C.O.M.A.S. Final Project Report

Aryan Choudhary

University of Colorado at Boulder

# Motivations:

With the progress in robotic technology, especially drones, impacting various fields like logistics and defense, there is a growing need for optimized and efficient algorithms to deploy these technologies in large numbers. A lot of work has covered individual control of such robotic applications, but multi-agent control remains a relatively new field. This project attempts to analyze a simple four-agent drone swarm (in this case the DJI Mavic Pro 2) and implementing known algorithms for rendezvous and emergent cyclic behavior, and subsequently analyzing them for efficiency.

# Main Problem Formulation:

The main modeling approach for the controller was based around a simple multi agent discrete time linear system.

…… (1)

This was responsible for modeling the control of each element of our state. For a grounded robot the state would consist of x and y coordinates, and orientation angle (). For an aerial drone, the states would be x, y, z, and yaw (). The states were represented as x and y co-ordinates.

A method of modeling grounded robots for rendezvous that we have seen is:

…… (2)

We know that this setup converges to the average of the positions (*Network Systems, F. Bullo, Second Example pg. 68*). Where [p] is a vector containing initial positions of p1, p2, p3, p4 in form, where x and y are co-ordinates of the robot. For faster rendezvous, we altered the matrix to yield values closer to the average on each iteration. Similarly, for other kinds of behavior, we altered the contents of to obtain behavior we are interested in. For example, in the case of cyclic pursuit (*Network Systems, F. Bullo, Appendix 1.6 pg. 11).* For constant circular motion, we could set the matrix to be equal to

We can intuitively see that this system would never be stable as each agent would try to reach the position of the next agent at each timestep. Underlying modeling approaches will include a PID wherever necessary for stable actuation.

Assumptions include:

1. Roll and pitch can be left out of multi-agent control as their deviation is unstable and rectified by the controller present in the code of the drone by default.
2. 2d models can translate to 3d models with the modification of state vector constituents.

The objectives are thus:

1. Obtain rendezvous in within feasible temporal constraints.
2. Observe cyclic behavior for the matrix discussed above.
3. Compare gain and time taken to rendezvous.
4. Compare matrices and time taken to rendezvous.

# Analysis:

The results were evaluated for successful runs only. There were occasionally runs where one of the drones would run off near or after convergence for reasons unknown at this time. Speculation suggests this was due to run-off values for disturbance that remained allocated. The results are as follows:

1. The convergence of the robots to the average position (within margin of error set to 0.5m) is seen based on Theorem 2.13 of the book and the corollary of this theorem which states the following result for a primitive row-stochastic matrix with dominant eigenvalue = 1.

…… (3)

Where is the dominant left eigenvector, is the initial state, and is the vector of all ones. As all the matrices we used were doubly stochastic, we can take this convergence one step further and say as is mentioned in section 11.2,

……(4)

In our case, they converge at the origin as is seen in the simulation (Figure 1,2,3) and verified in the code. For the doubly stochastic matrix in (2), the algorithm converged in 8 iterations. For a doubly stochastic matrix with eigenvalue 1/3, it converged in 3 iterations. For eigenvalue 1/4, it converged in 1 iteration. We also analyzed time taken to converge by eigenvalue of matrix (Figure 4) and time taken to converge by gain multiplied to the matrix in (2) (Figure 5). Based on our observations a smaller gain yields faster convergence and congruently a smaller eigenvalue does the same.

1. Similarly, we see the cyclic patrol to be in continuous motion (Movie 2), as the matrix is no longer primitive, it will not settle.
2. We also analyzed the per-step convergence rate for our various doubly stochastic matrices by the relation provided to us in the text (Section 11.2):

…… (5)

To implement this, the program calculated the highest fraction of the 2-norm of the next step over the previous step for each of the matrices implemented. As our destination was the origin, we need only worry about the positions at each step. Results obtained are presented in table below. The lowest convergence rate is achieved by the fastest converging matrix.

|  |  |
| --- | --- |
| Eigenvalue at Gain 1 |  |
| 0.25 | 0 |
| 0.33 | 0.333333 |
| 0.5 | 1 |

Table 1. Eigenvalue at gain one and corresponding

# Simulation:

The simulation was carried out using Webots which is a common physics-based simulator used in academia. The controlling programs for the robots were coded in python. A patrolling controller [2] was already available to us negating the need to code the actuation of the motors from scratch. This code accepted waypoints as inputs and moved the drones from one waypoint to another. The position matrices were iteratively multiplied with the matrix, and the co-ordinates obtained were used to fill the waypoints array. As Webots is a visual simulator, we were able to observe the results in real-time. Various data points were collected such as duration, gain, etc. and plotted to showcase the performance.

# Conclusions:

We can thus conclude that a smaller eigenvalue for a doubly stochastic matrix leads to faster convergence. We also saw the feasibility of multi-agent control on extant robotic technology and how it can automate drone swarms. Both our assumptions held true as roll and pitch disturbances were taken care of by the controller by default and altitude was kept constant throughout. Due to this, it did not affect multi-agent control (although this could have very easily been added to the list of controllable states and been made dynamic). All four of our objectives were achieved.

# References:

[1] Network Systems, *F.Bullo*

[2] Webots Documentation

# Appendix:

All rights to Webots and DJI belong to their respective owners.

## Figures:

Background pattern

Description automatically generated

Figure 1. Start position of the robots.

A picture containing text, outdoor object, night sky

Description automatically generated

Figure 2. Robots mid-flight.

 A group of light bulbs

Description automatically generated with low confidence

Figure 3. On convergence. Oblique and top view.

Figure 4. Duration by eigenvalue of

Figure 5. Duration by gain for in (2)

## Movies:

Movie showing convergence for eigenvalue of 0.5:

<https://drive.google.com/file/d/1DTfIagbMBiMQ4PUxj8QLwOMpe6ytwu_V/view?usp=share_link>

Movie showing cyclical behavior:

<https://drive.google.com/file/d/1ZIl4V05jhV9nCGilF0BzE0LOwTMzNKxw/view?usp=share_link>

## Code for matrix with eigenvalue 1/4:

# Copyright 1996-2022 Cyberbotics Ltd.

#

# Licensed under the Apache License, Version 2.0 (the "License");

# you may not use this file except in compliance with the License.

# You may obtain a copy of the License at

#

# http://www.apache.org/licenses/LICENSE-2.0

#

# Unless required by applicable law or agreed to in writing, software

# distributed under the License is distributed on an "AS IS" BASIS,

# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.

# See the License for the specific language governing permissions and

# limitations under the License.

"""Example of Python controller for Mavic patrolling around the house.

Open the robot window to see the camera view.

This demonstrates how to go to specific world coordinates using its GPS, imu and gyroscope.

The drone reaches a given altitude and patrols from waypoint to waypoint."""

from controller import Robot

from math import sqrt

import sys

try:

import numpy as np

except ImportError:

sys.exit("Warning: 'numpy' module not found.")

def clamp(value, value\_min, value\_max):

return min(max(value, value\_min), value\_max)

class Mavic (Robot):

# Constants, empirically found.

K\_VERTICAL\_THRUST = 68.5 # with this thrust, the drone lifts.

# Vertical offset where the robot actually targets to stabilize itself.

K\_VERTICAL\_OFFSET = 0.6

K\_VERTICAL\_P = 3.0 # P constant of the vertical PID.

K\_ROLL\_P = 50.0 # P constant of the roll PID.

K\_PITCH\_P = 30.0 # P constant of the pitch PID.

MAX\_YAW\_DISTURBANCE = 0.4

MAX\_PITCH\_DISTURBANCE = -1

# Precision between the target position and the robot position in meters

target\_precision = 0.5

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

Robot.\_\_init\_\_(self)

self.time\_step = 8

# Get and enable devices.

self.camera = self.getDevice("camera")

self.camera.enable(self.time\_step)

self.imu = self.getDevice("inertial unit")

self.imu.enable(self.time\_step)

self.gps = self.getDevice("gps")

self.gps.enable(self.time\_step)

self.gyro = self.getDevice("gyro")

self.gyro.enable(self.time\_step)

self.front\_left\_motor = self.getDevice("front left propeller")

self.front\_right\_motor = self.getDevice("front right propeller")

self.rear\_left\_motor = self.getDevice("rear left propeller")

self.rear\_right\_motor = self.getDevice("rear right propeller")

self.camera\_pitch\_motor = self.getDevice("camera pitch")

self.camera\_pitch\_motor.setPosition(0.7)

motors = [self.front\_left\_motor, self.front\_right\_motor,

self.rear\_left\_motor, self.rear\_right\_motor]

for motor in motors:

motor.setPosition(float('inf'))

motor.setVelocity(1)

self.current\_pose = 6 \* [0] # X, Y, Z, yaw, pitch, roll

self.target\_position = [0, 0, 0]

self.target\_index = 0

self.target\_altitude = 0

def set\_position(self, pos):

"""

Set the new absolute position of the robot

Parameters:

pos (list): [X,Y,Z,yaw,pitch,roll] current absolute position and angles

"""

self.current\_pose = pos

def move\_to\_target(self, waypoints, verbose\_movement=False, verbose\_target=True):

"""

Move the robot to the given coordinates

Parameters:

waypoints (list): list of X,Y coordinates

verbose\_movement (bool): whether to print remaning angle and distance or not

verbose\_target (bool): whether to print targets or not

Returns:

yaw\_disturbance (float): yaw disturbance (negative value to go on the right)

pitch\_disturbance (float): pitch disturbance (negative value to go forward)

"""

if self.target\_position[0:2] == [0, 0]: # Initialization

self.target\_position[0:2] = waypoints[0]

if verbose\_target:

print("First target: ", self.target\_position[0:2])

# if the robot is at the position with a precision of target\_precision

if all([abs(x1 - x2) < self.target\_precision for (x1, x2) in zip(self.target\_position, self.current\_pose[0:2])]):

self.target\_index += 1

if self.target\_index > len(waypoints) - 1:

self.target\_index=len(waypoints) - 1

self.target\_position[0:2] = waypoints[self.target\_index]

if verbose\_target:

print("Target reached! New target: ",

self.target\_position[0:2])

# This will be in ]-pi;pi]

self.target\_position[2] = np.arctan2(

self.target\_position[1] - self.current\_pose[1], self.target\_position[0] - self.current\_pose[0])

# This is now in ]-2pi;2pi[

angle\_left = self.target\_position[2] - self.current\_pose[5]

# Normalize turn angle to ]-pi;pi]

angle\_left = (angle\_left + 2 \* np.pi) % (2 \* np.pi)

if (angle\_left > np.pi):

angle\_left -= 2 \* np.pi

# Turn the robot to the left or to the right according the value and the sign of angle\_left

yaw\_disturbance = self.MAX\_YAW\_DISTURBANCE \* angle\_left / (2 \* np.pi)

# non proportional and decreasing function

pitch\_disturbance = clamp(

np.log10(abs(angle\_left)), self.MAX\_PITCH\_DISTURBANCE, 0.1)

if verbose\_movement:

distance\_left = np.sqrt(((self.target\_position[0] - self.current\_pose[0]) \*\* 2) + (

(self.target\_position[1] - self.current\_pose[1]) \*\* 2))

print("remaning angle: {:.4f}, remaning distance: {:.4f}".format(

angle\_left, distance\_left))

return yaw\_disturbance, pitch\_disturbance

def run(self):

t1 = self.getTime()

x1=np.array([[-10.0],[10.0],[-10.0],[10.0]])

y1=np.array([[-10.0],[-10.0],[10.0],[10.0]])

roll\_disturbance = 0

pitch\_disturbance = 0

yaw\_disturbance = 0

jkg=1

a=jkg\*np.array([

[1/4, 1/4 , 1/4 , 1/4 ],

[ 1/4, 1/4 , 1/4 , 1/4 ],

[ 1/4, 1/4 , 1/4 , 1/4 ],

[ 1/4, 1/4 , 1/4 , 1/4 ],

])

#this sets the waypoints array which has the coordinates the robot travels to thus setting the path

waypoints=np.empty((10,2))

i=0

rst=0.0

while i in range(10):

x1=np.matmul(a,x1)

waypoints[i][0]=float(x1[0])

y1=np.matmul(a,y1)

waypoints[i][1]=float(y1[0])

if i>=1:

if int(sqrt(waypoints[i-1][0]\*\*2+waypoints[i-1][1]\*\*2))!=0:

rst0=sqrt(waypoints[i][0]\*\*2+waypoints[i][1]\*\*2)/sqrt(waypoints[i-1][0]\*\*2+waypoints[i-1][1]\*\*2)

if rst<rst0:

rst=rst0

#condition for convergence

if int(x1[0])==int(x1[1]) and int(x1[1])==int(x1[2]) and int(x1[2])== int(x1[3]) and (int(y1[0])==int(x1[1])) and int(y1[1])==int(y1[2]) and int(y1[2]) == int(y1[3]) :

print("convergence achieved in ", i+1," iterations")

break

i+=1

# target altitude of the robot in meters

self.target\_altitude = 2

print(waypoints)

print("\nrstep is ",rst,"\n")

while self.step(self.time\_step) != -1:

# Read sensors

roll, pitch, yaw = self.imu.getRollPitchYaw()

x\_pos, y\_pos, altitude = self.gps.getValues()

roll\_acceleration, pitch\_acceleration, \_ = self.gyro.getValues()

self.set\_position([x\_pos, y\_pos, altitude, roll, pitch, yaw])

if altitude > self.target\_altitude-1:

# as soon as it reach the target altitude, compute the disturbances to go to the given waypoints.

if self.getTime() - t1 > 0.1:

yaw\_disturbance, pitch\_disturbance = self.move\_to\_target(

waypoints)

t1 = self.getTime()

roll\_input = self.K\_ROLL\_P \* clamp(roll, -1, 1) + roll\_acceleration + roll\_disturbance

pitch\_input = self.K\_PITCH\_P \* clamp(pitch, -1, 1) + pitch\_acceleration + pitch\_disturbance

yaw\_input = yaw\_disturbance

clamped\_difference\_altitude = clamp(self.target\_altitude - altitude + self.K\_VERTICAL\_OFFSET, -1, 1)

vertical\_input = self.K\_VERTICAL\_P \* pow(clamped\_difference\_altitude, 3.0)

front\_left\_motor\_input = self.K\_VERTICAL\_THRUST + vertical\_input - yaw\_input + pitch\_input - roll\_input

front\_right\_motor\_input = self.K\_VERTICAL\_THRUST + vertical\_input + yaw\_input + pitch\_input + roll\_input

rear\_left\_motor\_input = self.K\_VERTICAL\_THRUST + vertical\_input + yaw\_input - pitch\_input - roll\_input

rear\_right\_motor\_input = self.K\_VERTICAL\_THRUST + vertical\_input - yaw\_input - pitch\_input + roll\_input

self.front\_left\_motor.setVelocity(front\_left\_motor\_input)

self.front\_right\_motor.setVelocity(-front\_right\_motor\_input)

self.rear\_left\_motor.setVelocity(-rear\_left\_motor\_input)

self.rear\_right\_motor.setVelocity(rear\_right\_motor\_input)

# To use this controller, the basicTimeStep should be set to 8 and the defaultDamping

# with a linear and angular damping both of 0.5

robot = Mavic()

robot.run()