EE16A: Homework 3

In [2]:
```
%matplotlib inline
from numpy import zeros, cos, sin, arange, around, hstack
from matplotlib import pyplot as plt
from matplotlib import animation
from matplotlib.patches import Rectangle
import numpy as np
from scipy.interpolate import interp1d
import scipy as sp
import wave
import scipy.io.wavfile
import operator
from IPython.display import Audio
```

Problem 2: Elementary Matrices

Part (b)

In [13]:
```
A = np.array([[1, -2, 0, -5],
              [0, 1, 0, 3],
              [-2, -3, 1, -6],
              [0, 1, 0, 2]]);

E = np.linalg.inv(A)
E
```

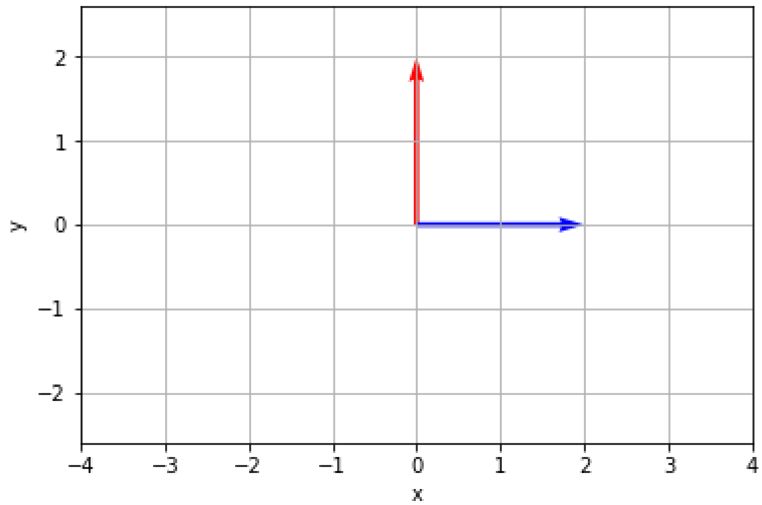
Out[13]:
```
array([[ 1.,  1., -0.,  1.],
       [-0., -2., -0.,  3.],
       [ 2.,  2.,  1.,  5.],
       [-0.,  1., -0., -1.]])
```

Problem 3: Mechanical Inverses

Part (d)

In [14]:
```
def rotation_matrix(v, theta):
    """
    Inputs:
        v: Numpy array with an x- and y-component.
        theta: Float.
    Returns:
        Numpy array with an x- and y-component.
    """
    A = np.array([[np.cos(theta), -np.sin(theta)],
                  [np.sin(theta), np.cos(theta)]])
    return A.dot(v)

def plot_rotation_matrix(v, theta):
    """
    Inputs:
        v: Numpy array with an x- and y-component.
        theta: Float.
    Returns:
        None.
    """
    # plotting the transformation
    origin = [0], [0]
    u = rotation_matrix(v, theta)
    plt.axis('equal')
    plt.quiver(*origin, [u[0], v[0]], [u[1], v[1]], color=['r', 'b'], scale

    # setting appropriate plot boundaries
    boundary = np.linalg.norm(v)*2
    plt.xlim(-boundary, boundary)
    plt.ylim(-boundary, boundary)

    # plot cleanliness
    plt.xlabel("x")
    plt.ylabel("y")
    plt.grid()
    return
```

In [15]:
```
# Change v and theta to see how the rotation operation affects it
v = np.array([2, 0])
theta = np.pi/2

plot_rotation_matrix(v, theta)
```



Problem 6: Segway Tours

Run the following block of code first to get all the dependencies.

In [16]:
```
# %load gauss_elim.py
from gauss_elim import gauss_elim
```

In [17]:
```
from numpy import zeros, cos, sin, arange, around, hstack
from matplotlib import pyplot as plt
from matplotlib import animation
from matplotlib.patches import Rectangle
import numpy as np
from scipy.interpolate import interp1d
import scipy as sp
```

Dynamics

In [18]:
```
# Dynamics: state to state
A = np.array([[1, 0.05, -.01, 0],
              [0, 0.22, -.17, -.01],
              [0, 0.1, 1.14, 0.10],
              [0, 1.66, 2.85, 1.14]]);
# Control to state
b = np.array([.01, .21, -.03, -0.44])
nr_states = b.shape[0]

# Initial state
state0 = np.array([-0.3853493, 6.1032227, 0.8120005, -14])

# Final (terminal state)
stateFinal = np.array([0, 0, 0, 0])
```

Part (d), (e), (f)

In [19]:
```
# You may use gauss_elim to help you find the row reduced echelon form.
```

Part (g)

Preamble

This function will take care of animating the segway.

In [20]:
```
# frames per second in simulation
fps = 20
# length of the segway arm/stick
stick_length = 1.

def animate_segway(t, states, controls, length):
    #Animates the segway

    # Set up the figure, the axis, and the plot elements we want to animate
    fig = plt.figure()

    # some config
    segway_width = 0.4
    segway_height = 0.2

    # x coordinate of the segway stick
    segwayStick_x = length * np.add(states[:, 0],sin(states[:, 2]))
    segwayStick_y = length * cos(states[:, 2])

    # set the limits
    xmin = min(around(states[:, 0].min() - segway_width / 2.0, 1), around(s
    xmax = max(around(states[:, 0].max() + segway_height / 2.0, 1), around(

    # create the axes
    ax = plt.axes(xlim=(xmin-.2, xmax+.2), ylim=(-length-.1, length+.1), as

    # display the current time
    time_text = ax.text(0.05, 0.9, '', transform=ax.transAxes)

    # display the current control
    control_text = ax.text(0.05, 0.8, '', transform=ax.transAxes)

    # create rectangle for the segway
    rect = Rectangle([states[0, 0] - segway_width / 2.0, -segway_height / 2
        segway_width, segway_height, fill=True, color='gold', ec='blue')
    ax.add_patch(rect)

    # blank line for the stick with o for the ends
    stick_line, = ax.plot([], [], lw=2, marker='o', markersize=6, color='bl

    # vector for the control (force)
    force_vec = ax.quiver([],[],[],[],angles='xy',scale_units='xy',scale=1)

    # initialization function: plot the background of each frame
    def init():
        time_text.set_text('')
        control_text.set_text('')
        rect.set_xy((0.0, 0.0))
        stick_line.set_data([], [])
        return time_text, rect, stick_line, control_text

    # animation function: update the objects
    def animate(i):
        time_text.set_text('time = {:2.2f}'.format(t[i]))
        control_text.set_text('force = {:2.3f}'.format(controls[i]))
        rect.set_xy((states[i, 0] - segway_width / 2.0, -segway_height / 2)
```

```
        stick_line.set_data([states[i, 0], segwayStick_x[i]], [0, segwaySti
        return time_text, rect, stick_line, control_text

    # call the animator function
    anim = animation.FuncAnimation(fig, animate, frames=len(t), init_func=i
            interval=1000/fps, blit=False, repeat=False)
    return anim
    # plt.show()
```

Plug in your controller here

In [26]:

```
controls = np.array([-13.24875075, 23.73325125, -11.57181872, 1.46515973])
```

Simulation

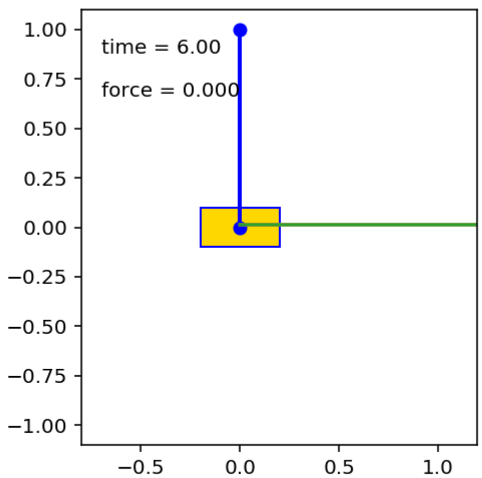
In [27]:

```
# This will add an extra couple of seconds to the simulation after the inpu
# the effect of this is just to show how the system will continue after the
controls = np.append(controls,[0, 0])

# number of steps in the simulation
nr_steps = controls.shape[0]

# We now compute finer dynamics and control vectors for smoother visualizat
Afine = sp.linalg.fractional_matrix_power(A,(1/fps))
Asum = np.eye(nr_states)
for i in range(1, fps):
    Asum = Asum + np.linalg.matrix_power(Afine,i)

bfine = np.linalg.inv(Asum).dot(b)

# We also expand the controls in the "intermediate steps" (only for visuali
controls_final = np.outer(controls, np.ones(fps)).flatten()
controls_final = np.append(controls_final, [0])

# We compute all the states starting from x0 and using the controls
states = np.empty([fps*(nr_steps)+1, nr_states])
states[0,:] = state0;
for stepId in range(1,fps*(nr_steps)+1):
    states[stepId, :] = np.dot(Afine,states[stepId-1, :]) + controls_final[

# Now create the time vector for simulation
t = np.linspace(1/fps,nr_steps,fps*(nr_steps),endpoint=True)
t = np.append([0], t)
```

Visualization

In [28]:
```
%matplotlib nbagg
# %matplotlib qt
anim = animate_segway(t, states, controls_final, stick_length)
anim
```

Figure 1



Out[28]:
```
<matplotlib.animation.FuncAnimation at 0xd17cf9ef0>
```

Problem 7: Audio File Matching

This notebook continues the audio file matching problem. Be sure to have song.wav and clip.wav in the same directory as the notebook.

In this notebook, we will look at the problem of searching for a small audio clip inside a song.

The song "Mandelbrot Set" by Jonathan Coulton is licensed under [CC BY-NC 3.0 (http://creativecommons.org/licenses/by-nc/3.0/)](http://creativecommons.org/licenses/by-nc/3.0/)

In [29]:
```
given_file = 'song.wav'
target_file = 'clip.wav'
rate_given,  given_signal  = scipy.io.wavfile.read(given_file)
rate_target, target_signal = scipy.io.wavfile.read(target_file)
given_signal  = given_signal[:2000000].astype(float)
target_signal = target_signal.astype(float)
def play_clip(start, end, signal=given_signal):
    return Audio(data=signal[start:end], rate=rate_given)

def run_comparison(target_signal, given_signal, idxs=None):
    # Run everything if not called with idxs set to something
    if idxs is None:
        idxs = [i for i in range(len(given_signal)-len(target_signal))]
    return idxs, [vector_compare(target_signal, given_signal[i:i+len(target
                for i in idxs]

play_clip(0, len(given_signal),given_signal)

#scipy.io.wavfile.write(target_file, rate_given, (-0.125*given_signal[1380(
```

Out[29]:

0:04 / 0:45

We will load the song into the variable `given_signal` and load the short clip into the variable `target_signal`. Your job is to finish code that will identify the short clip's location in the song. The clip we are trying to find will play after executing the following block.

In [30]:
```
play_clip(0, len(target_signal),signal=target_signal)
```

Out[30]:

0:00 / 0:01

Part (d)

Run the following cell. Do your results here make sense given your answers to previous parts of the problem? What is the function `vector_compare` doing?

In [36]:
```
def vector_compare(desired_vec, test_vec):
    """This function compares two vectors, returning a number.
    The test vector with the highest return value is regarded as being clos
    return np.dot(desired_vec.T, test_vec)/(np.linalg.norm(desired_vec)*np.

print("PART A:")
print(vector_compare(np.array([1,1,1]), np.array([1,1,1])))
print(vector_compare(np.array([1,1,1]), np.array([-1,-1,-1])))
print("PART C:")
print(vector_compare(np.array([1,2,3]), np.array([1,2,3])))
print(vector_compare(np.array([1,2,3]), np.array([2,3,4])))
print(vector_compare(np.array([1,2,3]), np.array([3,4,5])))
print(vector_compare(np.array([1,2,3]), np.array([4,5,6])))
print(vector_compare(np.array([1,2,3]), np.array([5,6,7])))
print(vector_compare(np.array([1,2,3]), np.array([6,7,8])))
```

```
PART A:
0.9999999999666668
-0.9999999999666668
PART C:
0.9999999999928572
0.9925833339660043
0.9827076298202766
0.9746318461941077
0.968329663729021
0.9633753381636556
```

Run the following code that runs `vector_compare` on every subsequence in the song- it will probably take at least 5 minutes. How do you interpret this plot to find where the clip is in the song?

In [37]:
```
import time

t0 = time.time()
idxs, song_compare = run_comparison(target_signal, given_signal)
t1 = time.time()
plt.plot(idxs, song_compare)
print ("That took %(time).2f minutes to run" % {'time':(t1-t0)/60.0} )
```

```
That took 1.27 minutes to run
```

Part (e)

The code below uses `song_compare` to print the index of `given_signal` where `target_signal` begins. Can you interpret how the code finds `index` ? Verify that the code is correct by playing the song at that index using the `play_clip` function.

In [34]:
```
index, value = max(enumerate([abs(i) for i in song_compare]), key=operator.
print (index)
play_clip(index,index+len(target_signal))
```

```
1380000
```

Out[34]:

0:00 / 0:01

In [ ]:

In [ ]: