}\right)$$
then the marginal distributions are

$$x\sim N(a, A)$$$$y\sim N(b,B)$$
and the conditional distributions are

$$x|y \sim N(a+CB^{-1}(y-b), A-CB^{-1}C^T)$$$$y|x \sim N(b+C^TA^{-1}(x-a), B-C^TA^{-1}C)$$
important take away: given the joint Gaussian distribution we can derive the conditionals

Review on Linear Dynamical System

Latent variable: $$z_n = Az_{n-1}+w$$

Observed variable: $$x_n = Cz_{n}+v$$

Gaussian noise terms: $$w\sim N(0, \Gamma)$$ $$v\sim N(0, \Sigma)$$ $$z_0\sim N(\mu_0, \Gamma_0)$$

As a consequence, z_n, x_n and their joint distributions are Gaussian so we can easily compute the marginals and conditionals.

<img src='img/LDS.svg', width = 300, height=300>

right now n depends only on what was one time step back $n-1$ (Markov chain)

Given the graphical model of the LDS we can write out the joint probability for both temporal sequences:

$$P(\mathbf{z}, \mathbf{x}) = P(z_0)\prod_{n=1...N} P(z_n|z_{n-1}) \prod_{n=0...N} P(x_n|z_{n})$$
all probabilities are implicitely conditioned on the parameters of the model

Kalman

We want to infer the latent variable z_n given the observed variable x_n.

$$P(z_n|x_1, ..., x_n, x_{n+1}, ..., x_N)\sim N(\hat{\mu_n}, \hat{V_n})$$

Forward: Filtering

obtain estimates of latent by running the filtering from $n=0,....N$

prediction given latent space parameters

<img src='img/LDS_latent.svg', width = 110, height=90>

$$z_n^{pred}\sim N(\mu_n^{pred},V_n^{pred})$$$$\mu_n^{pred}=A\mu_{n-1}$$
this is the prediction for z_n obtained simply by taking the expected value of z_{n-1} and projecting it forward one step using the transition probability matrix A

$$V_n^{pred}=AV_{n-1}A^T+\Gamma$$
same for the covariance taking into account the noise covariance $|Gamma$

correction (innovation) from observation

<img src='img/LDS_observed.svg', width = 40, height=80>

project to observational space: $$x_n^{pred}\sim N(C\mu_n^{pred}, CV_n^{pred}C^T+\Sigma)$$

correct prediction by actual data: $$z_n^{innov}\sim N(\mu_n^{innov}, V_n^{innov})$$

$$\mu_n^{innov}=\mu_n^{pred}+K_n(x_n-C\mu_n^{pred})$$$$V_n^{innov}=(I-K_nC)V_n^{pred}$$
Kalman gain matrix: $$K_n=V_n^{pred}C^T(CV_n^{pred}C^T+\Sigma)^{-1}$$

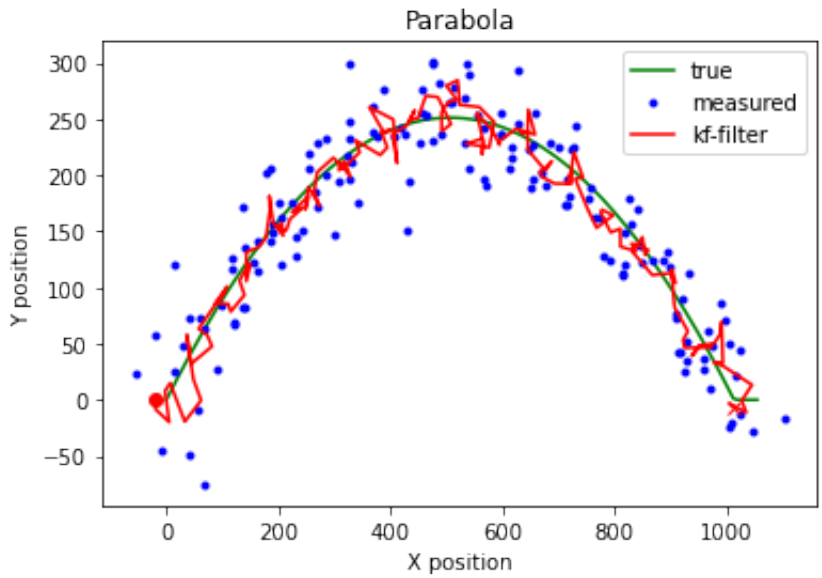
we use the latent-only prediction to project it to the observational space and compute a correction proportional to the error x_n-CAz_{n-1} between prediction and data, coefficient of this correction is the Kalman gain matrix

<img src='img/Kfilter_Bishop.png', width = 600, height=600> from Bishop (2006), chapter 13.3

if measurement noise is small and dynamics are fast -> estimation will depend mostly on observed data

Kalman Filter to predict true (latent) trajectory from observed variable using Pykalman API

In [4]:
```
kf = KalmanFilter(n_dim_state=data.shape[1], n_dim_obs=data.shape[1])
# fit the model (use EM algorithm to estimate the parameters, we will not wor
kf.em(data, n_iter=6)
# Kalman filtering
filtered_state_means, filtered_state_covariances = kf.filter(data)
fig = plot_kalman(x,y,nx,ny, filtered_state_means[:,0], filtered_state_means[
```



Backward: Smoothing

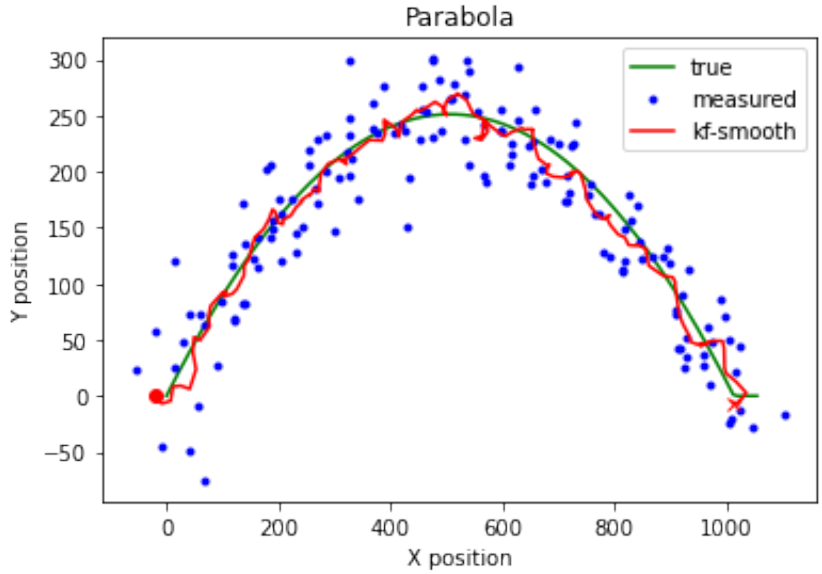
<img src='img/LDS_smooth.svg', width = 110, height=100>

obtain estimates by propagating from x_n back to x_1 using results of forward pass ($\mu_n^{innov}, V_n^{innov}, V_n^{pred}$)

$$N(z_n|\mu_n^{smooth}, V_n^{smooth})$$$$\mu_n^{smooth}=\mu_n^{innov}+J_n(\mu_{n+1}^{smooth}-A\mu_n^{innov})$$$$V_n^{smooth}=V_n^{innov}+J_n(V_{n+1}^{smooth}-V_{n+1}^{pred})J_n^T$$$$J_N=V_n^{innov}A^T (V_{n+1}^{pred})^{-1}$$
This gives us the final estimate for z_n.

$$\hat{\mu_n}=\mu_n^{smooth}$$$$\hat{V_n}=V_n^{smooth}$$

In [5]:
```
# Kalman smoothing
smoothed_state_means, smoothed_state_covariances = kf.smooth(data)
fig = plot_kalman(x,y,nx,ny, smoothed_state_means[:,0], smoothed_state_means[
```



Kalman Filter Implementation

In this part of the exercise, you will implement the Kalman filter. Specifically, you need to implement the following method:

- filter: assume learned parameters, perform the forward calculation
- smooth: assume learned parameters, perform both the forward and backward calculation

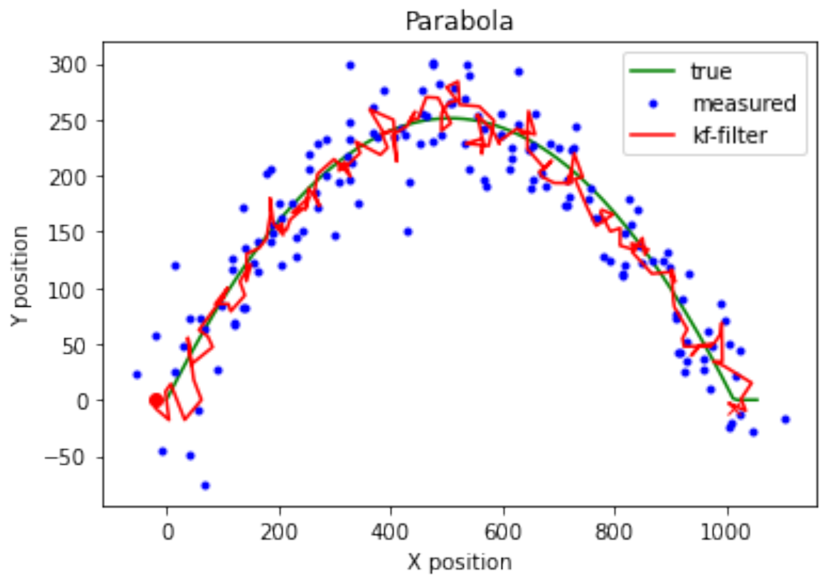
In [6]:
```
class MyKalmanFilter:
    """
    Class that implements the Kalman Filter
    """
    def __init__(self, n_dim_state=2, n_dim_obs=2):
        """
        @param n_dim_state: dimension of the laten variables
        @param n_dim_obs: dimension of the observed variables
        """
        self.n_dim_state = n_dim_state
        self.n_dim_obs = n_dim_obs
        self.transition_matrices = np.eye(n_dim_state)
        self.transition_offsets = np.zeros(n_dim_state) # you can ignore this
        self.transition_covariance = np.eye(n_dim_state)
        self.observation_matrices = np.eye(n_dim_obs, n_dim_state)
        self.observation_covariance = np.eye(n_dim_obs)
        self.observation_offsets = np.zeros(n_dim_obs) # you can ignore this
        self.initial_state_mean = np.zeros(n_dim_state)
        self.initial_state_covariance = np.eye(n_dim_state)
```

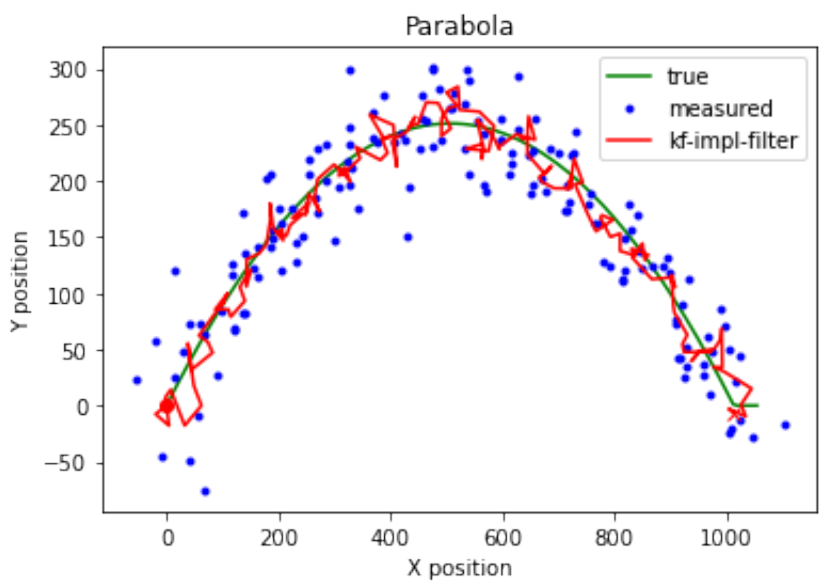
```python
    def filter(self, X):
        """
        Method that performs Kalman filtering
        @param X: a numpy 2D array whose dimension is [n_example, self.n_dim_
        @output: filtered_state_means: a numpy 2D array whose dimension is [n_
        @output: filtered_state_covariances: a numpy 3D array whose dimension
        """

        # validate inputs
        n_example, observed_dim = X.shape
        assert observed_dim==self.n_dim_obs

        # create holders for outputs
        filtered_state_means = np.zeros( (n_example, self.n_dim_state) )
        filtered_state_covariances = np.zeros( (n_example, self.n_dim_state,

        #############################
        # TODO: implement filtering #
        #############################

        V_previous = self.initial_state_covariance
        mean_pre = self.initial_state_mean
        A = self.transition_matrices
        C = self.observation_matrices

        for i in range(1,n_example):
            Vn = np.matmul(np.matmul(A,V_previous),A.T)
            Vn_pred = Vn + self.transition_covariance
            mean_n_pred = np.matmul(A,mean_pre)

            temp = np.linalg.inv(np.matmul(np.matmul(C,Vn_pred),C.T)+self.obs
            K = np.matmul(np.matmul(Vn_pred,C.T),temp)

            mean_inno = mean_n_pred + np.matmul(K,(X[i,:]-np.matmul(C,mean_n_
            filtered_state_means[i,:] = mean_inno
            V_inno = np.matmul((np.eye(observed_dim) - np.matmul(K,C)),Vn_pre
            filtered_state_covariances[i,:,:] = V_inno

            V_previous = V_inno
            mean_pre = mean_inno


        return filtered_state_means, filtered_state_covariances

    def smooth(self, X):
        """
        Method that performs the Kalman Smoothing
        @param X: a numpy 2D array whose dimension is [n_example, self.n_dim_
        @output: smoothed_state_means: a numpy 2D array whose dimension is [n_
        @output: smoothed_state_covariances: a numpy 3D array whose dimension
        """
        # TODO: implement smoothing
```

```
        # validate inputs
        n_example, observed_dim = X.shape
        assert observed_dim==self.n_dim_obs

        # run the forward path
        mu_list, v_list = self.filter(X)

        # create holders for outputs
        smoothed_state_means = np.zeros( (n_example, self.n_dim_state) )
        smoothed_state_covariances = np.zeros( (n_example, self.n_dim_state,

        #############################
        # TODO: implement smoothing #
        #############################
        A = self.transition_matrices
        C = self.observation_matrices
        mu_pre = mu_list[-1]
        v_pre = v_list[-1]

        for i in range(n_example-2, -1, -1):

            V_pred = np.matmul(np.matmul(A, v_list[i]), A.T) + self.transitio
            J_N = np.matmul(np.matmul(v_list[i], A.T), np.linalg.inv(V_pred))


            mean_smooth = mu_list[i] + np.matmul(J_N, (mu_pre - np.matmul(A, 
            V_smooth = v_list[i] + np.matmul(np.matmul(J_N, ( v_pre - V_pred)

            smoothed_state_means[i] = mean_smooth
            smoothed_state_covariances[i] = V_smooth
            mu_pre = mean_smooth
            v_pre = V_smooth

        smoothed_state_means[-1] = mu_list[-1]
        smoothed_state_covariances[-1] = v_list[-1]




        return smoothed_state_means, smoothed_state_covariances

    def import_param(self, kf_model):
        """
        Method that copies parameters from a trained Kalman Model
        @param kf_model: a Pykalman object
        """
        need_params = ['transition_matrices', 'observation_matrices', 'transi
                   'observation_offsets', 'transition_covariance',
                   'observation_covariance', 'initial_state_mean', 'initial_st
        for param in need_params:
            setattr(self, param, getattr(kf_model, param))
```

Filtering

In [7]:
```
kf = KalmanFilter(n_dim_state=data.shape[1], n_dim_obs=data.shape[1])
kf.em(data)
my_kf = MyKalmanFilter(n_dim_state=data.shape[1], n_dim_obs=data.shape[1])
my_kf.import_param(kf)
filtered_state_means, filtered_state_covariances = kf.filter(data)
filtered_state_means_impl, filtered_state_covariances_impl = my_kf.filter(dat
_ = plot_kalman(x,y,nx,ny, filtered_state_means[:,0], filtered_state_means[:,
_ = plot_kalman(x,y,nx,ny, filtered_state_means_impl[:,0], filtered_state_mea
```

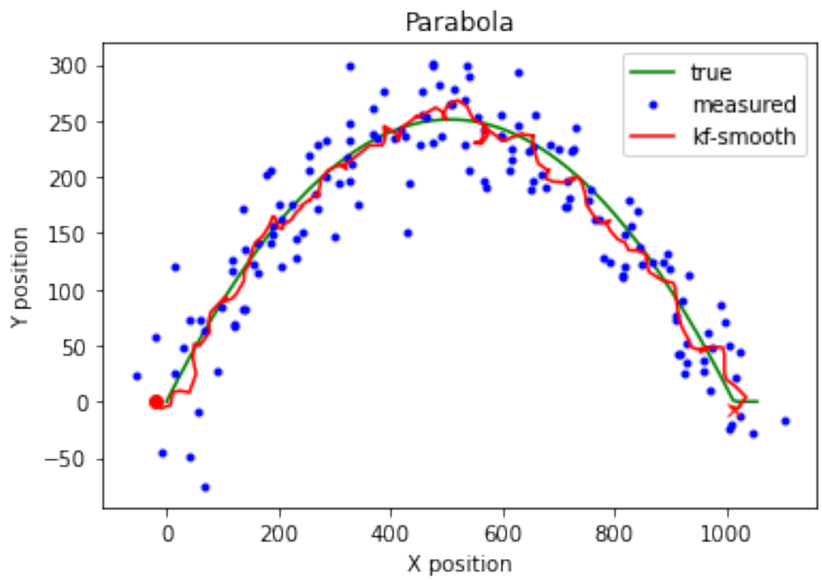


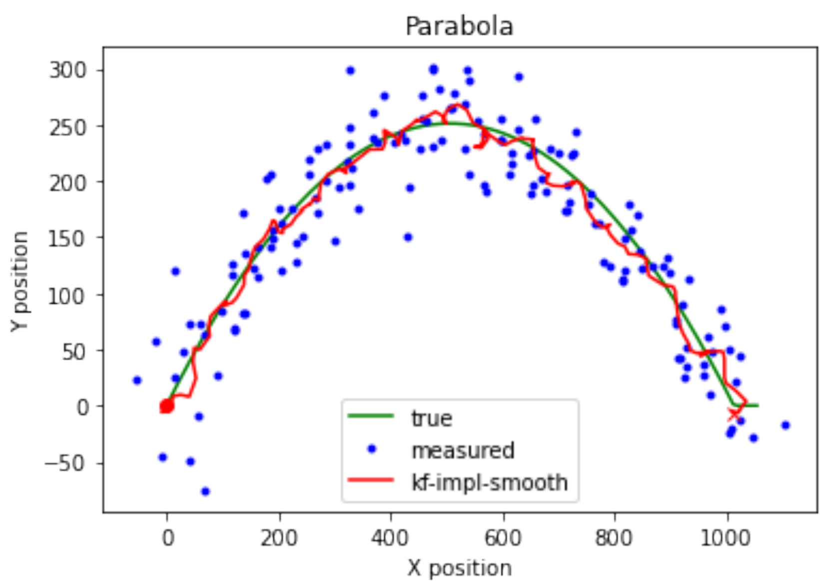


Smoothing

In [8]:

```
kf = KalmanFilter(n_dim_state=data.shape[1], n_dim_obs=data.shape[1])
kf.em(data)
my_kf = MyKalmanFilter(n_dim_state=data.shape[1], n_dim_obs=data.shape[1])
my_kf.import_param(kf)
smoothed_state_means, smoothed_state_covariances = kf.smooth(data)
smoothed_state_means_impl, smoothed_state_covariances_impl = my_kf.smooth(dat
fig = plot_kalman(x,y,nx,ny, smoothed_state_means[:,0], smoothed_state_means[
fig = plot_kalman(x,y,nx,ny, smoothed_state_means_impl[:,0], smoothed_state_m
```





Please turn in the code before 10/06/2020 11:59 pm EST. Please name your notebook netid.ipynb.

Your work will be evaluated based on the code and plots. You don't need to write down your answers to these questions in the text blocks.

In [ ]:

```
```

In [ ]:

```
```