# EECS 106B : Lab #1

Spring 2018

**Yichi Zhang, Jingjun Liu, Jiarong Li**
**27005775, 3033483513, 3033483511**

February 14, 2018

## Section 1

# Youtube Link of the Results:

# https://youtu.be/slmNfUcozF8

## Section 2

# Control Methods

### Controller 1: Workspaces Velocity Control

Firstly, we get the position of AR tag $x_d$from camera by a listener function and the end effector position of robot arm $x$ by a package function. Next, we calculate error by

$$e = x - x_d$$

Secondly, we implemented a PD Control on the system using

$$v = -K_v \dot{e} - K_p e$$

where $v \in R^3$ is the end effector velocity expressed in workspace and $\dot{e} = e_{previous} - e_{current}$
Thirdly, we get the joint-space velocity by

$$\dot{\theta} = J^{-}1V$$

At last, we set $\dot{\theta}$ by *setjointvelocities* in Baxter interface.

### Controller 2: Joint Velocity Control

Firstly, we get the position of AR tag $x_d$ and calculate the inverse kinematics to get $\theta_d$, and the joints states of robot arm $\theta$ by a package function. Next, we calculate error by

$$e = \theta - \theta_d$$

Secondly, we implemented a PD Control on the system using

$$\dot{\theta} = -K_v \dot{e} - K_p e$$

where $\theta \in R^7$ is the joint velocity expressed in workspace.
At last, we set $\dot{\Theta}$ by *setjointvelocities* in Baxter interface.

### Controller 3: Joint-space Torque Control

To begin with, we get the workspace velocity and calculate joint-space velocity by

$$e = J^{-1}V$$

Then, we $\ddot{\theta}_d$ by

$$\ddot{\theta}_d = J_{-1}\ddot{x} + \frac{d}{dt}J^{-1}\dot{x}$$

where $\ddot{x}$ is set as zero except in circular path and $\frac{d}{dt}J^{-}1 == J^{-1}_{previous} - J^{-1}_{current}$
Next, we calculate the torque by

$$\tau = M(\theta)\ddot{\theta}_d + C(\theta, \theta)\theta_d + N(\theta, \theta) - K_v \dot{e} - K_p \dot{e}$$

where $K_v,\ K_p \in R^{7\times 7}$, $C(\theta, \theta),\ N(\theta, \theta)$ is set as zero.

# Section 3

# Paths Function Methods

## Path 1: Linear Path

### Target Position

We get target position $x_d$ by lookupTag function and add 15cm to z axis to make robot hand not touch the table.

### Target Velocity

We get target velocity $\dot{x_d}$ by divide the distance between target position $x_d$ and robot end effector position $x$ into average pieces.

### Target Angle

We get target angle by calculate the inverse kinematics of target position $x_d$.

## Path 2: Circular Path

### Target Position

Firstly, we get the tag position. Then, we calculate target position $x_d$ in polar coordinate system by setting up an angular velocity.

### Target Velocity

We get target velocity $\dot{x_d}$ using

$$v = w \times r$$

where $w$ is set by us and $r$ is the position from AR Tag to current end effector position.

### Target Angle

We get target angle by calculate the inverse kinematics of target position $x_d$.

### Target Acceleration

We get target acceleration $\ddot{x_d}$ using

$$\ddot{x_d} = w^2 \times r \times \hat{a}$$

where $w$ is set by us, $r$ is the position from AR Tag to current end effector position and $\hat{a} = \frac{(x_{center} - x_{target})}{\|(x_{center} - x_{target})\|}$.

## Path 3: Multiple(Rectangle) Paths

### Target Position

Firstly, we get the tag position. Then, we calculate 4 corners position $x_{d1}$ to $x_{d4}$ and turn current target position $x_{d(i)}$ to $x_{d(i+1)}$ when the current robot position is arrive at $x_{d(i)}$.

### Target Angle

We get target angle by calculate the inverse kinematics of target position $x_d$.

Problem 3 continued on next page. . . 

### Path 4: Linear Path following AR marker

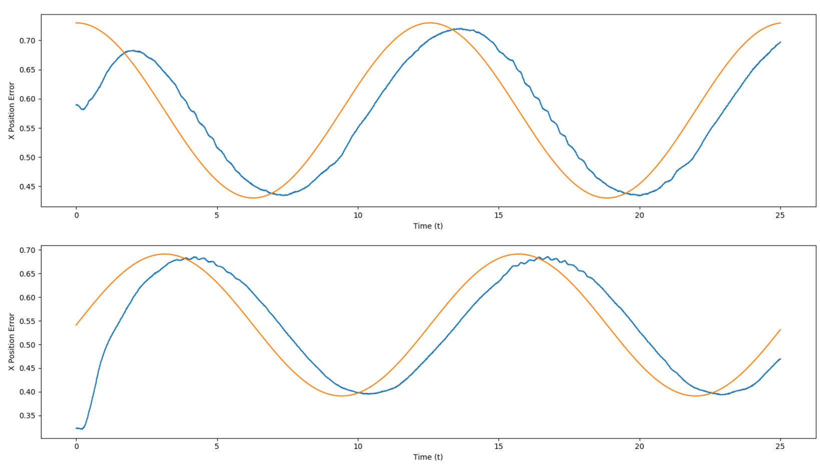
We just charge the target position part in Path 1: Linear Path, where we use LookupTag function each time we call GetTargetPosition function.

## Section 4

# Results

### Joint Position Control

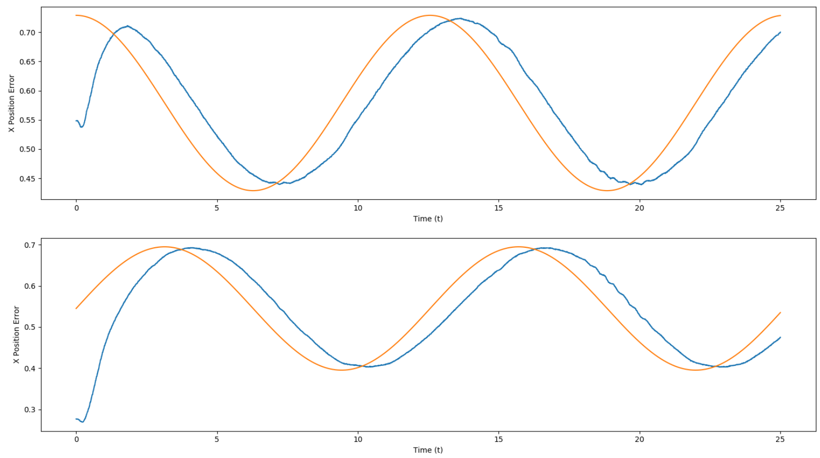
The actual positions of the end effector in x and y directions are plotted in the figure below, compared with the target positions. The Baxter's left arm is executing under joint position control and trying to follow a circular path in this case. The position controller behaves well in linear, circular, multi-path tracking. In our implementation, the robot arm always scan the position of the AR tag and tries to track it when executing the linear path. The orange line represents target positions and the blue line demonstrates actual




positions. As we can see in the figure, the end effector is not on the desired position initially, but after about 3 seconds, it moves to the target position and starts to follow the desired trajectory smoothly. There is, however, a static residual between the actual trajectory and the desired trajectory, which is also shown in the figure above. This is because the PD controller will probably lead to a static error after the system reaches stability. Adding an integration term will help solving this problem, but PID controller can not be proved as Lyapunov stable, so we simply implement the PD controller here. We will keep seeing this static error in our experiments. It is also deserved mentioning that the end effector shakes a little bit in the time interval from 15s to 18s. After checking the status when the robot arm is running, we find that it is because the arm is approaching the maximum length while it is drawing the outer arc of the circle, so the control is less smooth.

### Joint Velocity Control

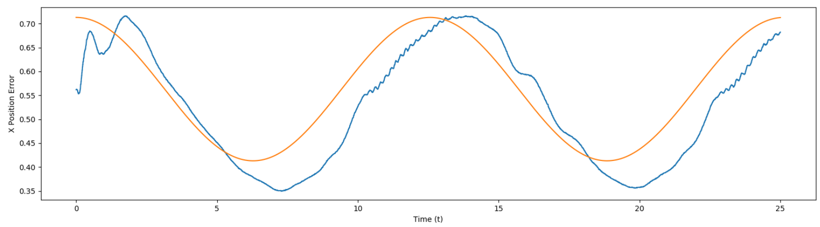
The actual positions of the end effector in x and y directions are plotted in the figure below, compared with the target positions. The Baxter's left arm is executing under joint velocity control and trying to follow a circular path in this case. The velocity controller behaves well in linear, circular, multi-path tracking. In our implementation, the robot arm always scan the position of the AR tag and tries to track it when executing the linear path. The orange line represents target positions and the blue line demonstrates actual positions.

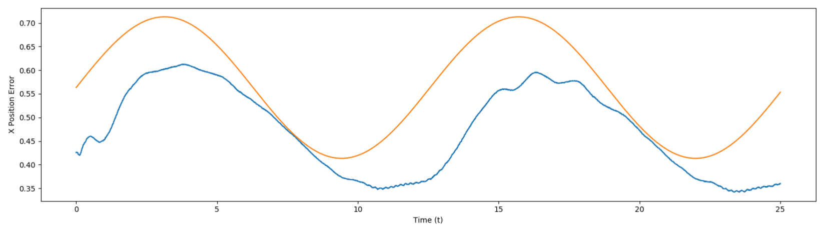



As we can see in the figure, the end effector is not on the desired position initially, but after about 3 seconds, it moves to the target position and starts to follow the desired trajectory smoothly. There is still a static residual between the actual trajectory and the desired trajectory, which we have mentioned in the previous part. This is because the property of the PD controller. In addition, the whole behavior of velocity controller is better than position controller from a global view. The actual trajectory is smoother than that of the position controller.

## Joint Torque Control

After careful debugging and painful parameter tuning, we finally succeeded to implement the torque controller and let the robot arm run in a roughly smooth condition. It basically achieves the goal of trajectory tracking, for example, linear path, circular path, multi-path. The actual positions in x and y directions as well as the desired positions are demonstrated in the figure below. We can see from the figure that the tracking trajectory is much more ugly that velocity and position control. This is maybe because some particular joints need to compensate the gravity and coriolis force, but we ignore these two parts and expect good parameters to reach the same effect instead of numerically calculating these two terms out. Actually these two terms are difficult to calculate, but if we want to fix the ugly trajectory under the torque controller circumstance, we should carefully calculate both of the terms out as accurate feed-forwards.




# Section 5

## Application

### Workspace Velocity Controller

This controller performs well in linear path. We believe it can be used on some simple works which can be described by their paths. For example, lifting things to a certain height.

### Joint Velocity Controller

This controllers performance is similar to Workspace Velocity Controller, and it spends less time eliminating the error and it is smoother. Therefore, we think it can handle some harder work. For example, if we want baxter to hand over a glass of water, we can use this controller to maintain balance. It is the same in some balance-needed works.

### Torque Controller

Although this controller shows the worst behavior and stability in lab. We still believe that it is the strongest controller. It can handle some really complicate works like following the path of a combination of several geometric trajectories. We think it can be used in controlling the exoskeleton. For example, we can measure the movement of humans arm; get the velocity, acceleration and then compute the torque needed for each joint; send it to exoskeleton and set a series of parameters. Ideally, it should move as we want.

## Section 6

# Bouns: Difficulties

### Difficulty in inverse kinematics

When we firstly used this function to get the IK of each joints, we set the parameters as the what we get from tracking the AR marker. However, it cannot get an answer. Then we figured out that it is the quaternion. The quaternion of AR marker cannot be reached by baxters end joint. So we set the parameter as the position of AR marker and quaternion of baxters end effectors joint.

### Difficulty in Kp,Ki of torque controller

When we firstly implemented torque controller, we use the K the same as the other two controllers. And basically the arm can not even move. Later we figured out that the K of this controller is more important than the other two, and it varies at every joint. Therefore, we set a matrix of K and after many times of experiments we finally decided a K that can move the arm along the target trace.

# Bouns: Compared with MoveIt

These controllers are smoother and quicker than MoveIt.

## Section 7

Create paths:

```
import rospy
import numpy as np
import math
from utils import *
import baxter_interface
import moveit_commander
from moveit_msgs.msg import OrientationConstraint, Constraints
from geometry_msgs.msg import PoseStamped

# import IPython
import tf
import tf2_ros
import time
from baxter_pykdl import baxter_kinematics
import signal



# from controllers import PDJointPositionController, PDJointVelocityController, PDJointTorqueController
#from paths import LinearPath, CircularPath, MultiplePaths

"""
Starter script for lab1.
Author: Chris Correa
"""

# IMPORTANT: the init methods in this file may require extra parameters not
# included in the starter code.
def lookup_tag(tag_number):

    listener = tf.TransformListener()
    from_frame = 'base'
    to_frame = 'ar_marker_{}'.format(tag_number)
    # if not listener.frameExists(from_frame) or not listener.frameExists(to_frame):
    #     print 'Frames not found'
    #     print 'Did you place AR marker {} within view of the baxter left hand camera?'.format(tag_number)
    #     exit(0)
    # t = rospy.Time(0)*
    # if listener.canTransform(from_frame, to_frame, t):
    listener.waitForTransform(from_frame, to_frame, rospy.Time(), rospy.Duration(4.0))
    t = listener.getLatestCommonTime(from_frame, to_frame)
    tag_pos, tag_rot = listener.lookupTransform(from_frame, to_frame, t)
    return (tag_pos + tag_rot)    # Return value is a 'list'

def getEndPointPosition(limb):
    pose = limb.endpoint_pose()
    poselist = list(tuple(pose['position']) + tuple(pose['orientation']))
    return poselist # return value is a 7x 'list'


class MotionPath:
    def target_position(self, time):
        raise NotImplementedError

    def target_velocity(self, time):
        raise NotImplementedError
```

```python

    def target_acceleration(self, time):
        raise NotImplementedError

class LinearPath(MotionPath):
    def __init__(self, kin, limb):
        self.kin = kin
        self.limb = limb
        self.end_point = getEndPointPosition(self.limb)
        # self.rotation = getEndPointPosition(self.limb)[3:]
        self.rotation = self.end_point[3:]
        self.cur_pos = self.end_point[:3]
        self.target_pos = [0,0,0,0,0,0,0]
        self.target_pos_torque = [0,0,0,0,0,0,0]
        #self.target[2] += 0.2
        #self.init_pos = cur_pos
        #self.velocity = list((np.array(tag_pos[0:3]) - np.array(cur_pos[0:3])) / dfactor)

    def target_position(self, t):
        tag_pos = lookup_tag(4)
        tag_pos[2] = tag_pos[2] + 0.15

        self.target_pos = tag_pos

        # return list(np.array(self.init_pos[0:3]) + np.array(self.velocity) * 2) + self.
    init_pos[3:]
        return tag_pos # 7 list

    def target_velocity(self, t):
        velocity_vec = -(np.array(self.cur_pos) - np.array(self.target_pos[:3]))
        x_d_dot = list(velocity_vec/np.linalg.norm(velocity_vec))+[0,0,0]
        return np.array(x_d_dot)

    def target_angle(self, t):
        tag_pos = lookup_tag(4)
        tag_pos[2] = tag_pos[2] + 0.15
        # good
        position = tag_pos[0:3]
        counter = 0
        # print tag_pos
        while True:
            counter += 1
            if counter > 200:
                break
            tar_joint_angle = self.kin.inverse_kinematics(position, self.rotation)
            if tar_joint_angle is not None:
                break
        # print(tar_joint_angle)
        return tar_joint_angle # return value is a list contains 7 joint angles

    def target_acceleration(self, t):
        return np.zeros(6)

class LinearPath_no_tracking(MotionPath):
    def __init__(self, kin, limb):
        self.kin = kin
        self.limb = limb
        self.end_point = getEndPointPosition(self.limb)
        # self.rotation = getEndPointPosition(self.limb)[3:]
        self.rotation = self.end_point[3:]
        self.cur_pos = self.end_point[:3]
        self.target_pos = lookup_tag(4)
```

Problem 7 continued on next page. . . 

```
        self.target_pos[2] += 0.15
        self.target_pos_torque = [0,0,0,0,0,0,0]

    def target_position(self, t):
        # tag_pos = lookup_tag(4)
        # tag_pos[2] = tag_pos[2] + 0.15
        # self.target_pos = tag_pos

        # return list(np.array(self.init_pos[0:3]) + np.array(self.velocity) * 2) + self.
    init_pos[3:]
        return self.target_pos # 7 list

    def target_velocity(self, t):
        self.end_point = getEndPointPosition(self.limb)
        self.cur_pos = self.end_point[:3]
        velocity_vec = -(np.array(self.cur_pos) - np.array(self.target_pos[:3]))
        x_d_dot = list(velocity_vec/np.linalg.norm(velocity_vec))+[0,0,0]
        return np.array(x_d_dot)

    def target_angle(self, t):
        position = self.target_pos[0:3]
        counter = 0
        # print tag_pos
        while True:
            counter += 1
            if counter > 200:
                break
            tar_joint_angle = self.kin.inverse_kinematics(position, self.rotation)
            if tar_joint_angle is not None:
                break
        # print(tar_joint_angle)
        return tar_joint_angle # return value is a list contains 7 joint angles

    def target_acceleration(self, t):
        return np.zeros(6)


class CircularPath(MotionPath):
    def __init__(self, kin, limb, radius=0.15):
        self.radius = radius
        self.kin = kin
        self.theta = 0
        self.angular_v = 0.5
        self.limb = limb
        self.center_pos = lookup_tag(4)
        self.center_pos[2] = self.center_pos[2]+0.2
        self.target_pos = self.center_pos
        self.rotation = getEndPointPosition(self.limb)[3:]
               # if radius is not None:
        #     self.circle_pos = self.set_circle_pos(radius)
        # else:
        #     pass
    # def set_circle_pos(self, radius):
    #     tot_dis = sqrt(self.center_pos^2 + self.init_pos^2)
    #     ratio = 1 - radius / tot_dis
    #     return self.init_pos + (self.center_pos - self.init_pos) * ratio

    # def move_to_circle(self):
    #     clinearpath = LinearPath(self.center_pos, self.circle_pos)

    def target_position(self, t):
        self.theta = np.remainder(self.angular_v*t,2*np.pi)
```

```
        x = self.radius*np.cos(self.theta) + self.center_pos[0]
        y = self.radius*np.sin(self.theta) + self.center_pos[1]
        target_pos = [x,y] + [self.center_pos[2]] + self.rotation

        # self.center_pos = center_pos
        self.target_pos = target_pos
        return target_pos # 7x list

    def target_angle(self, t):
        tag_pos = self.target_position(t)
        # good
        position = tag_pos[0:3]
        rotation = tag_pos[3:]
        tar_joint_angle = self.kin.inverse_kinematics(position, rotation)
        # print(tar_joint_angle)
        return tar_joint_angle # return value is a list contains 7 joint angles

    def target_velocity(self, t):
        w = np.array([0,0,self.angular_v])
        r = np.array(self.center_pos[0:3]) - np.array(self.target_pos[0:3])
        x_d_dot = list(hat(w).dot(r))+[0,0,0]
        return np.array(x_d_dot)


    def target_acceleration(self, t):
        acc_vec = (np.array(self.center_pos)[:3] - np.array(self.target_pos)[:3])
        acc_dir = acc_vec / np.linalg.norm(acc_vec)
        acc = self.angular_v * self.angular_v * self.radius * acc_dir
        return np.array(list(acc) + [0, 0, 0])

# You can implement multiple paths a couple ways.  The way I chose when I took
# the class was to create several different paths and pass those into the
# MultiplePaths object, which would determine when to go onto the next path.

class MultiplePaths(MotionPath):
    def __init__(self, kin):
        self.kin = kin
        self.tag_poses = []
        for i in range(4):
            tag_pos = lookup_tag(i)
            tag_pos[2] += 0.2
            self.tag_poses.append(tag_pos)
        self.index = 0
        self.limb = baxter_interface.Limb('left')
        """
    def target_position(self, t, dfactor=50):
        # print('t:',t)
        print('t//1',t//1)
        if np.remainder(t//1, dfactor) < 1:
            self.index = np.remainder(self.index + 1, 4)
        print('index:',self.index)
        return self.tag_poses[int(self.index)]
        """
    def target_position(self, t):
        cur_pos = getEndPointPosition(self.limb)
        tar_pos = self.tag_poses[self.index]
        error = 0.01
        if np.linalg.norm(np.array(cur_pos[:2]) - np.array(tar_pos[:2])) < error:
            self.index = np.remainder(self.index + 1, 4)
        # print tar_pos
        return self.tag_poses[self.index]
```

```
    # def target_velocity(self, t):
    #     # return lookup_tag(4)
    #     return list(np.array(self.init_pos[0:3]) + np.array(self.velocity) * 2) + self.
    init_pos[3:]
    def target_angle(self, t):
        if np.remainder(t, dfactor) < 0.1:
            self.index = np.remainder(self.index + 1, 4)
            tar_pos = self.tag_poses[self.index]
        position = tag_pos[0:3]
        rotation = tag_pos[3:]
        tar_joint_angle = self.kin.inverse_kinematics(position, rotation)
        print(tar_joint_angle)
        return tar_joint_angle # return value is a list contains 7 joint angles

    def target_acceleration(self, t):
        return 0
```

./paths.py

Controller Implementation:

```
import rospy
import numpy as np
from utils import *
from geometry_msgs.msg import PoseStamped
from paths import *

from baxter_pykdl import baxter_kinematics

"""
Starter script for lab1.
Author: Chris Correa
"""
class Controller:
    # def __init__():
    #     self.current_joint_pos = [0,0,0,0,0,0,0]

    def step_path(self, path, t):
        raise NotImplementedError

    def finished(self, t_flag, t):
        if t >= t_flag:
            return True
        else:
            return False



    def execute_path(self, path, finished, timeout=None, log=True):
        start_t = rospy.Time.now()
        # print('star_t:',start_t)
        t_flag = 25
        times = list()
        actual_positions_x = list()
        actual_positions_y = list()
        # actual_velocities = list()
        target_positions_x = list()
        target_positions_y = list()
        # target_velocities = list()
        r = rospy.Rate(200)
        while True:
            t = (rospy.Time.now() - start_t).to_sec()
            # print(t_flag,t)
```

```
            if timeout is not None and t >= timeout:
                return False
            self.step_path(path, t)  # move
            if log:
                times.append(t)
                actual_positions_x.append(self.current_pos_x)
                actual_positions_y.append(self.current_pos_y)
                # actual_velocities.append(self.current_joint_vel)
                target_positions_x.append(self.target_pos_x)
                target_positions_y.append(self.target_pos_y)
                # target_velocities.append(path.target_velocity(t))
            if self.finished(t_flag, t):
                break
            r.sleep()

        if log:
            import matplotlib.pyplot as plt

            # print(target_positions_x)

            # np_actual_positions = np.zeros((len(times), 3))
            # np_actual_velocities = np.zeros((len(times), 3))
            # for i in range(len(times)):
            #     # print actual_positions[i]
            #     actual_positions_dict = dict((joint, actual_positions[i][j]) for j, joint
    in enumerate(self.limb.joint_names()))
            #     # print "dictionary version", actual_positions_dict
            #     np_actual_positions[i] = self.kin.forward_position_kinematics(joint_values
    =actual_positions_dict)[:3]
            #     np_actual_velocities[i] = self.kin.jacobian(joint_values=
    actual_positions_dict)[:3].dot(actual_velocities[i])
            #     print(np_actual_positions)
            # target_positions = np.array(target_positions)
            # target_velocities = np.array(target_velocities)
            plt.figure()
            # print len(times), actual_positions.shape()
            plt.subplot(2,1,1)
            # plt.plot(times, np_actual_positions[:,0], label='Actual')
            # plt.plot(times, target_positions[:,0], label='Desired')
            # plt.xlabel("Time (t)")
            # plt.ylabel("X Position Error")
            plt.plot(times, actual_positions_x, label='Actual')
            plt.plot(times, target_positions_x, label='Desired')
            plt.xlabel("Time (t)")
            plt.ylabel("X Position Error")
            plt.subplot(2,1,2)
            plt.plot(times, actual_positions_y, label='Actual')
            plt.plot(times, target_positions_y, label='Desired')
            plt.xlabel("Time (t)")
            plt.ylabel("X Position Error")

            # plt.subplot(3,2,2)
            # plt.plot(times, np_actual_velocities[:,0], label='Actual')
            # plt.plot(times, target_velocities[:,0], label='Desired')
            # plt.xlabel("Time (t)")
            # plt.ylabel("X Velocity Error")

            # plt.subplot(3,2,3)
            # plt.plot(times, np_actual_positions[:,1], label='Actual')
            # plt.plot(times, target_positions[:,1], label='Desired')
            # plt.xlabel("time (t)")
            # plt.ylabel("Y Position Error")
```

Problem 7 continued on next page. . .

```
            # plt.subplot(3,2,4)
            # plt.plot(times, np_actual_velocities[:,1], label='Actual')
            # plt.plot(times, target_velocities[:,1], label='Desired')
            # plt.xlabel("Time (t)")
            # plt.ylabel("Y Velocity Error")

            # plt.subplot(3,2,5)
            # plt.plot(times, np_actual_positions[:,2], label='Actual')
            # plt.plot(times, target_positions[:,2], label='Desired')
            # plt.xlabel("time (t)")
            # plt.ylabel("Z Position Error")

            # plt.subplot(3,2,6)
            # plt.plot(times, np_actual_velocities[:,2], label='Actual')
            # plt.plot(times, target_velocities[:,2], label='Desired')
            # plt.xlabel("Time (t)")
            # plt.ylabel("Z Velocity Error")

            plt.show()

        return True

class PDJointPositionController(Controller): # PDWorkSpaceVelocityController
    def __init__(self, limb, kin, Kp, Kv):
        self.limb = limb
        self.kin = kin
        self.Kp = Kp
        self.Kv = Kv
        self.error_last = np.zeros(6)
        self.joint_names = self.limb.joint_names()
        self.current_pos_x = 0
        self.current_pos_y = 0
        self.target_pos_x = 0
        self.target_pos_y = 0
        # self.current_joint_vel = [0,0,0,0,0,0,0]

    def step_path(self, path, t):
        # YOUR CODE HERE

        target_pos = path.target_position(t) # 7x list
        cur_pos = getEndPointPosition(self.limb)  # 7x list
        self.current_pos_x = cur_pos[0]
        self.current_pos_y = cur_pos[1]
        self.target_pos_x = target_pos[0]
        self.target_pos_y = target_pos[1]
        # current_joint_pos_dic = self.limb.joint_angles()
        # for i in range(len(self.joint_names)):
        #     self.current_joint_pos[i] = current_joint_pos_dic[self.joint_names[i]]

        # joint_vel_dic = self.limb.joint_velocities()
        # for i in range(len(self.joint_names)):
        #     self.current_joint_vel[i] = joint_vel_dic[self.joint_names[i]]

        # print(self.current_joint_vel)
        error = np.array( list(np.array(target_pos[0:3]) - np.array(cur_pos[0:3]) ) +
    [0,0,0] )
        derror = error - self.error_last
        self.error_last = error
        workspace_velocity = self.Kp * error + self.Kv * derror
        jacobian_inv = np.array(self.kin.jacobian_pseudo_inverse())
        target_joint_velocity = jacobian_inv.dot(workspace_velocity)
```

```
        input_joint_velocity = {}
        #print joint
        for i in range(len(self.joint_names)):
          input_joint_velocity[self.joint_names[i]] = target_joint_velocity[i]
        self.limb.set_joint_velocities(input_joint_velocity)



class PDJointVelocityController(Controller):
    def __init__(self, limb, kin, Kp, Kv):
        self.limb = limb
        self.kin = kin
        self.Kp = Kp
        self.Kv = Kv
        self.error_last = np.zeros(7)
        self.joint_names = self.limb.joint_names()

        self.current_pos_x = 0
        self.current_pos_y = 0
        self.target_pos_x = 0
        self.target_pos_y = 0

    def step_path(self, path, t):
        # YOUR CODE HERE
        target_joint_angle = path.target_angle(t) # 7x list
        cur_joint_position = getEndPointPosition(self.limb) # 7x list

        self.current_pos_x = cur_joint_position[0]
        self.current_pos_y = cur_joint_position[1]
        target_position = path.target_position(t)
        self.target_pos_x = target_position[0]
        self.target_pos_y = target_position[1]

        cur_joint_angle = self.kin.inverse_kinematics(cur_joint_position[:3], path.rotation)
        error = np.array(target_joint_angle) - np.array(cur_joint_angle) # 7x np.array
        derror = error - self.error_last
        self.error_last = error
        target_joint_velocity =  self.Kp * error + self.Kv * derror
        joint_velocity = {}
        for i in range(len(self.joint_names)):
            joint_velocity[self.joint_names[i]] = target_joint_velocity[i]
        self.limb.set_joint_velocities(joint_velocity)



class PDJointTorqueController(Controller):
    def __init__(self, limb, kin, Kp, Kv):
        self.limb = limb
        self.kin = kin
        self.Kp = Kp
        self.Kv = Kv
        self.error_last = np.zeros(7)
        self.jacobian_inverse_last = np.zeros((7,6))
        self.x_error_last = np.zeros(6)
        self.N = self.limb.joint_efforts()

        self.current_pos_x = 0
        self.current_pos_y = 0
        self.target_pos_x = 0
        self.target_pos_y = 0

    def getEndPointVelocity(self,limb):
        velocities = limb.joint_velocities()
```

```
        # velocitylist = [velocities['right_s0'],velocities['right_s1'],velocities['right_e0'],velocities['right_e1'],velocities['right_w0'],velocities['right_w1'],velocities['right_w2']]
        velocitylist = [velocities['left_s0'],velocities['left_s1'],velocities['left_e0'],velocities['left_e1'],velocities['left_w0'],velocities['left_w1'],velocities['left_w2']]

        return velocitylist # return value is a 7x 'list'

    def step_path(self, path, t):
        # YOUR CODE HERE
        joint_names = self.limb.joint_names()
        target_pos = path.target_position(t)

        self.target_pos_x = target_pos[0]
        self.target_pos_y = target_pos[1]

        target_pos_acc = np.array(path.target_acceleration(t))# np.array x6
        jacobian_inv = np.array(self.kin.jacobian_pseudo_inverse()) # array 7x6
        d_jacobian_inv = jacobian_inv - self.jacobian_inverse_last
        self.jacobian_inverse_last = jacobian_inv
        mass = np.array(self.kin.inertia())
        x_d_dot = path.target_velocity(t)
        cur_pos = getEndPointPosition(self.limb)  # 7x list

        self.current_pos_x = cur_pos[0]
        self.current_pos_y = cur_pos[1]

        target_pos = path.target_position(t) # 7x list
        vel_desire = self.kin.inverse_kinematics(target_pos[:3], path.rotation) #array 7
        vel_cur = self.kin.inverse_kinematics(cur_pos[:3],cur_pos[3:])
        error = vel_desire - vel_cur
        # derror = error - self.error_last
        derror = jacobian_inv.dot(x_d_dot) - self.getEndPointVelocity(self.limb)
        self.error_last = error
        ddtheta_desire = jacobian_inv.dot(target_pos_acc) + d_jacobian_inv.dot(x_d_dot)
        target_torque = mass.dot(ddtheta_desire) + np.array(self.Kp.dot(error)) + np.array(self.Kv.dot(derror))
        torque = {}

        # print joint_names
        for i in range(len(joint_names)):
            torque[joint_names[i]] = target_torque[i]

        # print('original_s1:',self.limb.joint_efforts()['left_e1'])
        # print('torque_s1:',torque['left_e1'])
        # print('\n')
        # print 'N', N
        # print 'target_torque', target_torque
        # print 'torque', torque
        # print '\n\n'
        self.limb.set_joint_torques(torque)

        # raw_input('enter')
```

./controllers.py

Main function:

```
#!/usr/bin/env python
"""
Starter script for lab1.
Author: Chris Correa
"""
```

```
import copy
import rospy
import sys
import argparse

import baxter_interface
import moveit_commander
from moveit_msgs.msg import OrientationConstraint, Constraints
from geometry_msgs.msg import PoseStamped

# import IPython
import tf
import tf2_ros
import time
import numpy as np
from utils import *
from baxter_pykdl import baxter_kinematics
import signal
from controllers import PDJointPositionController, PDJointVelocityController, PDJointTorqueController
#from paths import LinearPath, CircularPath, MultiplePaths
from paths import *

'''
def lookup_tag(tag_number):

    listener = tf.TransformListener()
    from_frame = 'base'
    to_frame = 'ar_marker_{}'.format(tag_number)
    # if not listener.frameExists(from_frame) or not listener.frameExists(to_frame):
    #     print 'Frames not found'
    #     print 'Did you place AR marker {} within view of the baxter left hand camera?'.format(tag_number)
    #     exit(0)
    # t = rospy.Time(0)*
    # if listener.canTransform(from_frame, to_frame, t):
    listener.waitForTransform(from_frame, to_frame, rospy.Time(), rospy.Duration(4.0))
    t = listener.getLatestCommonTime(from_frame, to_frame)
    tag_pos, tag_rot = listener.lookupTransform(from_frame, to_frame, t)
    return (tag_pos + tag_rot)    # Return value is a 'list'
'''

if __name__ == "__main__":
    def sigint_handler(signal, frame):
        sys.exit(0)
    signal.signal(signal.SIGINT, sigint_handler)
    moveit_commander.roscpp_initialize(sys.argv)
    rospy.init_node('moveit_node')
    time.sleep(1)

    parser = argparse.ArgumentParser()
    parser.add_argument('-ar_marker', '-ar', type=float, default=1)
    parser.add_argument('-controller', '-c', type=str, default='position') # or velocity or torque
    parser.add_argument('-arm', '-a', type=str, default='left') # or right
    args = parser.parse_args()

    arm = 'left'

    limb = baxter_interface.Limb(arm)
    kin = baxter_kinematics(arm)
```

```python
    if args.controller == 'position':
        # YOUR CODE HERE
        Kp = 1
        Kv = 1
        controller = PDJointPositionController(limb, kin, Kp, Kv)

    if args.controller == 'velocity':
        # YOUR CODE HERE
        Kp = 1
        Kv = 1
        controller = PDJointVelocityController(limb, kin, Kp, Kv)
    if args.controller == 'torque':
        # YOUR CODE HERE
        Kp = np.diag(np.array([10 ,40,2,17,2,5,2])) #for circular path
        # Kp = np.diag(np.array([1.5,10,1,1.5,1.3,0.1,0.1]))
        Kv = np.diag(np.array([5,5,1,5,1,3,1])) # for circular path
        # Kv = np.diag(np.array([0.5,1,1,1,0.1,0.1,0.1]))
        controller = PDJointTorqueController(limb, kin, Kp, Kv)

    #raw_input('Press <Enter> to start')
    print('\n')
    # YOUR CODE HERE

    # cur_pos = getEndPointPosition(limb)
    # linearpath = LinearPath(kin, limb)
    # linearpath_no_tracking = LinearPath_no_tracking(kin, limb)
    circularpath = CircularPath(kin, limb)
    # multiplepaths = MultiplePaths(kin)

    # controller.execute_path(linearpath, None)
    # controller.execute_path(linearpath_no_tracking, None)
    controller.execute_path(circularpath, None)
    # controller.execute_path(multiplepaths, None)

    #circularPath = CircularPath(tag_pos,cur_pos)
    #controller.execute_path(circularPath, None)
    # IMPORTANT: you may find useful functions in utils.py
```

./main.py