Multi-Robot Formation

Robot Control System

Bryson Howell
Kelechi Akwataghibe
Juan Aranda

Final Project

Math Department
University of Tennessee ,Knoxville
SPRING 2020
Monday 4th May, 2020

Contents

0.1 Introduction

Multi-robot formation control is described as the study of control and learning of multiple robots. With the ability to be used in various different ways, formation control is an implementation from nature. Formation control was inspired by fish schooling, bird flocking, and ant swarming where the application onto autonomous robots is important as it aids with unmanned aerial vehicles to search and rescue missions and surveillance. [1] For formation controls of multiple robots, the main objective is to coordinate the robots to work together as one to fulfill a certain task, whether that is observing an area or moving to an objective.The robots establish forces which attract and repel one another in order to avoid collision and maintain certain distances. Originally using flocking, robots could be programmed to specific formations using said fields, but later on, a more dynamic form of programming allowed for robots to adjust their formation during operation.

0.2 Problem Statement and Assumptions

Given a group of robots moving in an obstacle free environment, we aim to investigate and optimize parameters that would predict a higher chance of success . By success, we mean that:

a The robots arrange themselves without collision.

b This coordination happens within a specified time-frame.

c The robots are all able to arrive within an acceptable distance of their target positions

To begin, we assume that there are no obstacles obstructing our robots, and that all robots are of identical make and size. We utilize a vector approach to coordinate the formation of the robots. Assuming the current coordinates of robots R_i are (x_i, y_i) and the coordinates of the goal are (x_t, y_t) , the behavior vector for moving to goal (1) has been determined to be:

$$V_{\text{goal}} = \frac{1}{\sqrt{(x_t - x_i)^2 + (y_t - y_i)^2}} \left(\begin{bmatrix} x_t - x_i \\ y_t - y_i \end{bmatrix} \right) \tag{1}$$

We decided to multiply this vector by a fixed weight, f_1 to gauge how the collision rate if affected by the level of aggressiveness towards the goal. Our principle was to try to achieve our end result with as few parameters as possible.
To handle collisions between robots, another behavior vector was utilized. This one would which would enable a vector to move in an opposite direction to avoid collision with another vector in front of it. The behavior vector for avoidance is given to be:

$$V_{\text{avoid-robot}} = \frac{1}{\sqrt{(x_j - x_i)^2 + (y_j - y_i)^2}} \left(\begin{bmatrix} \pm(x_j - x_i) \\ \pm(y_j - y_i) \end{bmatrix} \right) \tag{2}$$

According to theory, the control parameters of the behavior is:

$$f_2 = \begin{cases} 0 & d_r \in (b_r, +\infty) \\ a_r d_r & d_r \in [2r_0, b_r] \end{cases}$$

where

The Final Velocity of a Robot was then calculate by dotting each behavior vector with the corresponding control parameter:

$$V_{final} = V_{move-to-goal} \cdot f_1 + V_{avoid-robot} \cdot f_2$$

A lot of considerations were made in sticking with the above approaches which we will discuss more extensively later on. Suffice it to say that , the vector component added for collision avoidance drastically improved our results.

0.3 Methods

We implement an abstract model of our solution in Python. Our model operates under the assumptions that all robots are circular in shape with a diameter of 1 unit and that each is capable of omnidirectional movement. While it is not common for ground robots to have this ability, these assumptions accurately model the physical traits of quadrotors. All experiments take place in a rectangular grid with a width and height of 50 units.

At the start of each experiment, a number of robots are randomly distributed throughout the entire area while a set of random target points is generated within a 10x10 area at the center of the experiment space. Robot spawn positions and target points are distributed at least 1 unit apart so as not to cause collisions on start and to ensure that the final formation is achievable. The velocity of each robot is initialized as zero. At each timestep, the velocity of each robot is calculated as the sum of vectors (1) and (2). Additionally, we define an inertia value of I = 0.1 to simulate momentum, which adds a portion of the previous velocity vector to the new one. We also define a maximum velocity value vmax, which is kept constant at 1.0. The velocity of each robot is scaled to this value once it is calculated. This threshold is to control the amount of chaos that could quickly result in such a randomized environment. The position update algorithm is defined below as equation (3):

$$
\begin{aligned}
&\text{Until max timesteps is reached:} \\
&\quad \text{For each robot i:} \\
&\qquad v'[i] = v[i] * I + f_1 * (1) + f_2 * (2) \\
&\qquad pos'[i] = pos[i] + v'[i] \\
&\qquad \text{If } v'[i] > vmax: \\
&\qquad\quad \text{Scale velocity}
\end{aligned}
\tag{3}
$$

We define a collision as occurring when the distance between two robots is less than 1. When a collision occurs, no movement penalty is incurred. This is a limitation of our model, as it is possible for robots to end in the same physical location. However, as long as collisions are not catastrophic (such as when controlling aerial vehicles) robots bumping into each other should not severely impact the performance of the controller.

When two robots are moving in the same direction and are within a defined critical distance of each other , we utilize the avoidance component of the vector to cause one of the vectors to move in a direction opposite to its current velocity direction. This is done by negating the sign of both components of the collision avoidance vector

```
For each robot i:
    Search for robots j moving in the same direction as i
        If distance(i,j) ¡ critical distance
            avoiddx = avoiddx * (-1)
            avoiddy = avoiddy * (-1)
```
(4)

This is then used to update the final vector of the given robot.

0.4 Solution/Results

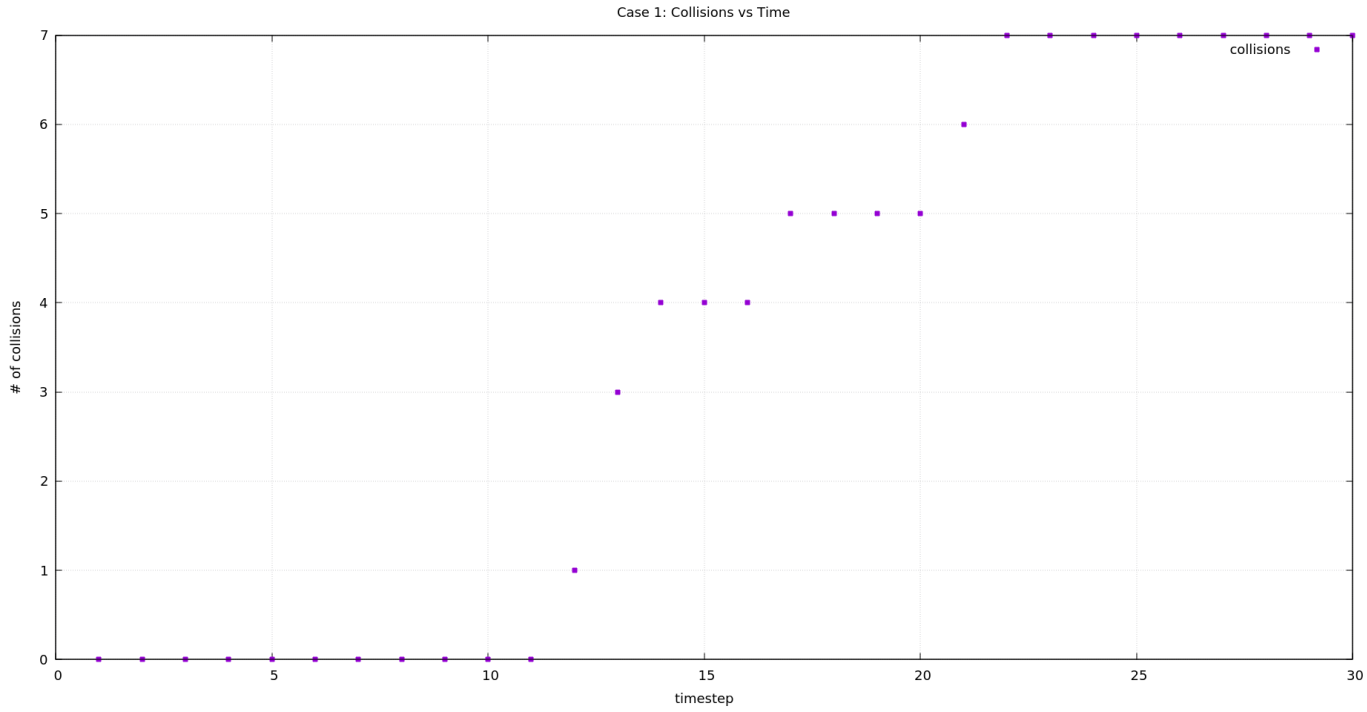
We ran simulations , altering the number of robots and control parameters in each case. The goal was to understand how effective the parameters were in minimizing average error and collisions.

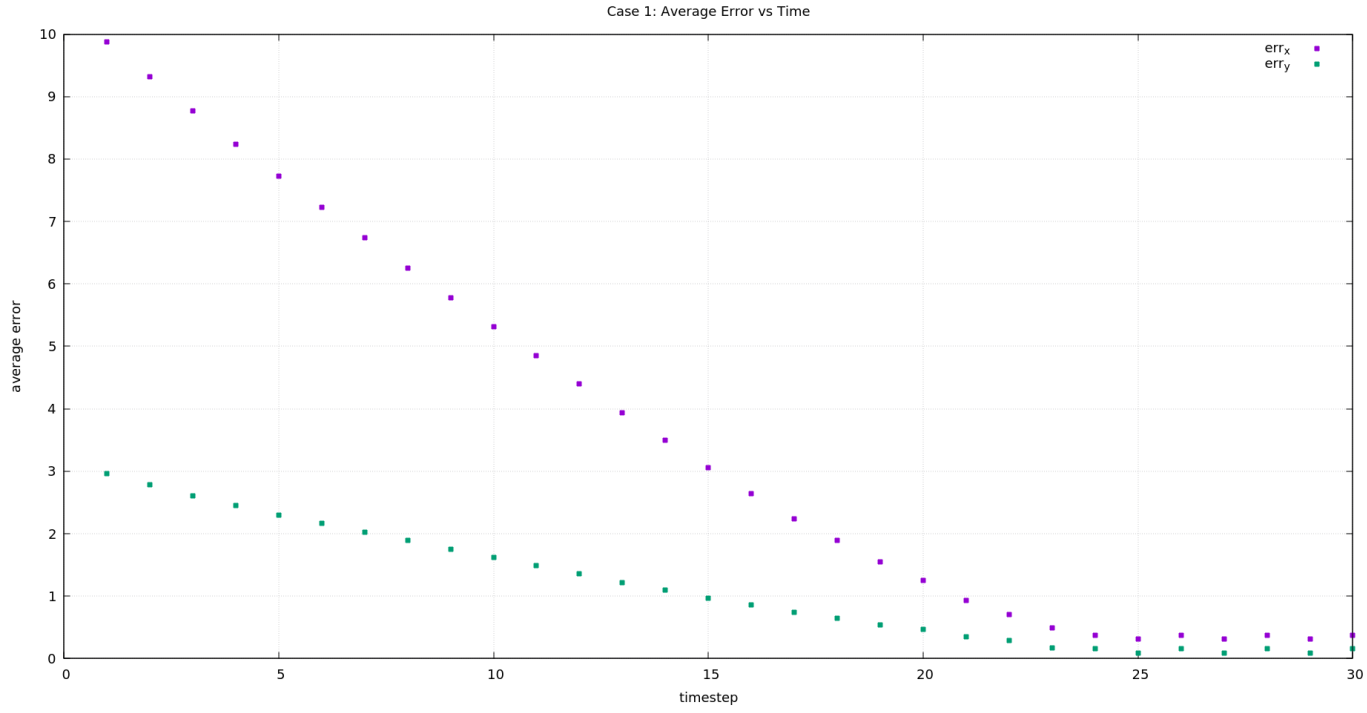
- Two graphs were plotted in each case:

- Number of Collisions vs Time: This plots the cumulative number of collisions per time step.
- Average error vs Time: This plots the average of the distances apart each robot has towards their target goal.

For each case, we simulated the initial positions of the robots by assigning random position vectors. To minimize the errors that come with "randomization", several of the cases were run multiple times and the most typical of the trends were selected. It is also important to note that many more parameters apart from the ones displayed were tested but we would only mention them briefly due to space constraints. We selected a handful of very instructive cases that seemed to predict a trend.

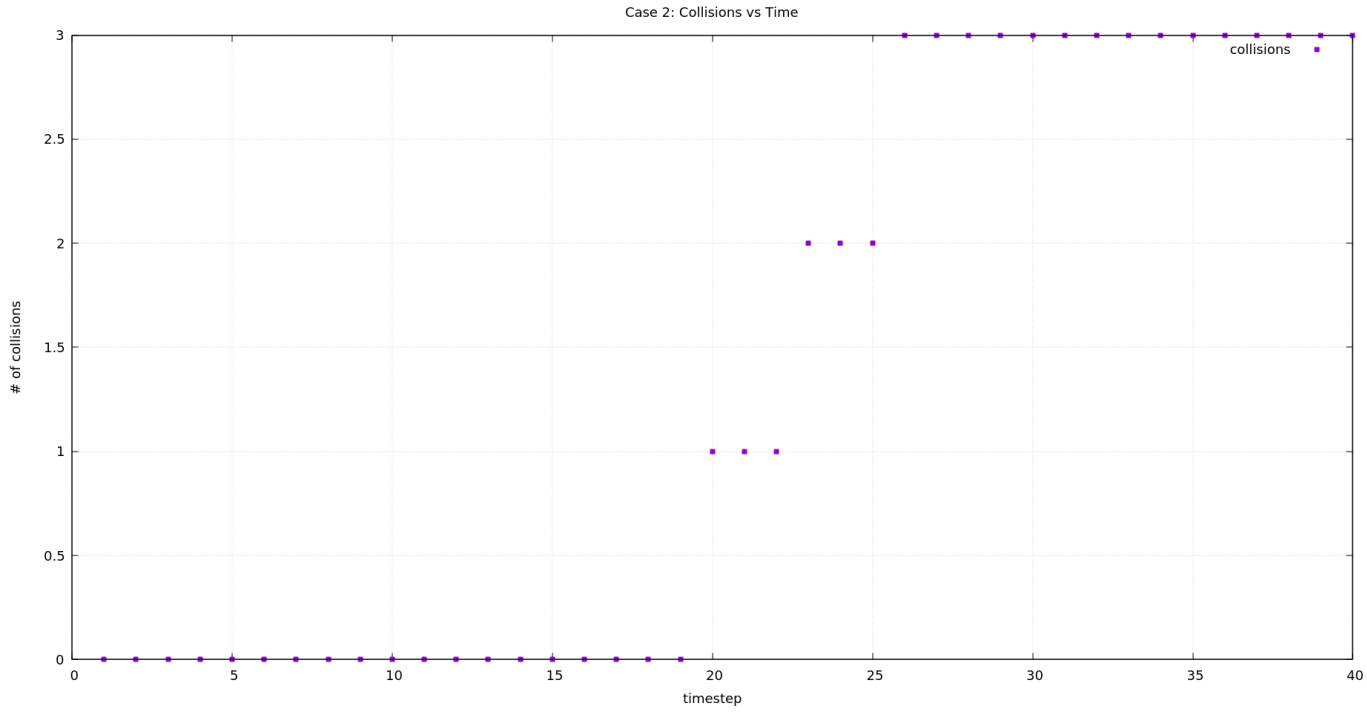
- **Case 1: N = 8, $f_1 = 1.0$, $f_2 = 0$**

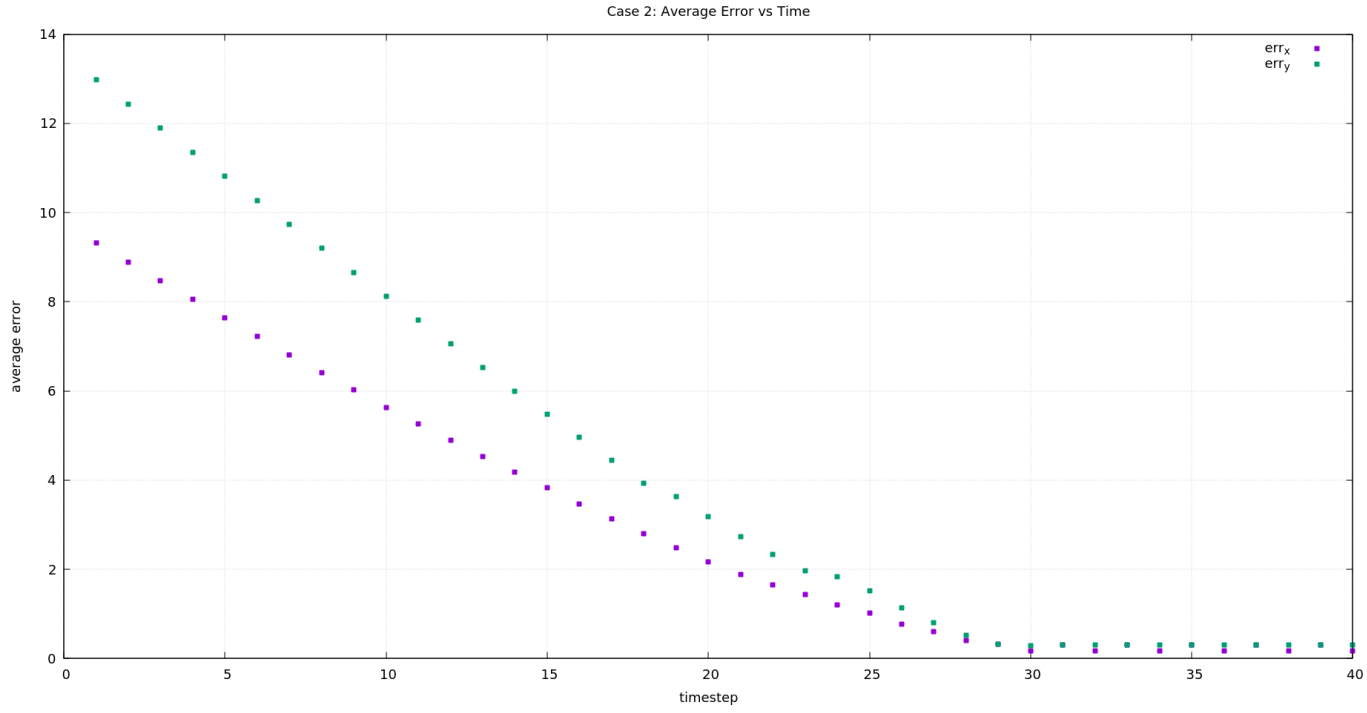




Without the vector accounting for the "avoid robot" behavior, with 6 robots, the system recorded about 7 collisions when the robots reached their destination. All the robots arrived at their target location within 30 timesteps.

- **Case 2: N = 8, $f_1 = 1.0$, $f_2 = 1.0$**

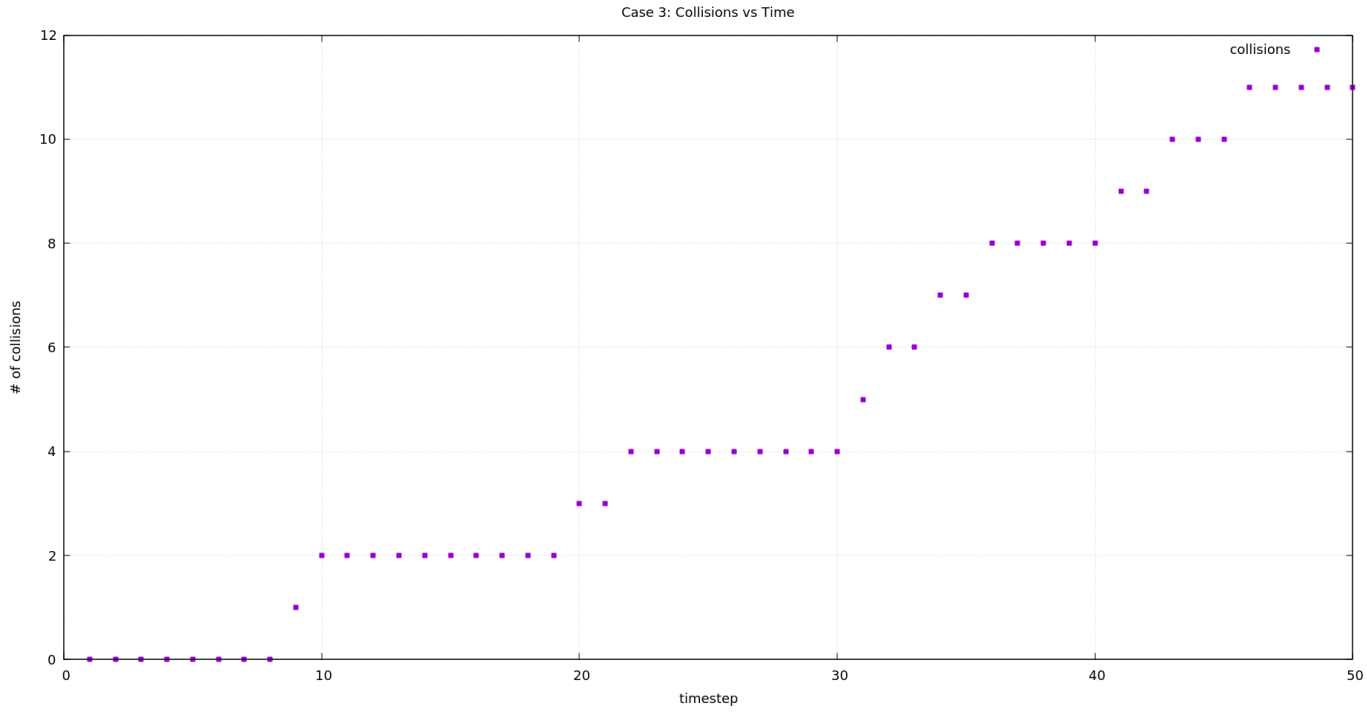


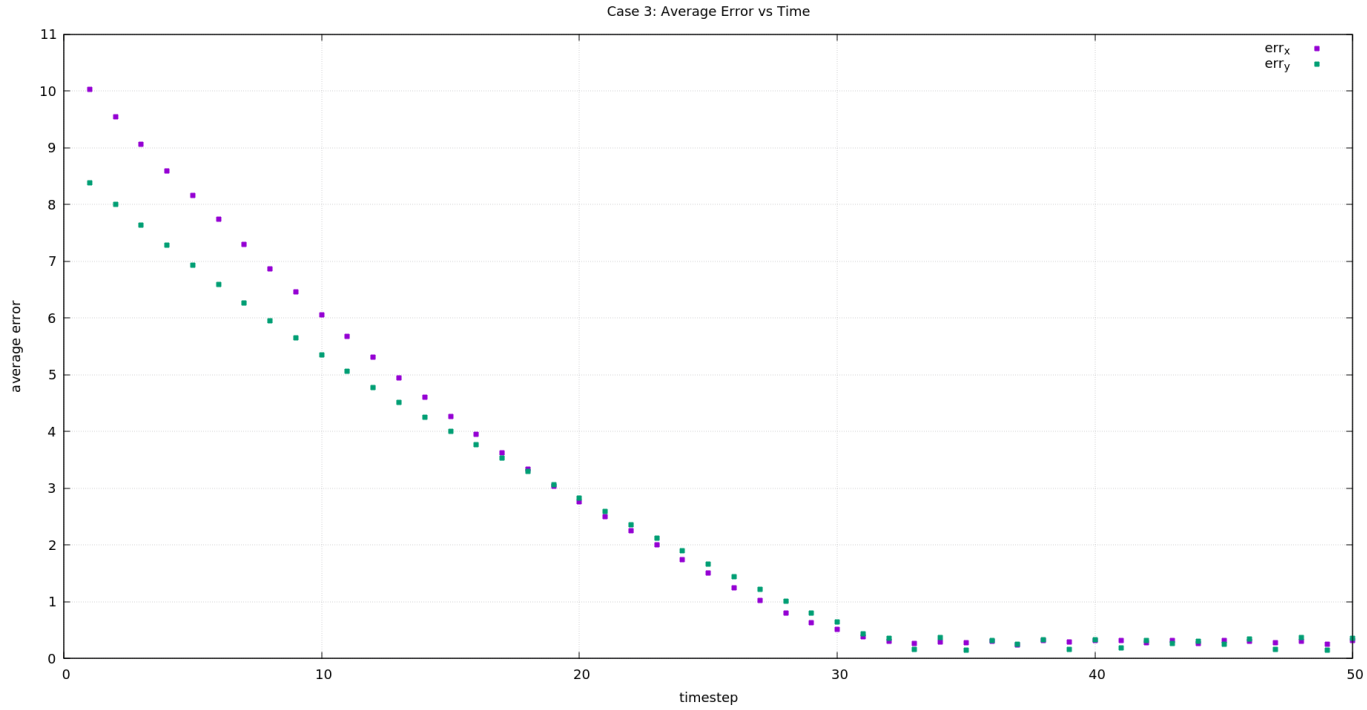


In a subsequent iteration with the same number of robots, we utilize an f_2 parameter value of 1.0 and we see that this halved the number of collisions. However, the robots do not all reach their target positions until after 50 steps. We suspect the "more" redirections that were taken to avoid collisions would have increased the total distance away from the target points;

hence, increasing the time taken to enter the formation

- **Case 3: N = 8, f_1 = 2.0, f_2 = 1.0**

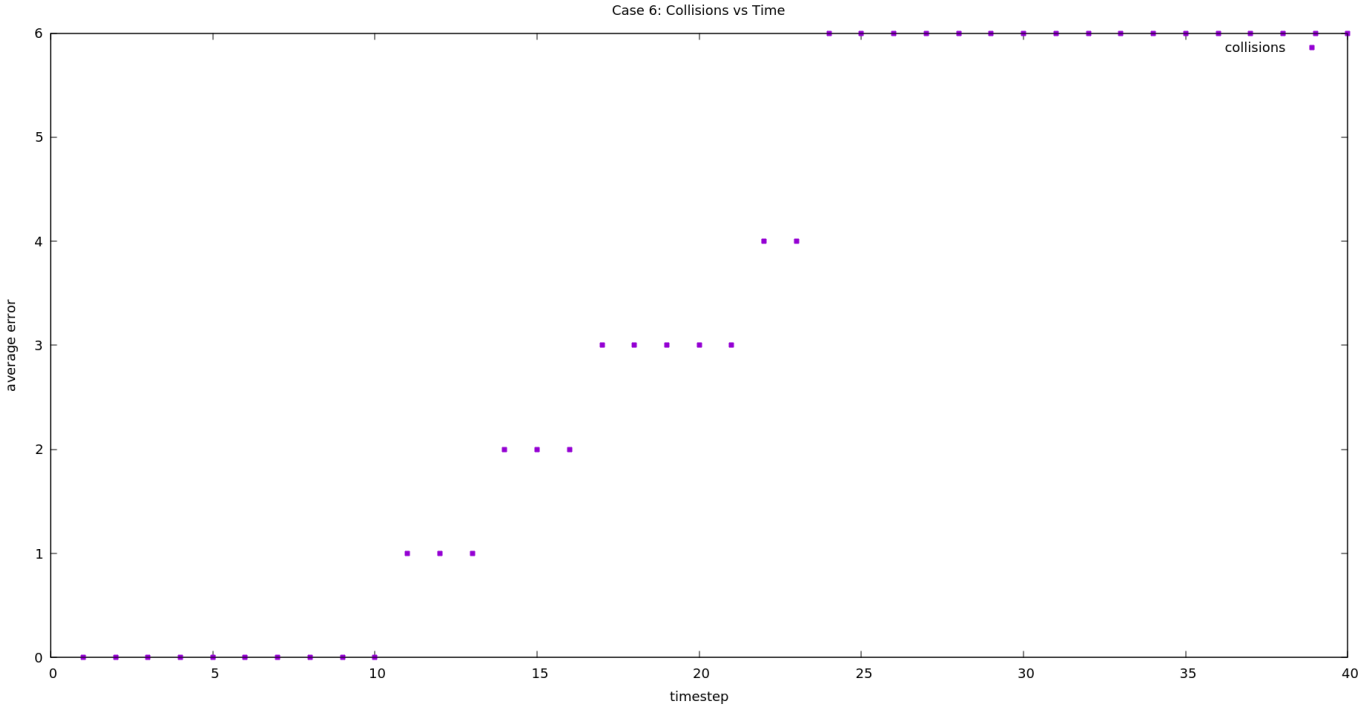


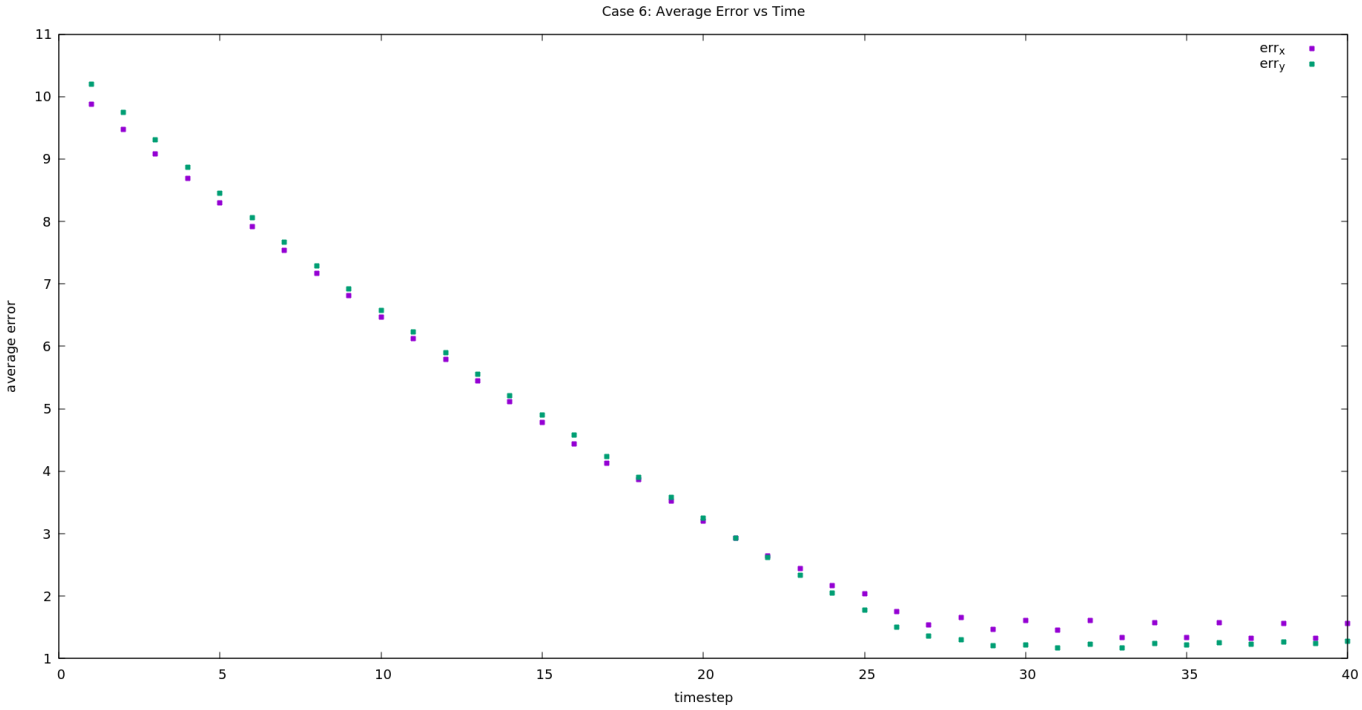


Keeping the f_2 parameter constant and increasing the driving parameter of the "move-to-goal" behavior vector drastically increases the number of collisions. This could be due to the

fact that as the robots tend more aggressively to the goal, there is a higher propensity for them to bump into each other

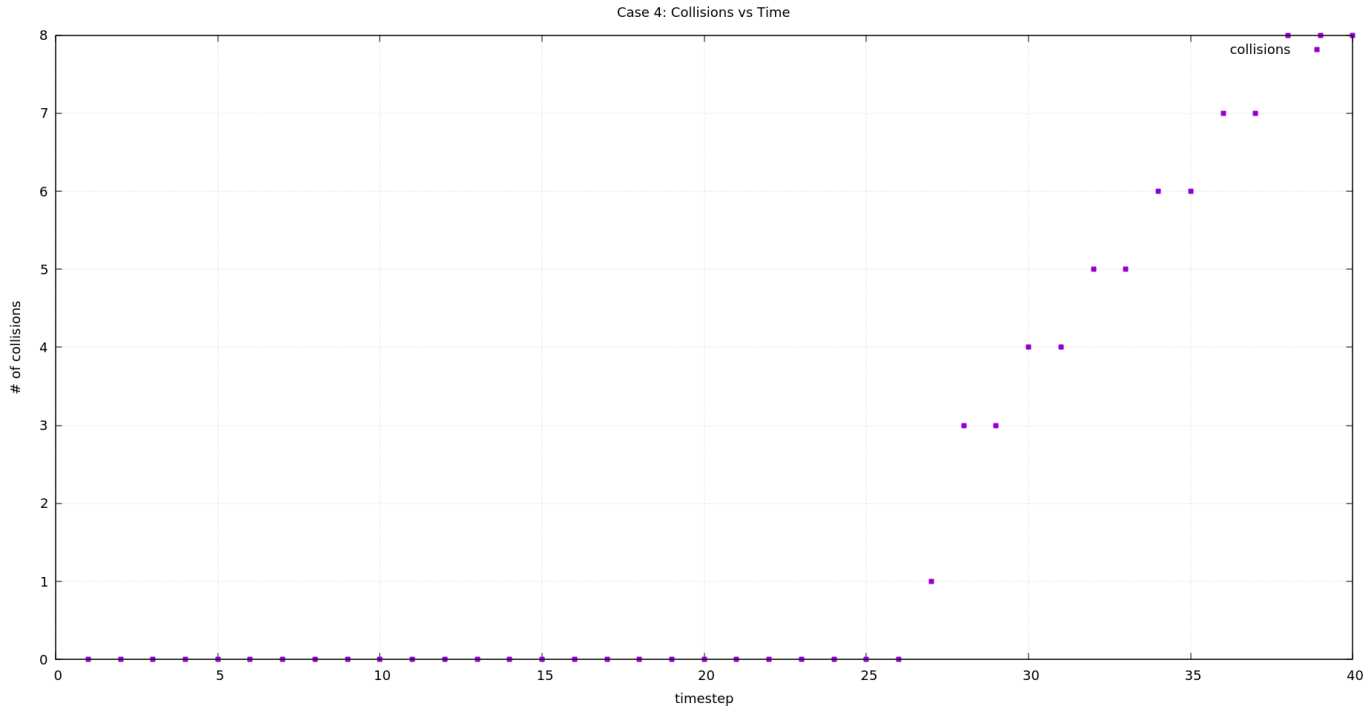
- **Case 4: N = 8, $f_1 = 1.0$, $f_2 = 2.0$**

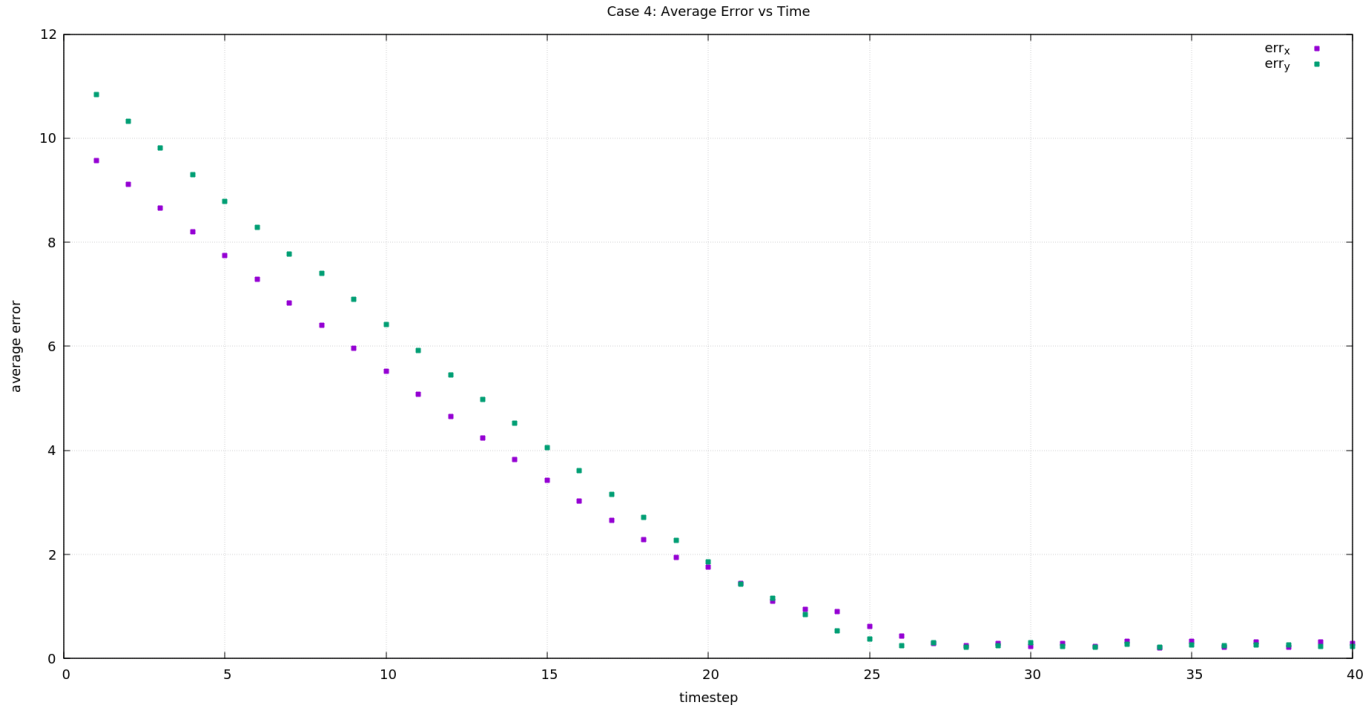




In this particular case, we kept the number of robots fixed and tried to increase the parameter weight that controls the avoid collision behavior. We expected there to be a further decrease in the number of collisions but surprisingly the number of collisions doubled from about 3 to 6. Subsequent increase in this factor gave rise to even more collision. It is not clear why, but we speculate that this increase largely has to do with the higher probability of colliding that comes about when the robots change direction more "drastically"

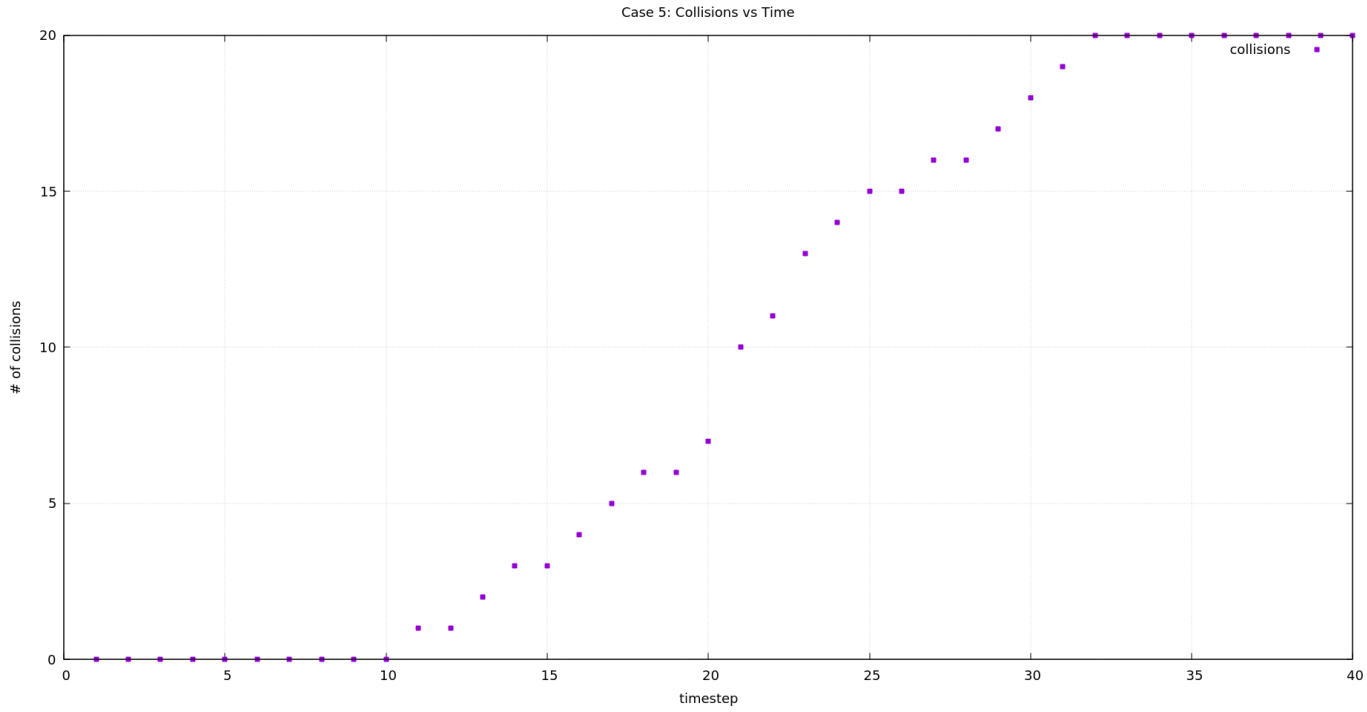
- **Case 5: N = 10, $f_1 = 1.0$, $f_2 = 2.0$**

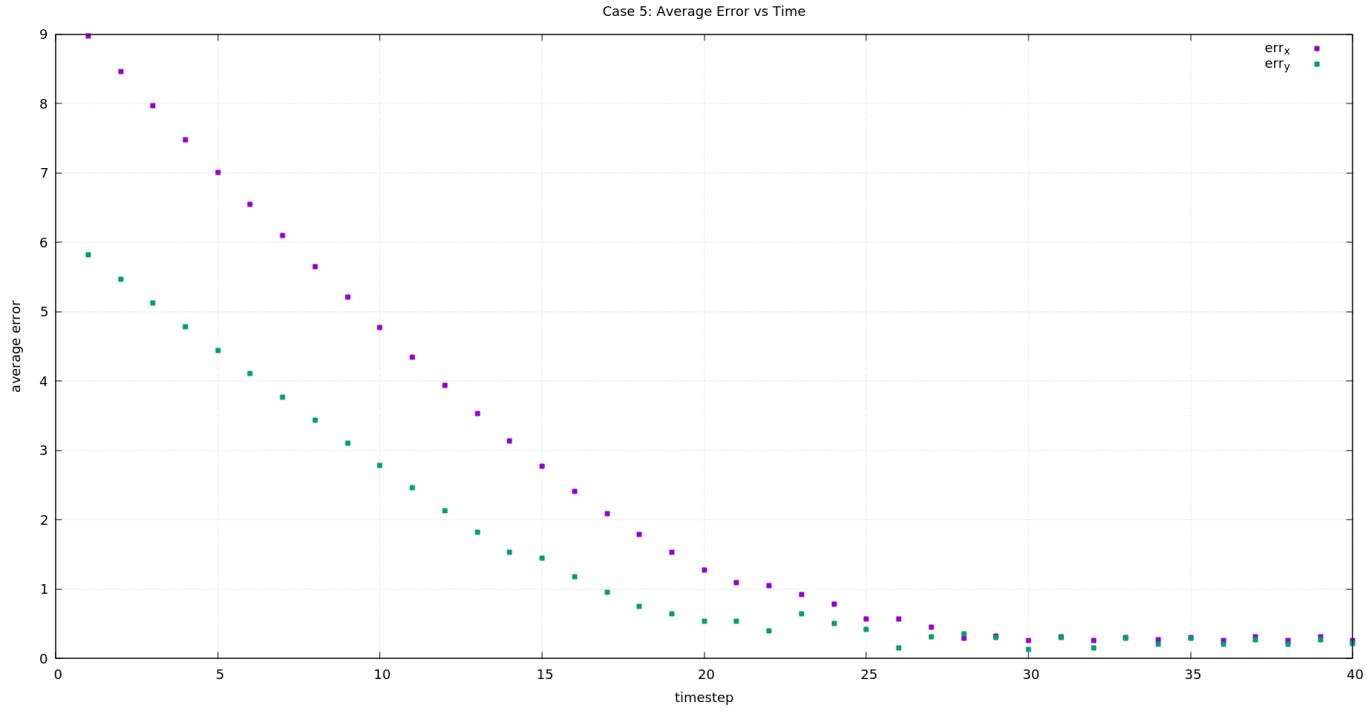




An increase in the number of robots , while keeping the parameters fixed, resulted in the greatest number of collisions recorded by the robots. Given the fixed nature of our ambient space, it is straightforward to see that as the population density of robots grows, the distance between individual robots decrease and thus, the likelihood of collision increases .

- **Case 6: N = 12, $f_1 = 1.0$, $f_2 = 1.0$**





This case further illustrates that of case 5. We have an almost 100% increase in the number of collisions by increasing the number of robots by20%

0.5 Discussion

We started out trying to build a controller that will coordinate the group into a given geometry using first order dynamics. Due to environmental constraints, and not having access to the lab equipment for simulation as planned, we decided to investigate a smaller portion of the problem and try to understand how the different parameters affected our success (i.e. gave minimal error and collision rate).

Trying to make the robots avoid colliding with one another was the most difficult part of this enterprise. We considered making the one robot stop moving if two of them are within a certain threshold of distance from each other. This seemed to temporarily alleviate the situation but it did not seem to be the most practical, especially in real life scenarios. It was important to us to try to keep constant motion of the robots and so we decided to allow the robots remain in motion while handling potential collisions.

The first key observation made, which was expected, was that an increase in the number of robots led to more collisions. This makes sense as the population density of the robots decreases the gross "distance apart" of the robots and subsequently, increases the chance of potential collisions.

It was observed that collisions dramatically increased when more than 10 robots were set in the area. This seems to predict a critical value of 10 robots given our area constraints.

The goal factor (f_1) was directly correlated with an increase in collisions. It seemed that the less aggressive the robots moved towards their targets, the less the likelihood of collisions.
Unexpectedly, we found the avoidance factor(f_2) to be inversely correlated with the number of collisions for a given range of f_2, after which, it seemed to vary directly as the number of collisions . There is no clear reason as to why this is the case, but we can intuitively guess that the parameter f_2 might depend on the physical constraints used in the code.

Currently, our results are still in very early stages and we have not been able to rigorously verify some of our trends. We hope to try out an actual simulation in real-life physics to get more pertinent data that will furnish better conclusions. While the approach we exemplified appears to be simple, it will be also interesting to investigate a more robust Differential Equation approach to model the situation.

0.6 Appendix

```
#Bryson Howell
#M475 - Team Project
#Implements a formation controller for a group of robots.

import numpy as np

xtargets = np.zeros(0,np.float64()) #Goal x position for each robot
ytargets = np.zeros(0,np.float64()) #Goal y positoin for each robot
size = np.int32()

def main():

    #global epochs, inertia, size
    #global xPos, yPos, xtargets, ytargets, vels
    global xtargets,ytargets,size

    #Get input from user
    epochs = np.int32(input("#Enter number of epochs: "))
    size = np.int32(input("#Enter number of robots: "))
    f1 = np.float64(input("#Enter goal control parameter: "))
    f2 = np.float64(input("#Enter avoidance control parameter: "))

    #Initalize arrays
    xPos = np.random.randint(-25,26,size,np.int32()).astype(np.float64())
    yPos = np.random.randint(-25,26,size,np.int32()).astype(np.float64())
    vels = np.zeros((size,2),np.float64())
    #Create target positions
    createTargets()

    #Set constants and values for loop
    inertia = 0.1
    thres = 0.25             #Accuracy threshold
    vmax = 1.0               #Maximum velocity
    br = 2.0                 #Crit distance b/w 2 robots
    dr = np.float64(0.0)     #Distance b/w robot and one it's going to hit
    q = np.float64(0.0)
    err_x = np.float64(0.0)  #Remaining distance between robot and goal
    err_y = np.float64(0.0)
    trajx = np.float64(0.0)  #Trajectory of robot towards the goal
    trajy = np.float64(0.0)
    goalx = np.float64(0.0)  #Goal component of velocity
    goaly = np.float64(0.0)
    avoidx = np.float64(0.0) #Avoidance component of velocity
    avoidy = np.float64(0.0)
    t = 0
    collisions = 0

    #Run until max epochs is reached
    while(t < epochs):
        converge_count = 0
        err_x = 0.0
        err_y = 0.0
        #Update position of each robot
        for i in range(0,size):
```

```
    #Goal component of velocity
    goalx =
        (1.0/dist(xPos[i],xtargets[i],yPos[i],ytargets[i]))*(xtargets[i]-xPos[i])
    goaly =
        (1.0/dist(xPos[i],xtargets[i],yPos[i],ytargets[i]))*(ytargets[i]-yPos[i])
    #Avoidance component of velocity
    trajx, trajy = unit(goalx,goaly,1)
    avoidx = 0.0
    avoidy = 0.0
    bx = xPos[i]
    by = yPos[i]
    for d in xrange(0,np.int32(br/0.5)):
        #search for nearest robot in goal Trajectory
        bx += trajx*0.5
        by += trajy*0.5
        for j in range(0,size):
            #Robot is along trajectory
            if(i != j and dist(bx,xPos[j],by,yPos[j]) < 1.0):
                #Robot is within critical distance of i
                dr = dist(xPos[i],xPos[j],yPos[i],yPos[j])
                if(dr < br):
                    avoidx = (1.0/dr)*np.abs(xPos[i]-xPos[j])
                    if(vels[j][0] > 0):
                        avoidx *= -1
                    avoidy = (1.0/dr)*np.abs(yPos[i]-yPos[j])
                    if(vels[j][1] > 0):
                        avoidy *= -1
            if(avoidx != 0 or avoidy != 0):
                break
        if(avoidx != 0 or avoidy != 0):
            break
    #Find velocities
    vels[i][0] = inertia*vels[i][0] + f1*goalx + f2*dr*avoidx
    vels[i][1] = inertia*vels[i][1] + f1*goaly + f2*dr*avoidy
    #Scale velocities (if it exceeds max)
    if(np.power(vels[i][0],2.0)+np.power(vels[i][1],2.0) > vmax*vmax):
        vels[i][0], vels[i][1] = unit(vels[i][0],vels[i][1],vmax)
    #Update position
    xPos[i] = xPos[i] + vels[i][0]
    yPos[i] = yPos[i] + vels[i][1]
    #Sum for avg error
    err_x += np.power(np.float64(xPos[i]-xtargets[i]),2.0)
    err_y += np.power(np.float64(yPos[i]-ytargets[i]),2.0)
    #Count robots that are close to their target
    if(dist(xPos[i],xtargets[i],yPos[i],ytargets[i]) < thres):
        converge_count += 1

#Check for collissions
for i in range(0,size):
    for j in range(i+1,size):
        if(dist(xPos[i],xPos[j],yPos[i],yPos[j]) < 1.0):
            collisions += 1

#Find average error in position
err_x = np.sqrt((1.0/(2.0*size))*err_x)
err_y = np.sqrt((1.0/(2.0*size))*err_y)
#print for graphs
```

```
        print("%d %d %d %.14f %.14f" % (t,converge_count,collisions,err_x,err_y))

        #Return when all robots are within range of their targets
        if(converge_count == size):
            break
        t = t+1

    return

def dist(x1,x2,y1,y2):

    return np.sqrt(np.power(x1-x2,2.0)+np.power(y1-y2,2.0))

#Creates a unit vector for each component, and scales the result
def unit(vx,vy,scale):

    ux = (vx/np.sqrt(np.power(vx,2.0)+np.power(vy,2.0)))*scale
    uy = (vy/np.sqrt(np.power(vx,2.0)+np.power(vy,2.0)))*scale

    return ux,uy

#Fills xtargets and ytargets with random points in a 10x10 rectangle that are at least
    1m apart
def createTargets():

    global size, xtargets, ytargets

    points = 0
    while(points < size):
        #Create random point within 10x10 area
        px = np.random.uniform(-5.0,5.0)
        py = np.random.uniform(-5.0,5.0)
        count = 0
        #Check to see if point is far enough away from others
        for i in range(0,points):
            if(dist(px,xtargets[i],py,ytargets[i]) > 1.0):
                count += 1
            else:
                break
        #Add point if it's good
        if(points == count):
            xtargets = np.append(xtargets,px)
            ytargets = np.append(ytargets,py)
            points += 1
    return

if __name__ == '__main__':
    main()
```