EE16A: Homework 2

Problem 1: Image Stitching

This section of the notebook continues the image stitching problem. Be sure to have a `figures` folder in the same directory as the notebook. The `figures` folder should contain the files:

```
Berkeley_banner_1.jpg
Berkeley_banner_2.jpg
stacked_pieces.jpg
lefthalfpic.jpg
righthalfpic.jpg
```

Note: This structure is present in the provided HW2 zip file.

Run the next block of code before proceeding

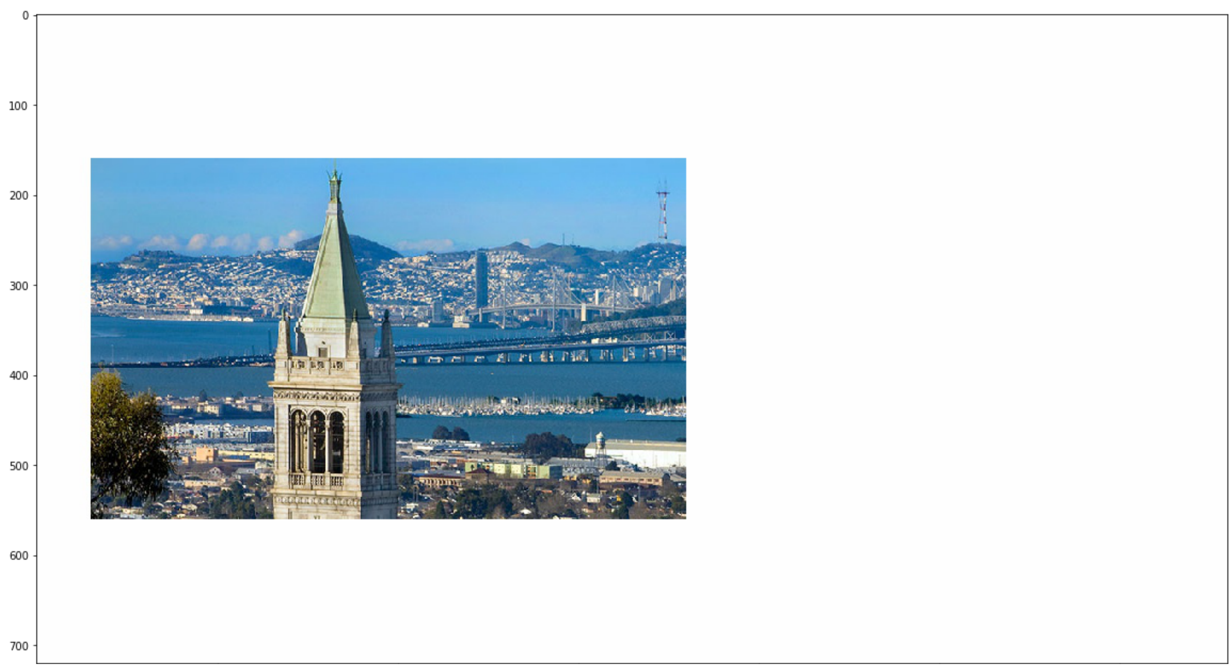
In [3]:
```
import numpy as np
import numpy.matlib
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
from numpy import pi, cos, exp, sin
import matplotlib.image as mpimg
import matplotlib.transforms as mtransforms


#%matplotlib inline

#loading images
image1=mpimg.imread('figures/Berkeley_banner_1.jpg')
image1=image1/255.0
image2=mpimg.imread('figures/Berkeley_banner_2.jpg')
image2=image2/255.0
image_stack=mpimg.imread('figures/stacked_pieces.jpg')
image_stack=image_stack/255.0


image1_marked=mpimg.imread('figures/lefthalfpic.jpg')
image1_marked=image1_marked/255.0
image2_marked=mpimg.imread('figures/righthalfpic.jpg')
image2_marked=image2_marked/255.0

def euclidean_transform_2to1(transform_mat,translation,image,position,LL,UL
    new_position=np.round(transform_mat.dot(position)+translation)
    new_position=new_position.astype(int)


    if (new_position>=LL).all() and (new_position<UL).all():
        values=image[new_position[0][0],new_position[1][0],:]
    else:
        values=np.array([2.0,2.0,2.0])

    return values

def euclidean_transform_1to2(transform_mat,translation,image,position,LL,UL
    transform_mat_inv=np.linalg.inv(transform_mat)
    new_position=np.round(transform_mat_inv.dot(position-translation))
    new_position=new_position.astype(int)

    if (new_position>=LL).all() and (new_position<UL).all():
        values=image[new_position[0][0],new_position[1][0],:]
    else:
        values=np.array([2.0,2.0,2.0])

    return values

def solve(A,b):
    try:
        z = np.linalg.solve(A,b)
    except:
        raise ValueError('Rows are not linearly independent. Cannot solve s
    return z
```

We will stick to a simple example and just consider stitching two images (if you can stitch two pictures, then you could conceivably stitch more by applying the same technique over and over again).

Daniel decided to take an amazing picture of the Campanile overlooking the bay. Unfortunately, the field of view of his camera was not large enough to capture the entire scene, so he decided to take two pictues and stitch them together.

The next block will display the two images.

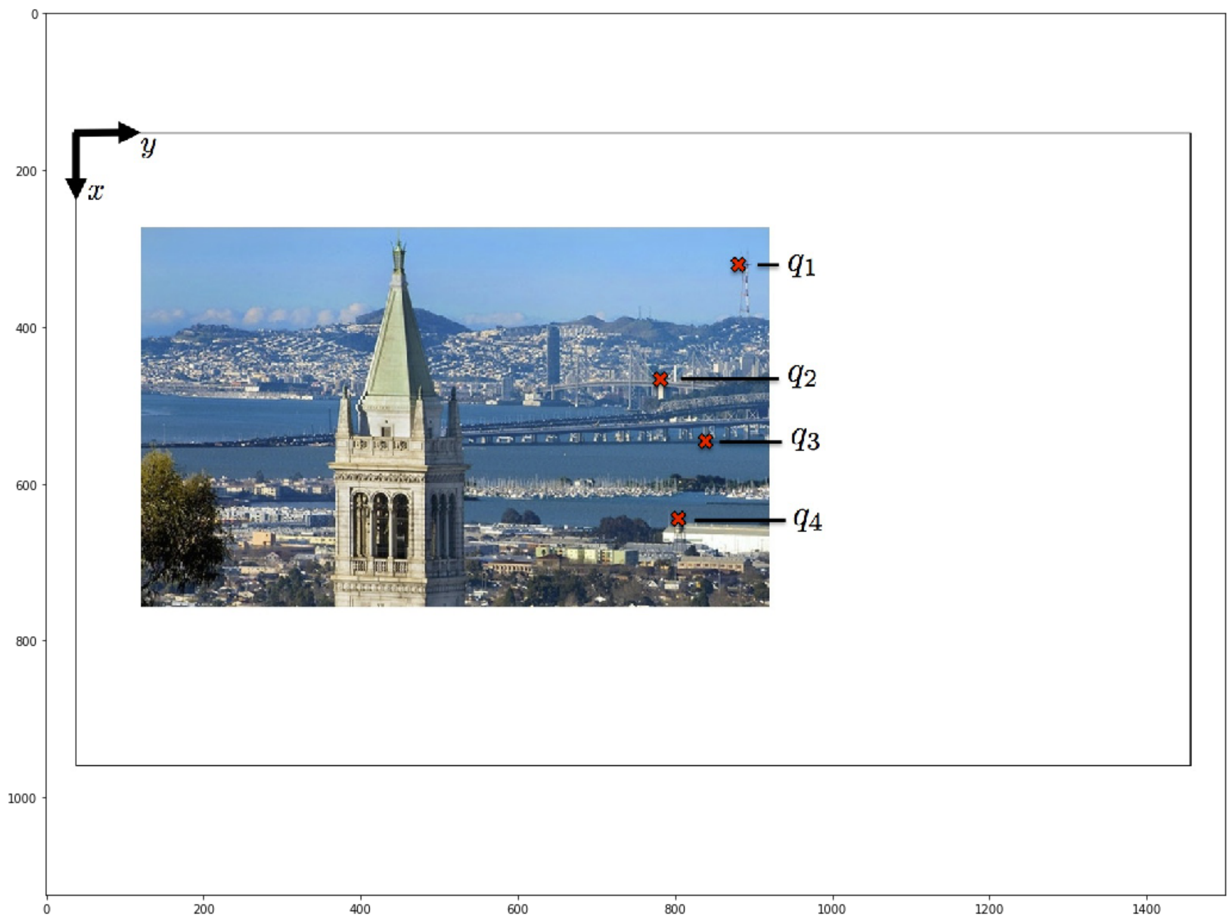
In [4]:
```
plt.figure(figsize=(20,40))

plt.subplot(311)
plt.imshow(image1)

plt.subplot(312)
plt.imshow(image2)

plt.subplot(313)
plt.imshow(image_stack)

plt.show()
```

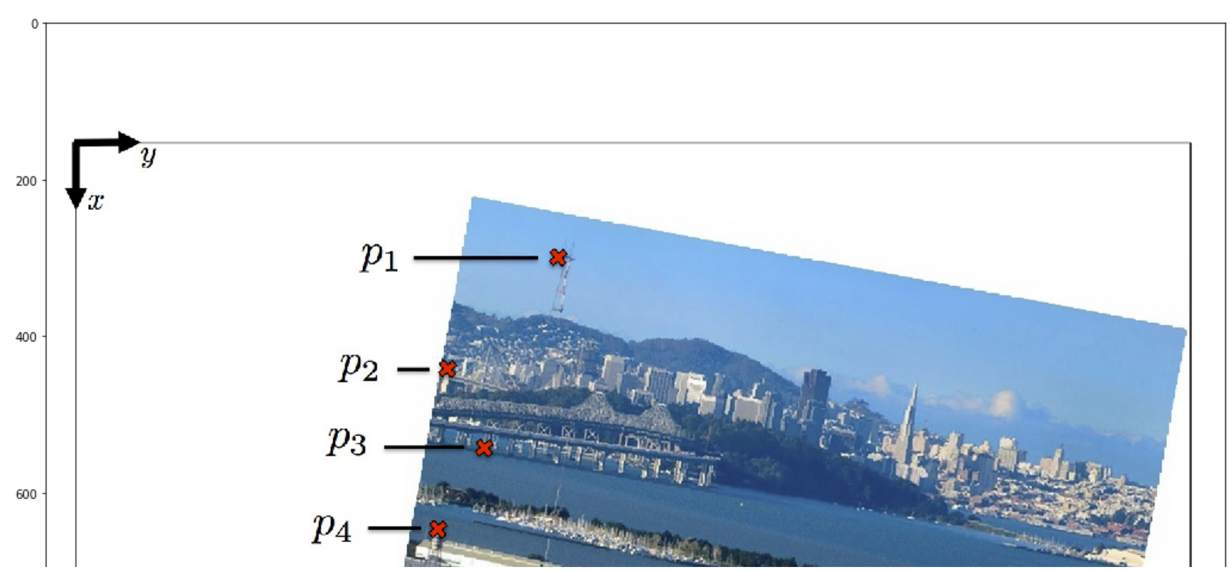


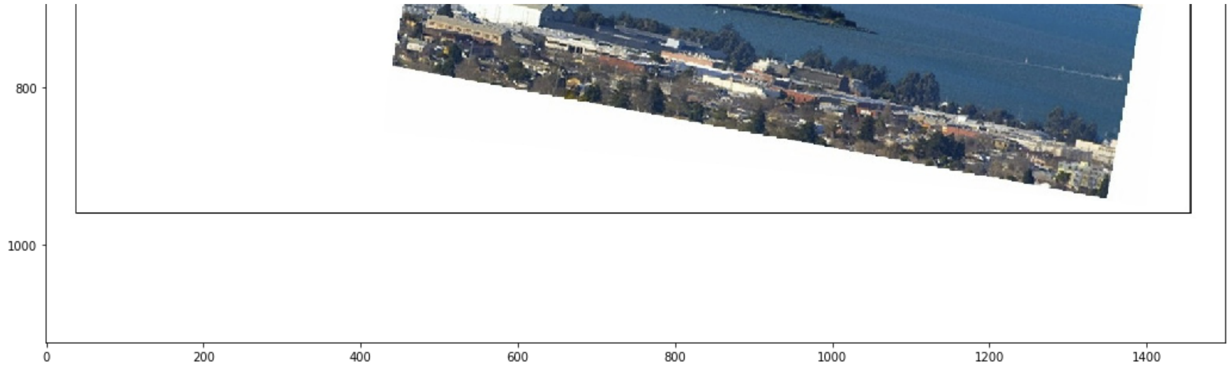
Once you apply Marcela's algorithm on the two images you get the following result (run the next block):

In [5]:
```
plt.figure(figsize=(20,30))

plt.subplot(211)
plt.imshow(image1_marked)

plt.subplot(212)
plt.imshow(image2_marked)
```

Out[5]: <matplotlib.image.AxesImage at 0x128c587b8>







As you can see Marcela's algorithm was able to find four common points between the two images. These points expressed in the coordinates of the first image and second image are

$$\vec{p_1} = \begin{bmatrix} 200 \\ 700 \end{bmatrix} \quad \vec{p_2} = \begin{bmatrix} 310 \\ 620 \end{bmatrix} \quad \vec{p_3} = \begin{bmatrix} 390 \\ 660 \end{bmatrix} \quad \vec{p_4} = \begin{bmatrix} 46(\\ 63(\end{bmatrix}$$

$$\vec{q_1} = \begin{bmatrix} 162.2976 \\ 565.8862 \end{bmatrix} \quad \vec{q_2} = \begin{bmatrix} 285.4283 \\ 458.7469 \end{bmatrix} \quad \vec{q_3} = \begin{bmatrix} 385.2465 \\ 498.1973 \end{bmatrix} \quad \vec{q_4} = \begin{bmatrix} 465.789 \\ 455.013 \end{bmatrix}$$

It should be noted that in relation to the image the positive x-axis is down and the positive y-axis is right. This will have no bearing as to how you solve the problem, however it helps in interpreting what the numbers mean relative to the image you are seeing.

Using the points determine the parameters $R_{11}, R_{12}, R_{21}, R_{22}, T_x, T_y$ that map the points from the first image to the points in the second image by solving an appropriate system of equations. Hint: you do not need all the points to recover the parameters.

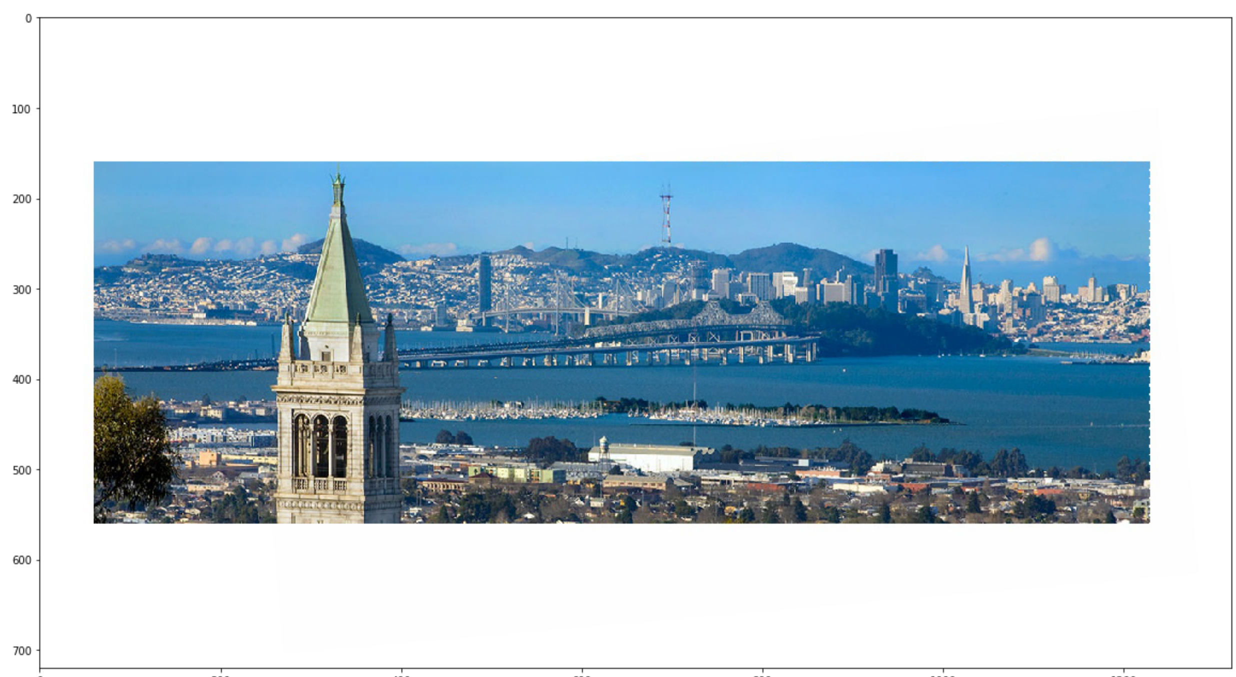
In [6]:
```
# Note that the following is a general template for solving for 6 unknowns
# You do not have to use the following code exactly.
# All you need to do is to find parameters R_11, R_12, R_21, R_22, T_x, T_y
# If you prefer finding them another way it is fine.

# fill in the entries
A = np.array([[200,700,0,0,1,0],
              [0,0,200,700,0,1],
              [310,620,0,0,1,0],
              [0,0,310,620,0,1],
              [390,660,0,0,1,0],
              [0,0,390,660,0,1]])

# fill in the entries
b = np.array([[162.2976],[565.8862],[285.4283],[458.7469],[385.2465],[498.1

A = A.astype(float)
b = b.astype(float)

# solve the linear system for the coefficiens
z = solve(A,b)

#Parameters for our transformation
R_11 = z[0,0]
R_12 = z[1,0]
R_21 = z[2,0]
R_22 = z[3,0]
T_x  = z[4,0]
T_y  = z[5,0]
```

Stitch the images using the transformation you found by running the code below.

Note that it takes about 40 seconds for the block to finish running on a modern laptop.

In [7]:
```
matrix_transform=np.array([[R_11,R_12],[R_21,R_22]])
translation=np.array([T_x,T_y])

#Creating image canvas (the image will be constructed on this)
num_row,num_col,blah=image1.shape
image_rec=1.0*np.ones((int(num_row),int(num_col),3))

#Reconstructing the original image

LL=np.array([[0],[0]]) #lower limit on image domain
UL=np.array([[num_row],[num_col]]) #upper limit on image domain

for row in range(0,int(num_row)):
    for col in range(0,int(num_col)):
        #notice that the position is in terms of x and y, so the c
        position=np.array([[row],[col]])
        if image1[row,col,0] > 0.995 and image1[row,col,1] > 0.995 and imag
            temp = euclidean_transform_2to1(matrix_transform,translation,im
            image_rec[row,col,:] = temp
        else:
            image_rec[row,col,:] = image1[row,col,:]


plt.figure(figsize=(20,20))
plt.imshow(image_rec)
plt.axis('on')
plt.show()
```

```
Clipping input data to the valid range for imshow with RGB data ([0..1] f
or floats or [0..255] for integers).
```



Part E: Failure Mode Points

$$\vec{p_1} = \begin{bmatrix} 390 \\ 660 \end{bmatrix} \qquad \vec{p_2} = \begin{bmatrix} 425 \\ 645 \end{bmatrix} \qquad \vec{p_3} = \begin{bmatrix} 460 \\ 630 \end{bmatrix}$$

$$\vec{q_1} = \begin{bmatrix} 385 \\ 450 \end{bmatrix} \qquad \vec{q_2} = \begin{bmatrix} 425 \\ 480 \end{bmatrix} \qquad \vec{q_3} = \begin{bmatrix} 465 \\ 510 \end{bmatrix}$$

In [8]:
```
# Note that the following is a general template for solving for 6 unknowns
# You do not have to use the following code exactly.
# All you need to do is to find parameters R_11, R_12, R_21, R_22, T_x, T_y
# If you prefer finding them another way it is fine.

# fill in the entries
A = np.array([[390,660,0,0,1,0],
              [0,0,390,660,0,1],
              [425,645,0,0,1,0],
              [0,0,425,645,0,1],
              [460,630,0,0,1,0],
              [0,0,460,625,0,1]])

# fill in the entries
b = np.array([[385],[450],[425],[480],[465],[510]])

A = A.astype(float)
b = b.astype(float)

# solve the linear system for the coefficiens
z = solve(A,b)

#Parameters for our transformation
R_11 = z[0,0]
R_12 = z[1,0]
R_21 = z[2,0]
R_22 = z[3,0]
T_x  = z[4,0]
T_y  = z[5,0]
```

```
---------------------------------------------------------------------------
LinAlgError                               Traceback (most recent call las
t)
<ipython-input-3-12ed8790aca9> in solve(A, b)
     51     try:
---> 52         z = np.linalg.solve(A,b)
     53     except:

~/anaconda3/lib/python3.7/site-packages/numpy/linalg/linalg.py in solve
(a, b)
    393     extobj = get_linalg_error_extobj(_raise_linalgerror_singular)
--> 394     r = gufunc(a, b, signature=signature, extobj=extobj)
    395

~/anaconda3/lib/python3.7/site-packages/numpy/linalg/linalg.py in _raise_
linalgerror_singular(err, flag)
     88 def _raise_linalgerror_singular(err, flag):
---> 89     raise LinAlgError("Singular matrix")
     90

LinAlgError: Singular matrix

During handling of the above exception, another exception occurred:

ValueError                                Traceback (most recent call las
t)
```

```
<ipython-input-8-13045444f49b> in <module>
     19 
     20 # solve the linear system for the coefficiens
---> 21 z = solve(A,b)
     22 
     23 #Parameters for our transformation

<ipython-input-3-12ed8790aca9> in solve(A, b)
     52         z = np.linalg.solve(A,b)
     53     except:
---> 54         raise ValueError('Rows are not linearly independent. Cann
ot solve system of linear equations uniquely. :)')
     55     return z

ValueError: Rows are not linearly independent. Cannot solve system of lin
ear equations uniquely. :)
```

Problem 2: Kinematic Model for a Simple Car

This script helps to visualize the difference between a nonlinear model and a corresponding linear approximation for a simple car. What you should notice is that the linear model is similar to the nonlinear model when you are close to the point where the approximation is made.

First, run the following block to set up the helper functions needed to simulate the vehicle models and plot the trajectories taken.

In [11]:
```
# DO NOT MODIFY THIS BLOCK!
''' Problem/Model Setup'''
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline

# Vehicle Model Constants
L = 1.0  # length of the car, meters
dt = 0.1 # time difference between timestep (k+1) and timestep k, seconds

''' Nonlinear Vehicle Model Update Equation '''
def nonlinear_vehicle_model(initial_state, inputs, num_steps):
    x     = initial_state[0] # x position, meters
    y     = initial_state[1] # y position, meters
    theta = initial_state[2] # heading (wrt x-axis), radians
    v     = initial_state[3] # speed, meters per second

    a = inputs[0]            # acceleration, meters per second squared
    phi = inputs[1]          # steering angle, radians

    state_history = []       # array to hold state values as the time step
    state_history.append([x,y,theta,v]) # add the initial state (i.e. k = 0

    for i in range(0, num_steps):
        # Find the next state, at time k+1, by applying the nonlinear model
        x_next     = x     + v * np.cos(theta) * dt
        y_next     = y     + v * np.sin(theta) * dt
        theta_next = theta + v/L * np.tan(phi) * dt
        v_next     = v     + a * dt

        # Add the next state to the history.
        state_history.append([x_next,y_next,theta_next,v_next])

        # Advance to the next state, at time k+1, to get ready for next loo
        x = x_next
        y = y_next
        theta = theta_next
        v = v_next

    return np.array(state_history)

''' Linear Vehicle Model Update Equation '''
def linear_vehicle_model(A, B, initial_state, inputs, num_steps):
    # Note: A should be a 4x4 matrix, B should be a 4x2 matrix for this lin

    x     = initial_state[0] # x position, meters
    y     = initial_state[1] # y position, meters
    theta = initial_state[2] # heading (wrt x-axis), radians
    v     = initial_state[3] # speed, meters per second

    a = inputs[0]            # acceleration, meters per second squared
    phi = inputs[1]          # steering angle, radians

    state_history = []       # array to hold state values as the time step
    state_history.append([x,y,theta,v]) # add the initial state (i.e. k = 0
```

```
    for i in range(0, num_steps):
        # Find the next state, at time k+1, by applying the nonlinear model
        state_next = np.dot(A, state_history[-1]) + np.dot(B, inputs)

        # Add the next state to the history.
        state_history.append(state_next)

        # Advance to the next state, at time k+1, to get ready for next loc
        state = state_next

    return np.array(state_history)

''' Plotting Setup'''
def make_model_comparison_plot(state_predictions_nonlinear, state_predictic
    f = plt.figure()
    plt.plot(state_predictions_nonlinear[0,0], state_predictions_nonlinear[
    plt.plot(state_predictions_nonlinear[:,0], state_predictions_nonlinear[
    plt.plot(state_predictions_linear[:,0], state_predictions_linear[:,1],
    plt.legend(loc='upper left')
    plt.xlim([4, 8])
    plt.ylim([9, 12])
    plt.show()
```

Part B

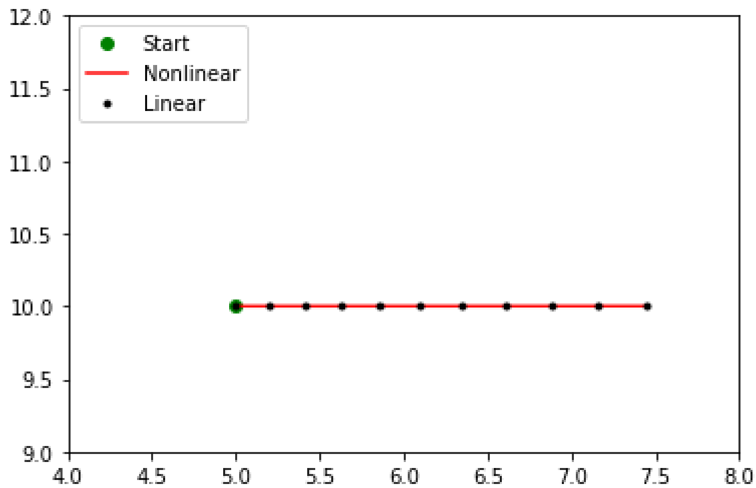
Task: Fill in the matrices A and B for the linear system approximating the nonlinear vehicle model under small heading and steering angle approximations.

In [12]:
```
# Your code here.
A = np.array([[1, 0, 0, 0.1],
              [0, 1, 0, 0],
              [0, 0, 1, 0],
              [0, 0, 0, 1]])

B = np.array([[ 0, 0],
              [ 0, 0],
              [ 0, 0],
              [ 0.1, 0]])
```

Part C

Task: Fill out the state and input values from Part C and look at the resulting plot. The plot should help you to visualize the difference between using a linear model and a nonlinear model for this specific starting state and input.

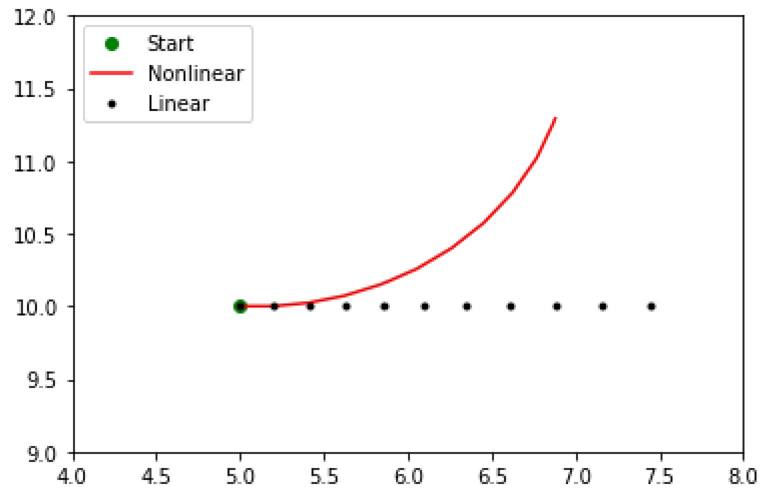
In [13]:
```
# Your code here.
x_init  = 5.0
y_init  = 10.0
theta_init = 0.0
v_init     = 2.0
a_input    = 1.0
phi_input  = 0.0001

state_init = [x_init, y_init, theta_init, v_init]
state_predictions_nonlinear = nonlinear_vehicle_model(state_init, [a_input,
state_predictions_linear = linear_vehicle_model(A, B, state_init, [a_input,

make_model_comparison_plot(state_predictions_nonlinear, state_predictions_l
```



Part D

Task: Fill out the state and input values from Problem D and look at the resulting plot. The plot should help you to visualize the difference between using a linear model and a nonlinear model for this specific starting state and input.

In [14]:
```
# Your code here.
x_init  = 5.0
y_init  = 10.0
theta_init = 0.0
v_init     = 2.0
a_input    = 1.0
phi_input  = 0.5

state_init = [x_init, y_init, theta_init, v_init]
state_predictions_nonlinear = nonlinear_vehicle_model(state_init, [a_input,
state_predictions_linear = linear_vehicle_model(A, B, state_init, [a_input,

make_model_comparison_plot(state_predictions_nonlinear, state_predictions_l
print(state_predictions_nonlinear[10])
print(state_predictions_linear[10])
```




```
[ 6.87984693 11.28998941  1.3384411   3.        ]
[ 7.45 10.    0.    3.  ]
```

In [ ]:

4 a. No, consider the following:

$G_1, G_2, \cdots G_6 = x, 0, x, 0, x, 0$

and

$G_1, G_2, \cdots G_6 = 0, x, 0, x, 0, x$

In both cases, P_{1-6} each contains $\frac{x}{2}$.

b. This problem can be expressed:

$$\begin{bmatrix} 1&1&0&0&0 \\ 0&1&1&0&0 \\ 0&0&1&1&0 \\ 0&0&0&1&1 \\ 1&0&0&0&1 \end{bmatrix} \begin{bmatrix} T_1 \\ T_2 \\ T_3 \\ T_4 \\ T_5 \end{bmatrix} = \begin{bmatrix} 2P_1 \\ 2P_2 \\ 2P_3 \\ 2P_4 \\ 2P_5 \end{bmatrix}$$

The reduced echelon form is

$$\begin{bmatrix} 1&0&0&0&0 \\ 0&1&0&0&0 \\ 0&0&1&0&0 \\ 0&0&0&1&0 \\ 0&0&0&0&1 \end{bmatrix} \begin{bmatrix} T_1 \\ T_2 \\ T_3 \\ T_4 \\ T_5 \end{bmatrix} = \begin{bmatrix} P_1+P_3+P_5-P_2-P_4 \\ P_1+P_2-P_3+P_4-P_5 \\ P_2-P_1+P_3-P_4+P_5 \\ P_1-P_2+P_3+P_4-P_5 \\ P_5-P_1+P_2+P_4-P_3 \end{bmatrix}$$

These are unique solutions, so it is possible to find the tip for each person.

c. If n is even, it is not possible. If n is odd, it is possible:

For all even n, the pattern from part a can exist where each tip alternates between x and 0. For all odd n, the matrix from b can be created but for an nxn matrix. This matrix has a reduced echelon form and can thus give unique solutions.

5. Let A be the matrix:

$$\begin{bmatrix} A_{11}, A_{12}, \cdots & A_{1n} \\ A_{21}, A_{22}, \cdots & A_{2n} \\ \vdots \quad \ddots & \vdots \\ A_{n1}, A_{n2} \cdots & A_{nn} \end{bmatrix}$$

then, each component in the set $\{A\vec{v}_1, A\vec{v}_2 \cdots A\vec{v}_k\}$ can be expressed by the dot product between $\vec{v}_i$ and A. In other words, each vector $\vec{v}_i$ is modified by the same corresponding components in A. Since scalar multiples of a linearly dependent vector are also linearly dependent, the resulting set $\{A\vec{v}_1, A\vec{v}_2 \cdots A\vec{v}_k\}$ is also all linearly dependent.

7. I worked with Nicole Wang ID: 3034013094

I did a problem a day throughout the week.

c) the trajectories are similar because the steering angle $\varphi(k)$ is very small, relative to the other variables.

d) The trajectories are very different because of the high steering angle $\varphi(k)$.

3 a) i. Yes, there is a copy of the element lost, so Bob still receives all 3 elements.

ii. $\begin{bmatrix} ? \\ b \\ c \end{bmatrix}$ $\begin{bmatrix} \cancel{a}\,? \\ b \\ c \\ ? \\ b \\ c \end{bmatrix}$

iii. $\begin{bmatrix} a \\ ? \\ c \end{bmatrix}$ $\begin{bmatrix} a \\ ? \\ c \\ a \\ ? \\ c \end{bmatrix}$

iv. $\begin{bmatrix} a \\ b \\ ? \end{bmatrix}$ $\begin{bmatrix} a \\ b \\ ? \\ a \\ b \\ ? \end{bmatrix}$

b) $\vec{k} = \begin{bmatrix} \alpha_1 a + \beta_1 b + \gamma_1 c \\ \alpha_2 a + \beta_2 b + \gamma_2 c \\ \alpha_3 a + \beta_3 b + \gamma_3 c \\ \alpha_4 a + \beta_4 b + \gamma_4 c \\ \alpha_5 a + \beta_5 b + \gamma_5 c \\ \alpha_6 a + \beta_6 b + \gamma_6 c \end{bmatrix}$

c) $v_1 = [1\ 0\ 0] = v_4$
$v_2 = [0\ 1\ 0] = v_5$
$v_3 = [0\ 0\ 1] = v_6$

d) $\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} a \\ b \\ c \end{bmatrix} = \begin{bmatrix} 7 \\ ? \\ ? \\ 3 \\ 4 \\ ? \\ ? \end{bmatrix}$

1. $a = 7$
2. $b = ?$
3. $c = ?$
4. $a + b = 3$
5. $a + c = 4$
6. $b + c = ?$
7. $a + b + c = ?$

$\Rightarrow a = 7,\ a + b = 3,\ a + c = 4$

$\Rightarrow a = 7,\ b = -4,\ c = -3$

Message: $\begin{bmatrix} 7 \\ -4 \\ -3 \end{bmatrix}$

e) i. Yes, because any combination of 4 different k_i's depend on all variables a, b, c.

ii. No. If ~~[illegible]~~ the symbols lost are k_3, k_5, k_7, then there are only 3 equations that are linearly dependent. He can only recover the message if the remaining 3 equations are linearly independent.

f) part d because the $\vec{v}_4$ and $\vec{v}_5$ etc. from strategy d allow for the message to be decoded ~~whether the first~~ in the case that the message lost was from 2 possibilities rather than 1.

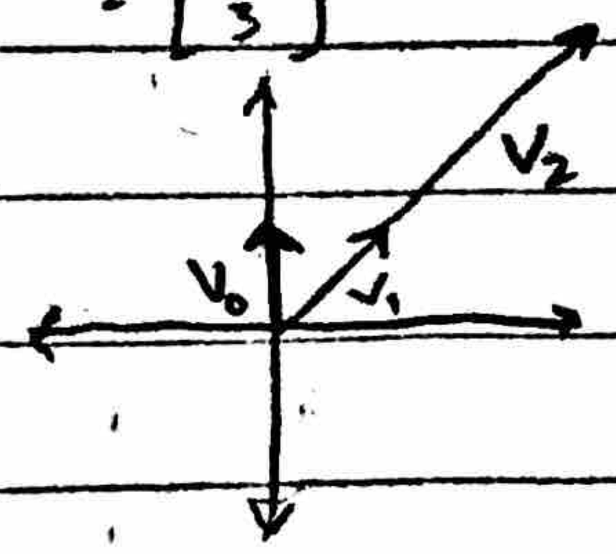
William Wang

EE 16A HW02

1a) $\vec{v}_2 = \begin{bmatrix} 2 & 2 \\ -2 & 2 \end{bmatrix}\begin{bmatrix} 0 \\ 1 \end{bmatrix} + \begin{bmatrix} 1 \\ 1 \end{bmatrix}$

$= \begin{bmatrix} 2 \\ 2 \end{bmatrix} + \begin{bmatrix} 1 \\ 1 \end{bmatrix}$

$= \begin{bmatrix} 3 \\ 3 \end{bmatrix}$



$\vec{v}_2$ is a $-45°$ rotation and scaled (3x) version of $\vec{v}_0$

b) Known: q_x, q_y, p_x, p_y

Unknown: $\vec{R}$ and $\vec{T}$

→ 6 unknown values

6 independent equations are needed. You will need 3 pairs of $\vec{p}$ and $\vec{q}$ to solve for the unknowns.

c) $q_{1,x} = R_{xx} \cdot P_{1,x} + R_{xy} \cdot P_{1,y} + T_x$

$q_{1,y} = R_{yx} \cdot P_{1,x} + R_{yy} \cdot P_{1,y} + T_y$

$q_{2,x} = R_{xx} \cdot P_{2,x} + R_{xy} \cdot P_{2,y} + T_x$

$q_{2,y} = R_{yx} \cdot P_{2,x} + R_{yy} \cdot P_{2,y} + T_y$

$q_{3,x} = R_{xx} \cdot P_{3,x} + R_{xy} \cdot P_{3,y} + T_x$

$q_{3,y} = R_{yx} \cdot P_{3,y} + R_{yy} \cdot P_{3,y} + T_y$

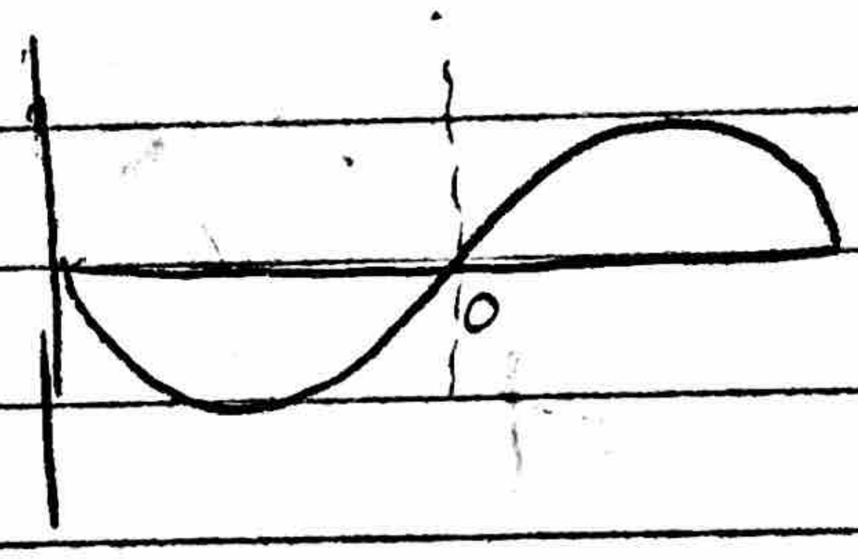
$$\begin{bmatrix} q_{1x} \\ q_{1y} \\ q_{2x} \\ q_{2y} \\ q_{3x} \\ q_{3y} \end{bmatrix} = \begin{bmatrix} P_{1,x} & P_{1,y} & 0 & 0 & 1 & 0 \\ 0 & 0 & P_{1,x} & P_{1,y} & 0 & 1 \\ P_{2,x} & P_{2,y} & 0 & 0 & 1 & 0 \\ 0 & 0 & P_{2,x} & P_{3,y} & 0 & 1 \\ P_{3,x} & P_{3,y} & 0 & 0 & 1 & 0 \\ 0 & 0 & P_{3,x} & P_{3,y} & 0 & 1 \end{bmatrix}\begin{bmatrix} R_{xx} \\ R_{xy} \\ R_{yx} \\ R_{yy} \\ T_x \\ T_y \end{bmatrix}$$

d) See ipynb.

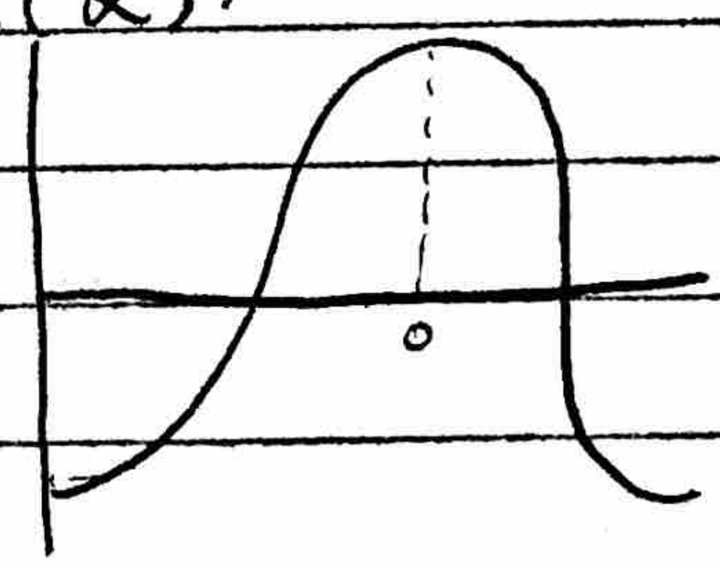
e) I receive an error that states the rows are not linearly independent.

f) If p_1, p_2, and p_3 are collinear, then there is a degenerate line because $\vec{p}_2 - \vec{p}_1$ is a multiple of $\vec{p}_3 - \vec{p}_1$.

2a) $\sin(\alpha)$:



$\cos(\alpha)$:



Yes, the approximations are accurate.

b) $A = \begin{bmatrix} 1 & 0 & 0 & \Delta t \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$

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