**Python program for Model Predictive Control (MPC) using CasADi library:**

import casadi as cs

import numpy as np

import matplotlib.pyplot as plt

# Define the system dynamics

A = np.array([[1, 1], [0, 1]])

B = np.array([[0], [1]])

C = np.array([[1, 0], [0, 1]])

D = np.array([[0], [0]])

# Define the MPC parameters

N = 5

dt = 0.1

Q = np.diag([1, 1])

R = np.array([[1]])

# Define the optimization problem

opti = cs.Opti()

# Define the state variables

x = opti.variable(2, N+1)

x0 = opti.parameter(2, 1)

# Define the control variables

u = opti.variable(1, N)

# Define the reference trajectory

x\_ref = opti.parameter(2, N+1)

u\_ref = opti.parameter(1, N)

# Define the initial state constraint

opti.subject\_to(x[:,0] == x0)

# Define the dynamic constraints

for k in range(N):

x\_next = cs.mtimes(A, x[:,k]) + cs.mtimes(B, u[:,k])

opti.subject\_to(x[:,k+1] == x\_next)

# Define the cost function

J = 0

for k in range(N):

J += cs.mtimes([(x[:,k] - x\_ref[:,k]).T, Q, (x[:,k] - x\_ref[:,k])])

J += cs.mtimes([(u[:,k] - u\_ref[:,k]).T, R, (u[:,k] - u\_ref[:,k])])

opti.minimize(J)

# Define the control constraints

opti.subject\_to(u <= 1)

opti.subject\_to(u >= -1)

# Set the initial state parameter

x0\_val = np.array([[0], [0]])

opti.set\_value(x0, x0\_val)

# Define the reference trajectory and control inputs

x\_ref\_val = np.zeros((2, N+1))

x\_ref\_val[0,:] = np.linspace(0, 1, N+1)

u\_ref\_val = np.zeros((1, N))

opti.set\_value(x\_ref, x\_ref\_val)

opti.set\_value(u\_ref, u\_ref\_val)

# Simulate the system and plot the results

x\_val = np.zeros((2, N+1))

u\_val = np.zeros((1, N))

for i in range(N):

# Update the optimization problem with the current state

opti.set\_initial(u, u\_val)

opti.set\_initial(x, x\_val)

# Solve the optimization problem

sol = opti.solve()

# Extract the control input

u\_val = opti.value(u[:,0])

# Update the system state

x\_val[:,i+1] = np.squeeze(cs.mtimes(A, x\_val[:,i]) + cs.mtimes(B, u\_val))

# Plot the results

plt.plot(x\_ref\_val[0,:], x\_ref\_val[1,:], 'r--', label='Reference')

plt.plot(x\_val[0,:], x\_val[1,:], 'b', label='MPC')

plt.legend()

plt.xlabel('x1')

plt.ylabel('x2')

plt.show()

This program defines a simple linear system and uses MPC to track a reference trajectory. The optimization problem is defined using the CasADi library, and the resulting control inputs are used to simulate the system. The results are plotted using Matplotlib library

import time

from dronekit import connect, VehicleMode, LocationGlobalRelative

# Connect to the PX4 vehicle

connection\_string = 'udp:127.0.0.1:14550'

vehicle = connect(connection\_string, wait\_ready=True)

# Set the vehicle mode to GUIDED

vehicle.mode = VehicleMode("GUIDED")

# Arm the vehicle

vehicle.armed = True

while not vehicle.armed:

print("Waiting for vehicle to arm...")

time.sleep(1)

# Define the mission waypoints

waypoints = [

LocationGlobalRelative(-35.363261, 149.165230, 10),

LocationGlobalRelative(-35.362933, 149.164652, 10),

LocationGlobalRelative(-35.363275, 149.164340, 10),

LocationGlobalRelative(-35.363700, 149.164889, 10)

]

**This program sets a fixed altitude of 20 meters for all waypoints. The program also sets the vehicle mode to RTL (Return to Launch) instead of LAND, which will cause the vehicle to automatically return to its launch point and land.**

# Move to each waypoint in turn with a fixed altitude of 20 meters

for waypoint in waypoints:

# Set the target waypoint with a fixed altitude of 20 meters

target\_altitude = 20

target\_location = LocationGlobalRelative(waypoint.lat, waypoint.lon, target\_altitude)

vehicle.simple\_goto(target\_location)

# Wait for the vehicle to reach the waypoint

while True:

current\_pos = vehicle.location.global\_relative\_frame

dist = current\_pos.distance\_to(target\_location)

if dist < 1:

break

time.sleep(1)

# Set the vehicle mode to RTL (Return to Launch)

vehicle.mode = VehicleMode("RTL")

# Wait for the vehicle to return to the launch point and land

while vehicle.armed:

print("Waiting for vehicle to land...")

time.sleep(1)

# Disconnect from the vehicle

vehicle.close()

import time

from dronekit import connect, VehicleMode, LocationGlobalRelative

# Connect to the PX4 vehicle

connection\_string = 'udp:127.0.0.1:14550'

vehicle = connect(connection\_string, wait\_ready=True)

# Set the vehicle mode to GUIDED

vehicle.mode = VehicleMode("GUIDED")

# Arm the vehicle

vehicle.armed = True

while not vehicle.armed:

print("Waiting for vehicle to arm...")

time.sleep(1)

# Define the mission waypoints

waypoints = [

LocationGlobalRelative(-35.363261, 149.165230, 10),

LocationGlobalRelative(-35.362933, 149.164652, 15),

LocationGlobalRelative(-35.363275, 149.164340, 20),

LocationGlobalRelative(-35.363700, 149.164889, 10)

]

# **Move to each waypoint in turn with a varying altitude**

**for waypoint in waypoints:**

# Set the target waypoint with a varying altitude

target\_altitude = waypoints.index(waypoint) \* 5 + 10

target\_location = LocationGlobalRelative(waypoint.lat, waypoint.lon, target\_altitude)

vehicle.simple\_goto(target\_location)

# Wait for the vehicle to reach the waypoint

while True:

current\_pos = vehicle.location.global\_relative\_frame

dist = current\_pos.distance\_to(target\_location)

if dist < 1:

break

time.sleep(1)

# Set the vehicle mode to LAND

vehicle.mode = VehicleMode("LAND")

# Wait for the vehicle to land

while vehicle.armed:

print("Waiting for vehicle to land...")

time.sleep(1)

# Disconnect from the vehicle

vehicle.close()

This program is similar to the previous ones, but with a few additional features. Instead of using a fixed altitude for all waypoints, this program sets a varying altitude for each waypoint based on its index in the waypoints list. The program uses the index() function to get the index of the current waypoint, multiplies it by 5, and adds 10 to get the target altitude. This will cause the vehicle to fly at a progressively higher altitude as it moves through the mission waypoints.