The goal of this exercise was to compute the current position of the quadrotor by integrating its velocity measurements over time.

Using the plot\_trajectory function to visualize the estimated position. The first parameter is the name of the trajectory. The second parameter is a 2 or 3 dimensional column vector containing the current x, y and optionally z position.

import numpy as np

from math import sin, cos

from plot import plot\_trajectory

class UserCode:

def \_\_init\_\_(self):

self.position = np.array([[0], [0]])

def measurement\_callback(self, t, dt, navdata):

'''

:param t: time since simulation start

:param dt: time since last call to measurement\_callback

:param navdata: measurements of the quadrotor

'''

# TODO: update self.position by integrating measurements contained in navdata

pos\_local=np.array([[navdata.vx\*dt],[navdata.vy\*dt]])

R=np.array([[cos(navdata.rotZ),-1.0\*sin(navdata.rotZ)],[sin(navdata.rotZ),cos(navdata.rotZ)]])

pos\_global=np.dot(R,pos\_local)

self.position+=pos\_global

plot\_trajectory("odometry", self.position)

In this exercise we wanted to compute the position of the quadrotor from observations of visual markers. The markers are detected in images recorded from the downfacing camera of the quadrotor. We also assume the position of the markers in the world are known.

The transformations from the world to the marker

**T**

*M*

*W*

and from the quadrotor to the marker

**T**

*M*

*Q*

are known. task is to compute the transformation from the world to the quadrotor

**T**

*Q*

*W*

given the two other transformations.

import numpy as np

class Pose3D:

def \_\_init\_\_(self, rotation, translation):

self.rotation = rotation

self.translation = translation

def inv(self):

'''

Inversion of this Pose3D object

:return inverse of self

'''

# TODO: implement inversion

inv\_rotation = self.rotation.transpose()

inv\_translation = -np.dot(inv\_rotation, self.translation)

return Pose3D(inv\_rotation, inv\_translation)

def \_\_mul\_\_(self, other):

'''

Multiplication of two Pose3D objects, e.g.:

a = Pose3D(...) # = self

b = Pose3D(...) # = other

c = a \* b # = return value

:param other: Pose3D right hand side

:return product of self and other

'''

# TODO: implement multiplication

self\_rotation= np.dot(self.rotation,other.rotation)

self\_translation = np.dot(self.rotation, other.translation)+self.translation

return Pose3D(self\_rotation, self\_translation)

#return Pose3D(self.rotation, self.translation)

def \_\_str\_\_(self):

return "rotation:\n" + str(self.rotation) + "\ntranslation:\n" + str(self.translation.transpose())

def compute\_quadrotor\_pose(global\_marker\_pose, observed\_marker\_pose):

'''

:param global\_marker\_pose: Pose3D

:param observed\_marker\_pose: Pose3D

:return global quadrotor pose computed from global\_marker\_pose and observed\_marker\_pose

'''

# TODO: implement global quadrotor pose computation

global\_quadrotor\_pose = (global\_marker\_pose\*observed\_marker\_pose.inv())

return global\_quadrotor\_pose

import math

import numpy as np

from plot import plot\_trajectory, plot\_point, plot\_covariance\_2d

class UserCode:

def \_\_init\_\_(self):

#TODO: Play with the noise matrices

#process noise

pos\_noise\_std = 0.005

yaw\_noise\_std = 0.005

self.Q = np.array([

[pos\_noise\_std\*pos\_noise\_std,0,0],

[0,pos\_noise\_std\*pos\_noise\_std,0],

[0,0,yaw\_noise\_std\*yaw\_noise\_std]

])

#measurement noise

z\_pos\_noise\_std = 0.005

z\_yaw\_noise\_std = 0.03

self.R = np.array([

[z\_pos\_noise\_std\*z\_pos\_noise\_std,0,0],

[0,z\_pos\_noise\_std\*z\_pos\_noise\_std,0],

[0,0,z\_yaw\_noise\_std\*z\_yaw\_noise\_std]

])

# state vector [x, y, yaw] in world coordinates

self.x = np.zeros((3,1))

# 3x3 state covariance matrix

self.sigma = 0.01 \* np.identity(3)

def rotation(self, yaw):

'''

create 2D rotation matrix from given angle

'''

s\_yaw = math.sin(yaw)

c\_yaw = math.cos(yaw)

return np.array([

[c\_yaw, -s\_yaw],

[s\_yaw, c\_yaw]

])

def normalizeYaw(self, y):

'''

normalizes the given angle to the interval [-pi, +pi]

'''

while(y > math.pi):

y -= 2 \* math.pi

while(y < -math.pi):

y += 2 \* math.pi

return y

def visualizeState(self):

# visualize position state

plot\_trajectory("kalman", self.x[0:2])

plot\_covariance\_2d("kalman", self.sigma[0:2,0:2])

def predictState(self, dt, x, u\_linear\_velocity, u\_yaw\_velocity):

'''

predicts the next state using the current state and

the control inputs local linear velocity and yaw velocity

'''

x\_p = np.zeros((3, 1))

x\_p[0:2] = x[0:2] + dt \* np.dot(self.rotation(x[2]), u\_linear\_velocity)

x\_p[2] = x[2] + dt \* u\_yaw\_velocity

x\_p[2] = self.normalizeYaw(x\_p[2])

return x\_p

def calculatePredictStateJacobian(self, dt, x, u\_linear\_velocity, u\_yaw\_velocity):

'''

calculates the 3x3 Jacobian matrix for the predictState(...) function

'''

s\_yaw = math.sin(x[2])

c\_yaw = math.cos(x[2])

dRotation\_dYaw = np.array([

[-s\_yaw, -c\_yaw],

[ c\_yaw, -s\_yaw]

])

F = np.identity(3)

F[0:2, 2] = dt \* np.dot(dRotation\_dYaw, u\_linear\_velocity)

return F

def predictCovariance(self, sigma, F, Q):

'''

predicts the next state covariance given the current covariance,

the Jacobian of the predictState(...) function F and the process noise Q

'''

return np.dot(F, np.dot(sigma, F.T)) + Q

def calculateKalmanGain(self, sigma\_p, H, R):

'''

calculates the Kalman gain

'''

return np.dot(np.dot(sigma\_p, H.T), np.linalg.inv(np.dot(H, np.dot(sigma\_p, H.T)) + R))

def correctState(self, K, x\_predicted, z, z\_predicted):

'''

corrects the current state prediction using Kalman gain, the measurement and the predicted measurement

:param K - Kalman gain

:param x\_predicted - predicted state 3x1 vector

:param z - measurement 3x1 vector

:param z\_predicted - predicted measurement 3x1 vector

:return corrected state as 3x1 vector

'''

# TODO: implement correction of predicted state x\_predicted

x\_predicted = x\_predicted + np.dot(K,(z - z\_predicted))

return x\_predicted

def correctCovariance(self, sigma\_p, K, H):

'''

corrects the sate covariance matrix using Kalman gain and the Jacobian matrix of the predictMeasurement(...) function

'''

return np.dot(np.identity(3) - np.dot(K, H), sigma\_p)

def predictMeasurement(self, x, marker\_position\_world, marker\_yaw\_world):

'''

predicts a marker measurement given the current state and the marker position and orientation in world coordinates

'''

z\_predicted = Pose2D(self.rotation(x[2]), x[0:2]).inv() \* Pose2D(self.rotation(marker\_yaw\_world), marker\_position\_world);

return np.array([[z\_predicted.translation[0], z\_predicted.translation[1], z\_predicted.yaw()]]).T

def calculatePredictMeasurementJacobian(self, x, marker\_position\_world, marker\_yaw\_world):

'''

calculates the 3x3 Jacobian matrix of the predictMeasurement(...) function using the current state and

the marker position and orientation in world coordinates

:param x - current state 3x1 vector

:param marker\_position\_world - x and y position of the marker in world coordinates 2x1 vector

:param marker\_yaw\_world - orientation of the marker in world coordinates

:return - 3x3 Jacobian matrix of the predictMeasurement(...) function

'''

# TODO: implement computation of H

H = np.array([[-math.cos(x[2]), -math.sin(x[2]), np.dot(-(marker\_position\_world[0]-x[0]), math.sin(x[2])) + np.dot((marker\_position\_world[1]-x[1]), math.cos(x[2]))], [math.sin(x[2]), -math.cos(x[2]), np.dot(-(marker\_position\_world[0]-x[0]), math.cos(x[2])) - np.dot((marker\_position\_world[1]-x[1]), math.sin(x[2]))], [0, 0, -1]])

return H

return np.zeros((3,3))

def state\_callback(self, t, dt, linear\_velocity, yaw\_velocity):

'''

called when a new odometry measurement arrives approx. 200Hz

:param t - simulation time

:param dt - time difference this last invocation

:param linear\_velocity - x and y velocity in local quadrotor coordinate frame (independet of roll and pitch)

:param yaw\_velocity - velocity around quadrotor z axis (independet of roll and pitch)

'''

self.x = self.predictState(dt, self.x, linear\_velocity, yaw\_velocity)

F = self.calculatePredictStateJacobian(dt, self.x, linear\_velocity, yaw\_velocity)

self.sigma = self.predictCovariance(self.sigma, F, self.Q);

self.visualizeState()

def measurement\_callback(self, marker\_position\_world, marker\_yaw\_world, marker\_position\_relative, marker\_yaw\_relative):

'''

called when a new marker measurement arrives max 30Hz, marker measurements are only available if the quadrotor is

sufficiently close to a marker

:param marker\_position\_world - x and y position of the marker in world coordinates 2x1 vector

:param marker\_yaw\_world - orientation of the marker in world coordinates

:param marker\_position\_relative - x and y position of the marker relative to the quadrotor 2x1 vector

:param marker\_yaw\_relative - orientation of the marker relative to the quadrotor

'''

z = np.array([[marker\_position\_relative[0], marker\_position\_relative[1], marker\_yaw\_relative]]).T

z\_predicted = self.predictMeasurement(self.x, marker\_position\_world, marker\_yaw\_world)

H = self.calculatePredictMeasurementJacobian(self.x, marker\_position\_world, marker\_yaw\_world)

K = self.calculateKalmanGain(self.sigma, H, self.R)

self.x = self.correctState(K, self.x, z, z\_predicted)

self.sigma = self.correctCovariance(self.sigma, K, H)

self.visualizeState()

class Pose2D:

def \_\_init\_\_(self, rotation, translation):

self.rotation = rotation

self.translation = translation

def inv(self):

'''

inversion of this Pose2D object

:return - inverse of self

'''

inv\_rotation = self.rotation.transpose()

inv\_translation = -np.dot(inv\_rotation, self.translation)

return Pose2D(inv\_rotation, inv\_translation)

def yaw(self):

from math import atan2

return atan2(self.rotation[1,0], self.rotation[0,0])

def \_\_mul\_\_(self, other):

'''

multiplication of two Pose2D objects, e.g.:

a = Pose2D(...) # = self

b = Pose2D(...) # = other

c = a \* b # = return value

:param other - Pose2D right hand side

:return - product of self and other

'''

return Pose2D(np.dot(self.rotation, other.rotation), np.dot(self.rotation, other.translation) + self.translation)