Task-Priority kinematic control Obstacle avoidance and Joint limit

Raúl López Musito

March 16, 2025

Introduction

This lab report presents the implementation of a priority control task for a 3-degree-of-freedom (3-DOF) planar robot Fig 1. The focus is on integrating an obstacle avoidance task and a joint limiting task while the robot attempts to reach a specified 2D target point. A hierarchical control approach is used to ensure that critical constraints, such as collision avoidance and joint limit enforcement, take priority while still allowing the robot to achieve its objective. The lower-priority task operates within the null space left by the higher-priority tasks, ensuring that it does not interfere with more critical constraints.

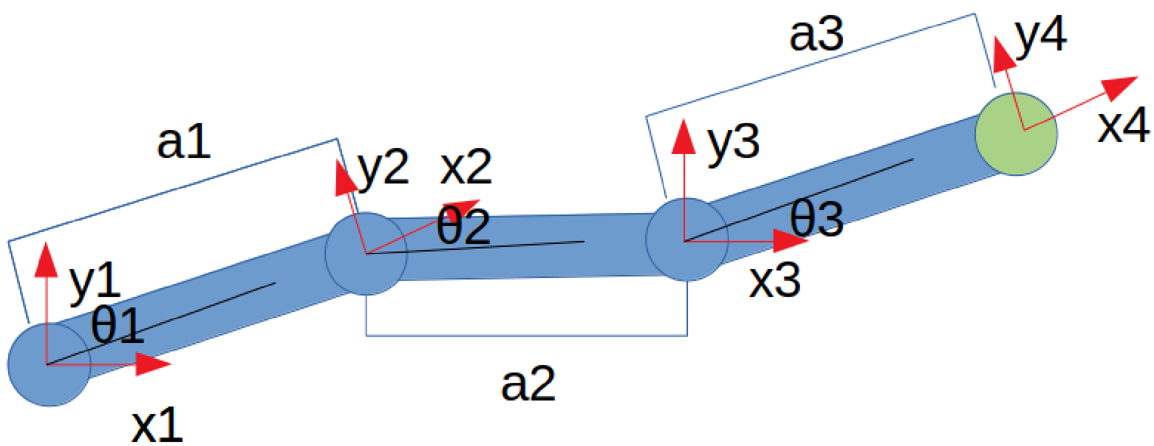

Fig 1. DH in the robot.

Obstacle avoidance:

Activating the obstacle avoidance task at all times can lead to inefficient robot movements or, in some cases, prevent the robot from reaching its target due to task prioritization constraints. To prevent this issue, the obstacle avoidance task is only activated within a specific range, ensuring that it affects the robot's motion only when necessary. This activation range is controlled using a hysteresis-based logic, which includes both an activation threshold and a deactivation threshold. The task is triggered when the robot comes too close to an obstacle and deactivates once it moves a safe distance away. This prevents frequent, unnecessary activations that could lead to erratic movements (chatter).

The activation of the obstacle avoidance task is determined using a variable, σ, which represents the direction in which the robot should move to avoid the obstacle. This value is calculated as the vector from the end effector to the center of the goal, divided by its norm, resulting in a unit vector pointing away from the obstacle. The activation and deactivation thresholds are set based on the distance from the center of the obstacle, tuning these parameters lead to smoother trajectories. The Jacobian used for this task is the same as the one for the 2D position control, as both tasks control the x and y coordinates of the robot's last joint.

Due to the way the obstacle avoidance task is implemented, it is only performed by the last joint (end effector, EE), and the rest of the arm is not taken into consideration. This limitation can be addressed by adding multiple obstacles along the arm's trajectory. However, as presented in this report, the current approach is specifically suited for SCARA-type robots, where the focus is on controlling the end effector's position. As demonstrated in Figure 2, the robot successfully reaches the target position, avoiding all obstacles, unless the randomly generated target is located inside an obstacle.

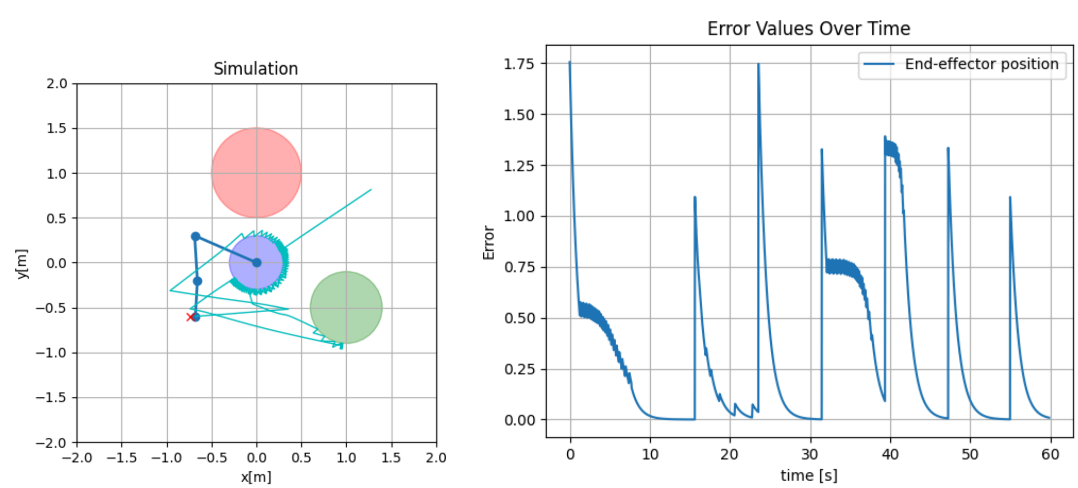

Fig 2. Reaching a 2d point avoiding obstacles.

Joint limit:

Similarly to the obstacle avoidance task, an activation function is implemented for both the upper and lower joint limits. This function includes activation and deactivation thresholds to ensure the task only engages when necessary. The activation logic is based on comparing the current joint position (σ) with predefined upper and lower limits, incorporating both activation and deactivation thresholds. If the joint position exceeds the lower limit minus the activation threshold, the task is activated in the negative direction, if the joint position is below the upper limit plus the activation threshold, the task is activated in the positive direction.

The error value is always either 1, 0, or -1, depending on the sign of the activation function, which is determined directly by whether the joint position exceeds the specified upper or lower limits. If the joint moves away from the limit beyond the deactivation threshold, the task is deactivated, ensuring the robot operates within safe bounds and prevents unnecessary or excessive corrections.

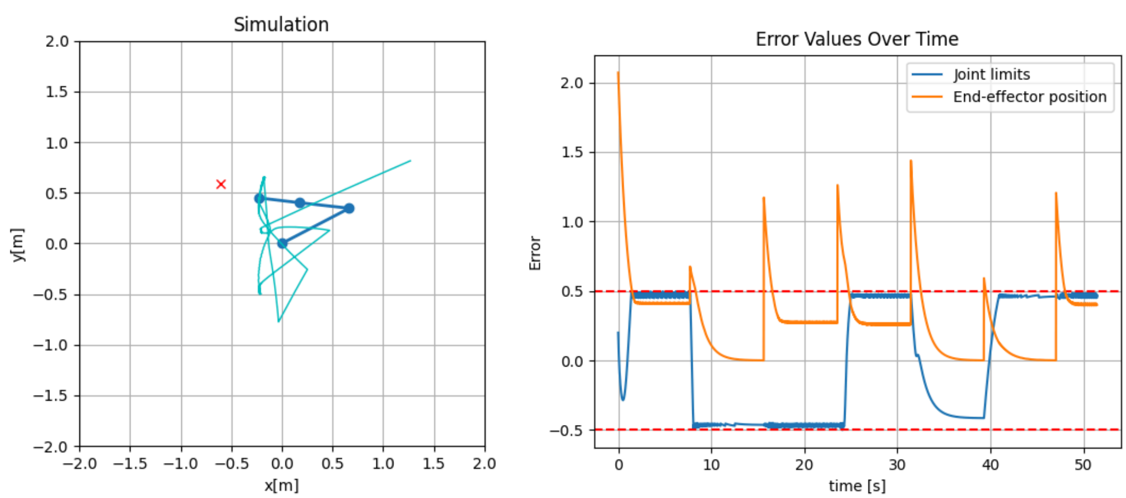

Fig 3. Reaching a 2d point limiting joint 1 angle.

As can be seen in Fig 3, the joint limit control works well, preventing the first joint from exceeding the set limits. This limitation significantly reduces the robot's operative space. As a result, the robot is unable to reach the desired position in several instances. The red lines in the figure represent the upper and lower joint limits, and it is evident that the joint (represented by the blue line) does not exceed these limits. This constraint ensures safety but also demonstrates the trade-off between maintaining joint limits and limiting the robot’s range of motion, which can impact its ability to reach certain target positions. It can be clearly seen that when the joint reaches the limit the end effector position error starts and asymptotic behavior.

Code:

Control Loop

```
from lab4_robotics import * # Includes numpy import
import matplotlib.pyplot as plt
import matplotlib.animation as anim
import matplotlib.patches as patch
import time

# Robot model
d = np.zeros(3)                                         # displacement along Z-axis
theta = np.array([0.2,0.5,0.4]).reshape(1,3)[0]         # rotation around Z-axis (q)
alpha = np.zeros(3)                                     # displacement along X-axis
a = np.array([0.75, 0.5, 0.4]).reshape(1,3)[0]          # rotation around X-axis
revolute = [True,True,True]                             # flags specifying the type of
joints
robot = Manipulator(d, theta, a, alpha, revolute)       # Manipulator object

# Task hierarchy definition

# Obstacle 1
obstaclePos_1 = np.array([0.0, 1.0]).reshape(2,1)
obstacleR_1 = 0.5

# Obstacle 2
obstaclePos_2 = np.array([0.0, 0.0]).reshape(2,1)
obstacleR_2 = 0.3

# Obstacle 3
obstaclePos_3 = np.array([1.0, -0.5]).reshape(2,1)
obstacleR_3 = 0.4

obstacle_vec = np.array([obstaclePos_1, obstaclePos_2, obstaclePos_3])
obstacle_r = np.array([obstacleR_1, obstacleR_2, obstacleR_3])
obstacle_color = ['red', 'blue', 'green']
limits = np.array([0.5, -0.5])

# !!!!!! ensure the last task is the 2D position task
# task definition
tasks = [
       Obstacle2D("Obstacle avoidance",obstaclePos_1, np.array([obstacleR_1,
obstacleR_1+0.05]), robot),
       Obstacle2D("Obstacle avoidance",obstaclePos_2, np.array([obstacleR_2,
obstacleR_2+0.05]), robot),
```

```
       Obstacle2D("Obstacle avoidance",obstaclePos_3, np.array([obstacleR_3,
obstacleR_3+0.05]), robot),
       JointLimit2D("Joint limits", 1, limits, tresholds=[0.03, 0.035]),
       Position2D("End-effector position",robot, 3)
       ]

# Simulation params
dt = 1.0/60.0
Tt = 1000
tt = np.arange(0, Tt, dt)
start_time = time.time()    # Start time

# Drawing preparation
fig = plt.figure()
ax = fig.add_subplot(111, autoscale_on=False, xlim=(-2, 2), ylim=(-2,2))
ax.set_title('Simulation')
ax.set_aspect('equal')
ax.grid()
ax.set_xlabel('x[m]')
ax.set_ylabel('y[m]')
for i in range(len(obstacle_vec)):
   obstacle_pos = obstacle_vec[i]
   obstacle_rad = obstacle_r[i]
   ax.add_patch(patch.Circle(obstacle_pos.flatten(), obstacle_rad,
color=obstacle_color[i], alpha=0.3))
line, = ax.plot([], [], 'o-', lw=2) # Robot structure
path, = ax.plot([], [], 'c-', lw=1) # End-effector path
point, = ax.plot([], [], 'rx') # Target

PPx = []
PPy = []

# Simulation initialization
def init():
   global tasks
   line.set_data([], [])
   path.set_data([], [])
   point.set_data([], [])
   return line, path, point

# Simulation loop
def simulate(t):
   global tasks
   global robot
   global PPx, PPy
   global start_time
   ### Recursive Task-Priority algorithm (w/set-based tasks)
```

```
    # The algorithm works in the same way as in Lab4.
    # The only difference is that it checks if a task is active.
    ###

    null_space = np.eye(robot.dof)                    # initial null space P (projector)
    dq = np.zeros(robot.dof).reshape(-1, 1)           # initial quasi-velocities

    for i in tasks:
        i.update(robot)                               # update task Jacobian and error
        if i.isActive():
            """ print ("i.getJacobian(): ", i.J)
            print ("null_space: ", null_space) """
            J = i.getJacobian()             # task full Jacobian
            Jbar = (J @ null_space)                         # projection of task in
null-space
            Jbar_inv = DLS(Jbar, 0.1)                       # pseudo-inverse or DLS
            """ print ("Jbar_inv: ", Jbar_inv)
            print ("j@dq: ", J@dq)
            print ("i.getError(): ", i.getError()) """
            print ("k: ", i.getK())
            dq += Jbar_inv @ ((i.getK()@i.getError()-J@dq) + i.ff)       # calculate
quasi-velocities with null-space tasks execution
            null_space = null_space - np.linalg.pinv(Jbar) @ Jbar   # update null-space
projector

    current_time = time.time()
    # Verify 10 sec
    if (current_time - start_time) >= 5: #or errorvec1[-1] < 0.1:
        start_time = current_time  # Reiniciar el tiempo
        tasks[-1].setRandomDesired() # the last task always has to be the 2d position

    # Update robot
    robot.update(dq, dt)

    # Update drawing
    PP = robot.drawing()
    line.set_data(PP[0,:], PP[1,:])
    PPx.append(PP[0,-1])
    PPy.append(PP[1,-1])
    path.set_data(PPx, PPy)
    point.set_data(tasks[-1].getDesired()[0], tasks[-1].getDesired()[1])

    return line, path, point

# Run simulation
animation = anim.FuncAnimation(fig, simulate, np.arange(0, Tt, dt),
                               interval=10, blit=True, init_func=init, repeat=True)
```

```
plt.show()

# Plot errors
plt.figure()
for i in tasks:
   if type(i) is Configuration2D:
       plt.plot(tt[:len(i.erroVec[0])], i.erroVec[0], label='Position Error')
       plt.plot(tt[:len(i.erroVec[1])], i.erroVec[1], label='Angular Error')
   elif type(i) is Obstacle2D :
       continue
   elif type(i) is JointLimit2D:
       plt.plot(tt[:len(robot.story)], robot.story, label=i.name)
   else:
       plt.plot(tt[:len(i.erroVec)], i.erroVec, label=i.name)

# paint line indicating limit of the angles
plt.axhline(y=limits[0], color='r', linestyle='--')
plt.axhline(y=limits[1], color='r', linestyle='--')
plt.title('Error Values Over Time')
plt.xlabel('time [s]')
plt.ylabel('Error')
plt.grid()

plt.legend()
plt.show()
```

Task Classes

```
from lab2_robotics import * # Includes numpy import


def jacobianLink(T, revolute, link): # Needed in Exercise 2
   '''
       Function builds a Jacobian for the end-effector of a robot,
       described by a list of kinematic transformations and a list of joint types.

       Arguments:
       T (list of Numpy array): list of transformations along the kinematic chain of
the robot (from the base frame)
       revolute (list of Bool): list of flags specifying if the corresponding joint is
a revolute joint
       link(integer): index of the link for which the Jacobian is computed

       Returns:
       (Numpy array): end-effector Jacobian
```

```
    '''
    # Code almost identical to the one from lab2_robotics...
    # 1. Initialize J and O.
    # 2. For each joint of the robot
    #   a. Extract z and o.
    #   b. Check joint type.
    #   c. Modify corresponding column of J.

    # Build the jacobian matrix
    J = np.zeros((6, link))
    O = T[-1][0:3,3] # End-effector position

    for i in range(link): # For each joint
        z = T[i][0:3,2] # Extract z
        Oi = T[i][0:3,3] # Extract o

        # Check joint type
        if revolute[i]:
            J[:,i] = np.concatenate((np.cross(z, O - Oi), z)) # Build the jacobian
matrix by asseigassigning the columns
        else:
            J[:,i] = np.concatenate((z, np.zeros(3)))
    return J

'''
    Class representing a robotic manipulator.
'''
class Manipulator:
    '''
        Constructor.

        Arguments:
        d (Numpy array): list of displacements along Z-axis
        theta (Numpy array): list of rotations around Z-axis
        a (Numpy array): list of displacements along X-axis
        alpha (Numpy array): list of rotations around X-axis
        revolute (list of Bool): list of flags specifying if the corresponding joint is
a revolute joint
    '''
    def __init__(self, d, theta, a, alpha, revolute):
        self.d = d
        self.theta = theta
        self.a = a
        self.alpha = alpha
        self.revolute = revolute
        self.dof = len(self.revolute)
        self.q = np.array(theta).reshape(-1, 1)
```

```
        self.story = []
        self.update(0.0, 0.0)


    '''
        Method that updates the state of the robot.

        Arguments:
        dq (Numpy array): a column vector of joint velocities
        dt (double): sampling time
    '''
    def update(self, dq, dt):
        self.q += dq * dt
        self.story.append(self.q[0][0])
        for i in range(len(self.revolute)):
            if self.revolute[i]:
                self.theta[i] = self.q[i]
            else:
                self.d[i] = self.q[i]
        self.T = kinematics(self.d, self.theta, self.a, self.alpha)

    '''
        Method that returns the characteristic points of the robot.
    '''
    def drawing(self):
        return robotPoints2D(self.T)

    '''
        Method that returns the end-effector Jacobian.
    '''
    def getEEJacobian(self):
        return jacobian(self.T, self.revolute)

    def getLINKJacobian(self, link):
        return jacobianLink(self.T, self.revolute, link)

    '''
        Method that returns the end-effector transformation.
    '''
    def getEETransform(self):
        return self.T[-1]

    '''
        Method that returns the position of a selected joint.

        Argument:
        joint (integer): index of the joint
```

```python
        Returns:
        (double): position of the joint
    '''
    def getJointPos(self, joint):
        return self.q[joint]

    '''
        Method that returns number of DOF of the manipulator.
    '''
    def getDOF(self):
        return self.dof

    def getLinkTransform(self, link):
        return self.T[link]


    def getLinkOrientation(self, link):
        linkT = self.getLinkTransform(link)
        return np.array(np.arctan2(linkT[1,0], linkT[0,0])).reshape((1,1))

'''
    Base class representing an abstract Task.
'''
class Task:
    '''
        Constructor.

        Arguments:
        name (string): title of the task
        desired (Numpy array): desired sigma (goal)
    '''
    def __init__(self, name, desired):
        self.name = name # task title
        self.sigma_d = desired # desired sigma
        self.erroVec = [] # error vector
        #self.l
        self.ff = None
        self.k = None

    def getFF(self):
        return self.ff

    def setFF(self, ff):
        self.ff = ff

    def getK(self):
```

```
        return self.k

    def setK(self, k):
        self.k = k
    '''
        Method updating the task variables (abstract).

        Arguments:
        robot (object of class Manipulator): reference to the manipulator
    '''
    def update(self, robot):
        pass

    '''
        Method setting the desired sigma.

        Arguments:
        value(Numpy array): value of the desired sigma (goal)
    '''
    def setDesired(self, value):
        self.sigma_d = value

    '''
        Method returning the desired sigma.
    '''
    def getDesired(self):
        return self.sigma_d

    '''
        Method returning the task Jacobian.
    '''
    def getJacobian(self):
        return self.J

    '''
        Method returning the task error (tilde sigma).
    '''
    def getError(self):
        return self.err

    """
        Method that activation state of the task.
    """
    def isActive (self):
        return self.active

'''
```

```
   Subclass of Task, representing the 2D position task.
'''
class Position2D(Task):
   def __init__(self, name, robot: Manipulator, link):
       super().__init__(name, self.setRandomDesired())
       self.J = np.zeros((2,robot.getDOF()))                    # Initialize with proper
dimensions
       self.err = np.zeros((2,1))                                 # Initialize with
proper dimensions
       self.setK(np.eye(2))
       self.setFF(np.zeros((2,1)))
       self.link = link
       self.active = True

   def update(self, robot: Manipulator):
       self.J = robot.getLINKJacobian(self.link)[:2,:].reshape((2,self.link))
# Update task Jacobian
       self.J = np.hstack((self.J, np.zeros((2, robot.dof - self.link))))
       self.err = np.array(self.getDesired() -
robot.getLinkTransform(self.link)[0:2,3].reshape((2,1))) # Update task error
       self.erroVec.append(np.linalg.norm(self.err))

   def setRandomDesired(self):
       random = (np.random.rand(2,1)*2-1).reshape((2,1))
       self.setDesired(random)
       return random

'''
   Subclass of Task, representing the 2D orientation task.
'''
class Orientation2D(Task):
   def __init__(self, name, desired, robot: Manipulator, link):
       super().__init__(name, desired)
       self.J = np.zeros((1,robot.getDOF()))# Initialize with proper dimensions
       self.err = np.zeros((1,1))# Initialize with proper dimensions
       self.setK(np.eye(1))
       self.setFF(np.zeros((1,1)))
       self.link = link


   def update(self, robot: Manipulator):
       self.J = robot.getLINKJacobian(self.link)[5,:].reshape((1,self.link))    #
Update task Jacobian
       self.J = np.pad(self.J, (0, robot.dof - self.link), mode='constant',
constant_values=0)
       current_transform = robot.getLinkTransform(self.link) # Compute current sigma
```

```
        current_sigma = np.array(np.arctan2(current_transform[1,0],
current_transform[0,0])).reshape((1,1)) # Compute current sigma
        print('current_sigma:',current_sigma)
        self.err = wrapangle(self.getDesired() - current_sigma.reshape((1,1))) # Update
task error
        print ("angular error: ", self.err)
        self.erroVec.append(self.err[0])
        pass # to remove

    def setRandomDesired(self):
        self.setDesired( (np.random.rand(1,1)*2*np.pi-np.pi).reshape((1,1)))
        self.setDesired(np.array([np.pi]).reshape(1,1))
        pass
'''
    Subclass of Task, representing the 2D configuration task.
'''
class Configuration2D(Task):
    def __init__(self, name, desired, robot: Manipulator, link):
        super().__init__(name, desired)
        self.J = np.zeros((3,robot.getDOF()))# Initialize with proper dimensions
        self.err = np.zeros((3,1))# Initialize with proper dimensions
        self.erroVec = [[],[]]
        self.setK(np.eye(3))
        self.setFF(np.zeros((3,1)))
        self.link = link



    def update(self, robot: Manipulator):
        positionJacobian =
robot.getLINKJacobian(self.link)[:2,:].reshape((2,self.link))                      #
Update task Jacobian
        positionJacobian = np.hstack((positionJacobian, np.zeros((2, robot.dof -
self.link))))
        self.J[0:2,:] = positionJacobian # Update task Jacobian

        orientationJacobian =
robot.getLINKJacobian(self.link)[5,:].reshape((1,self.link))   # Update task Jacobian
        orientationJacobian =  np.hstack((orientationJacobian, np.zeros((1, robot.dof -
self.link))))
        self.J[2,:] = orientationJacobian

        current_transform = robot.getLinkTransform(self.link) # Compute current sigma
        current_sigma_angle = np.arctan2(current_transform[1,0],
current_transform[0,0]) # Compute current sigma angle
        current_sigma_pos = current_transform[0:2,3] # Compute current sigma position
```

```python
        error_pos = self.getDesired()[0:2] - current_sigma_pos.reshape((2,1)) # Compute position error
        error_angle = self.getDesired()[2] - current_sigma_angle
        print ("angular error: ", error_angle)
        self.err = np.array([error_pos[0], error_pos[1], error_angle]).reshape((3,1)) # Update task error
        self.erroVec[0].append(np.linalg.norm(error_pos))
        self.erroVec[1].append(error_angle[0])
        pass
    def setRandomDesired(self):
        self.setDesired(np.array([np.random.rand(1,1)*2-1,np.random.rand(1,1)*2-1, np.random.rand(1,1)*2*np.pi-np.pi]).reshape((3,1)))
        pass

'''
    Subclass of Task, representing the joint position task.
'''
class JointPosition(Task):
    def __init__(self, name, desired, robot: Manipulator, link):
        super().__init__(name, desired)
        self.link = link
        self.J = np.zeros((1,self.link))# Initialize with proper dimensions
        self.err = np.zeros((1,1))# Initialize with proper dimensions
        self.setK(np.eye(1))
        self.setFF(np.zeros((1,1)))



    def update(self, robot: Manipulator):
        self.J = robot.getLINKJacobian(self.link)[5,:].reshape((1,self.link))    # Update task Jacobian
        self.J = np.pad(self.J, (0, robot.dof - self.link), mode='constant', constant_values=0)
        current_sigma = robot.getLinkOrientation(self.link) # Compute current sigma
        print('current_sigma:',current_sigma)
        self.err = wrapangle(self.getDesired() - current_sigma.reshape((1,1))) # Update task error
        print ("angular error: ", self.err)
        self.erroVec.append(self.err[0])
        pass # to remove

    def setRandomDesired(self):
        self.setDesired( (np.random.rand(1,1)*2*np.pi-np.pi).reshape((1,1)))
        pass

'''
    Subclass of Task, representing the Obstacle avoidance task.
```

```python
'''
class Obstacle2D(Task):
    def __init__(self, name, position, thresholds, robot: Manipulator,):
        super().__init__(name, None)
        self.position = position
        self.activation_tresh = thresholds[0]
        self.deactivation_tresh = thresholds[1]
        self.J = np.zeros((2,robot.dof))# Initialize with proper dimensions
        self.err = np.zeros((1,1))# Initialize with proper dimensions
        self.active = False
        self.setK(np.eye(2))
        self.setFF(np.zeros((2,1)))

    def activate (self, sigma):
        if self.active == False and np.linalg.norm(sigma) <= self.activation_tresh:
            self.active = True
        elif self.active == True and np.linalg.norm(sigma) >= self.deactivation_tresh:
            self.active = False

    def update(self, robot: Manipulator):
        self.J = robot.getEEJacobian()[:2,:].reshape((2,robot.dof))    # Update task
Jacobian
        current_sigma = robot.getEETransform()[:2,3].reshape((2,1)) - self.position#g
get EE x & y
        self.activate(current_sigma)
        print("current_sigma: ",current_sigma)
        self.err = current_sigma/np.linalg.norm(current_sigma)
        print ("angular error: ", self.err)
        pass # to remove

class JointLimit2D(Task):
    def __init__(self, name, link, limits, tresholds):
        super().__init__(name, None)
        self.link = link
        self.limits = limits
        self.activation_tresh = tresholds[0]
        self.deactivation_tresh = tresholds[1]
        self.J = np.zeros((1,self.link))# Initialize with proper dimensions
        self.err = np.zeros((1,1))# Initialize with proper dimensions
        self.setK(np.eye(1))
        self.setFF(np.zeros((1,1)))
        self.active = 0

    def update(self, robot: Manipulator):
        self.J = robot.getLINKJacobian(self.link)[5,:].reshape((1,self.link))    #
Update task Jacobian
```

```
        self.J = np.pad(self.J, (0, robot.dof - self.link), mode='constant',
constant_values=0)
        current_sigma = robot.getLinkOrientation(self.link) # Compute current sigma
        self.activate(current_sigma)
        print('current_sigma:',current_sigma)
        self.err = np.array([1*self.active]) # Update task error
        print ("angular error: ", self.err)
        self.erroVec.append(self.err)
        pass # to remove


    def activate (self, angle):
        if self.active == 0 and angle >=  self.limits[0] - self.activation_tresh:
            self.active = -1
        elif self.active == 0 and angle <= self.limits[1] + self.activation_tresh:
            self.active = 1
        elif self.active == -1 and angle <= self.limits[0] - self.deactivation_tresh:
            self.active = 0
        elif self.active == 1 and angle >= self.limits[1] + self.deactivation_tresh:
            self.active = 0

    def isActive(self):
        return abs(self.active)


    def setRandomDesired(self):
        self.setDesired( (np.random.rand(1,1)*2*np.pi-np.pi).reshape((1,1)))
        pass
```