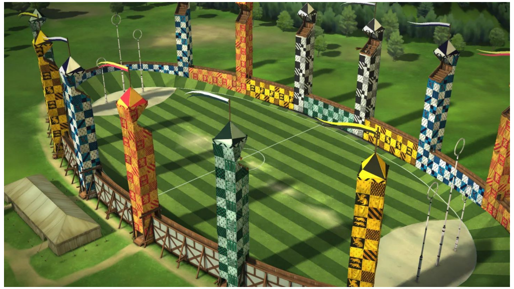
University of Massachusetts
UMASS Lowell
Learning with Purpose

Fundamentals of Robotics
EECE 5560

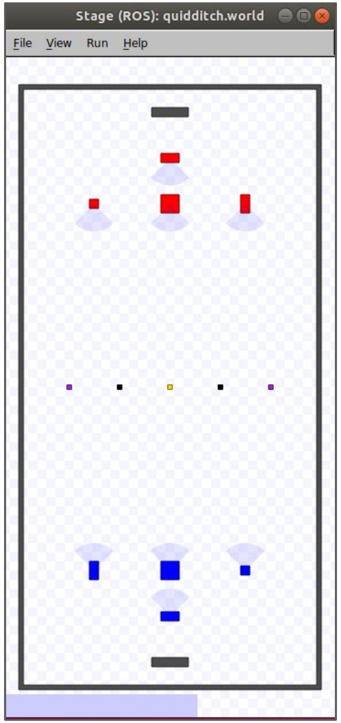
Final Project

Quidditch 2D Simulation Using Stage



GROUP:

1. *Ariel Pena-Martinez*
2. *Christian Dumas*
3. *Edmundo Peralta*
4. *Meg Dillard*
5. *Pablo Ruiz*
6. *Reema Saleem*

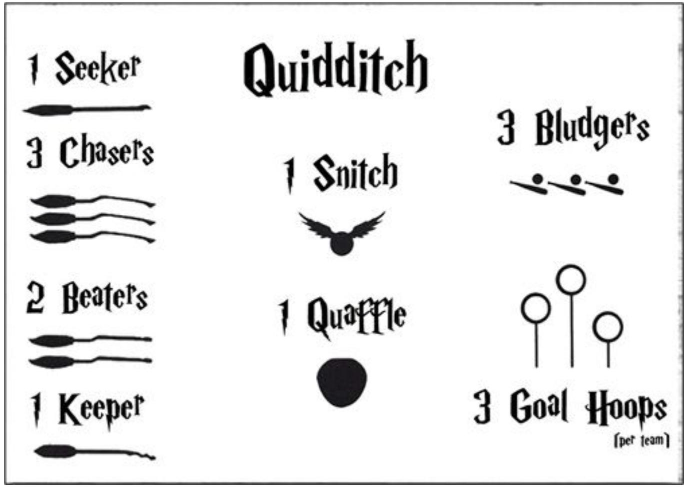


Project Overview

- Utilized ROS melodic and Stage ROS, a 2.5D simulator, to simulate a Quidditch match between Gryffindor (red) and Hufflepuff (blue) robots
- Developed 2 nodes per each position (Chaser, Beater, Keeper and Seeker) to represent each team
- 2 Quaffles and 2 Bludgers nodes were created to prevent issues of robots attempting to find the same ball
- 1 Snitch was made to be faster with both Seekers attempting to catch it and win the game
- World file was used to create the game space and agents/objects within that defined gamespace
- Gamerunner node was created to manage the gameplay and reset or start the game over when needed

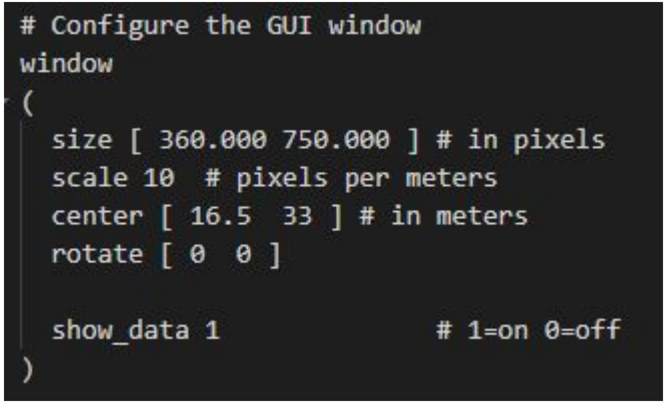
Our Modified Quidditch Rules

- 2 Teams: Red and Blue
    - 4 players on each team:
        - Chaser
        - Beater
        - Keeper
        - Seeker
- Balls:
    - 2 Quaffles, one for each Chaser
    - 2 Bludgers, one for each Beater
    - 1 Snitch



- Beaters capture their Bludger, then try to collide with opposing Chasers
- Chasers capture their Quaffle, then try to reach opposing Goals
- Keepers block opposing Chasers when they get within scoring range
- Seekers try to capture the Snitch, which tries to avoid them
- Game ends with capture of the Snitch

World File

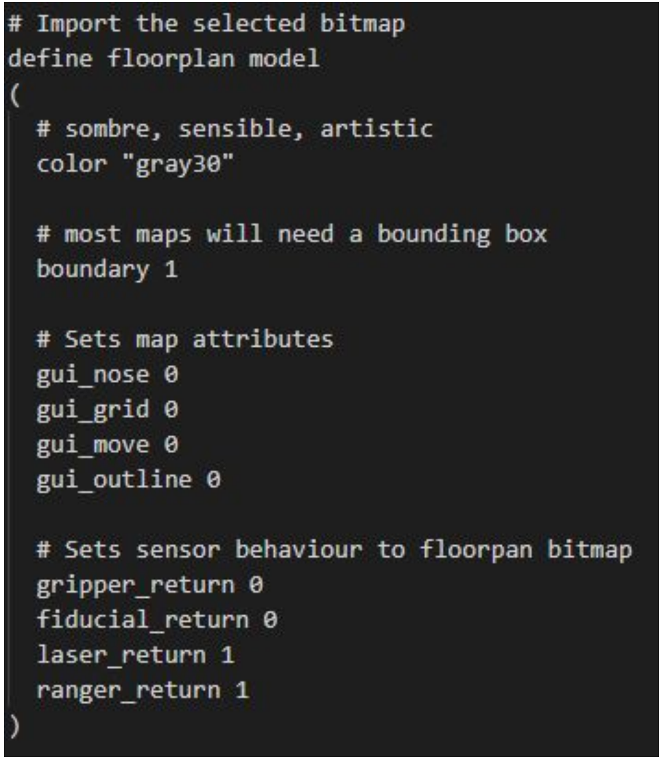


```
# Configure the GUI window
window
(
  size [ 360.000 750.000 ] # in pixels
  scale 10  # pixels per meters
  center [ 16.5  33 ] # in meters
  rotate [ 0  0 ]

  show_data 1             # 1=on 0=off
)
```

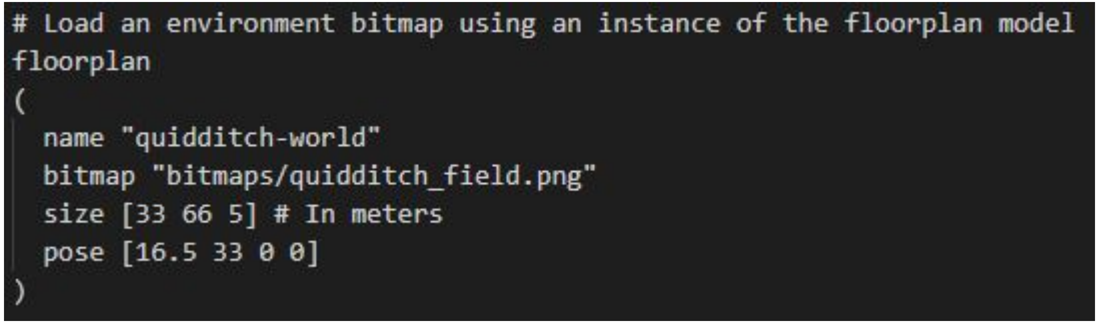
- The world file is at the heart of stage.
- It defines all the environmental features from the window size to the agents and objects involved in the simulation.
- The window is defined in pixels.
- It uses a specific notation which is similar to an object oriented programming language.
- Rosrun stage_ros stageros <FILE_NAME>.world

World File

- The floorplan model is similar to a class.
- The floorplan is an instance of the floorplan model.
- Different options can be set during the model declaration.
- The map features are imported from a bitmap file which can be a png, jpeg.
- The floorplan pose determines where the center of the map is in absolute window coordinates.



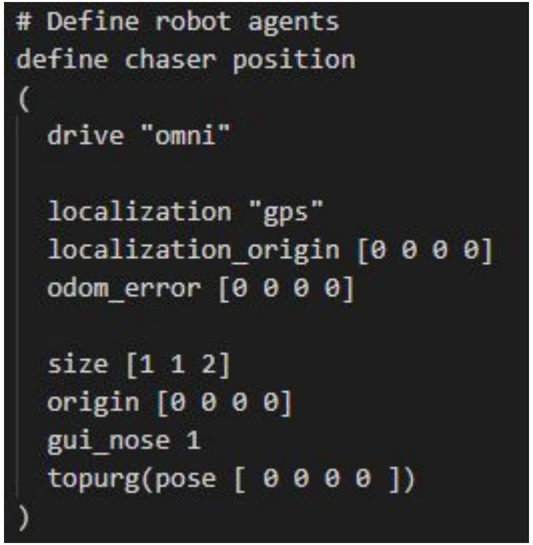
```
# Import the selected bitmap
define floorplan model
(
  # sombre, sensible, artistic
  color "gray30"

  # most maps will need a bounding box
  boundary 1

  # Sets map attributes
  gui_nose 0
  gui_grid 0
  gui_move 0
  gui_outline 0

  # Sets sensor behaviour to floorpan bitmap
  gripper_return 0
  fiducial_return 0
  laser_return 1
  ranger_return 1
)
```



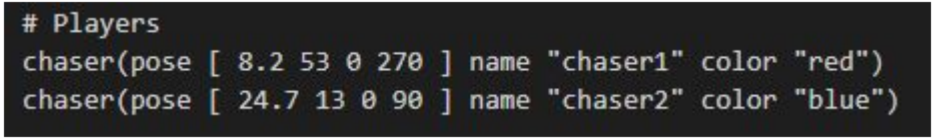
```
# Load an environment bitmap using an instance of the floorplan model
floorplan
(
  name "quidditch-world"
  bitmap "bitmaps/quidditch_field.png"
  size [33 66 5] # In meters
  pose [16.5 33 0 0]
)
```

World File

- Other “classes” can be added in stage including: position (agent), model (objects), sensors (ranger, laser, camera, gripper...).
- The drive option determines the way in which the robot moves. The options are differential “diff”, omnidirectional “omni”, and “car”.
- The localization option can be set to “gps” or “odom” and an artificial source of error can be induced using the odom_error option.
- The topics will be labeled with the order in which the robots were added, ie. /robot_X/odom.
- Two topics which are relevant for each agent, /odom and /cmd_velocity



```
# Define robot agents
define chaser position
(
  drive "omni"

  localization "gps"
  localization_origin [0 0 0 0]
  odom_error [0 0 0 0]

  size [1 1 2]
  origin [0 0 0 0]
  gui_nose 1
  topurg(pose [ 0 0 0 0 ])
)
```



```
# Players
chaser(pose [ 8.2 53 0 270 ] name "chaser1" color "red")
chaser(pose [ 24.7 13 0 90 ] name "chaser2" color "blue")
```

Gamerunner Node

- Whenever a Chaser reaches the same position as the opposing team’s Goal, that Team will score 10 points. This will be recorded and announced by the Quidditch Game node, then the Chasers, Beaters, Keepers, Quaffle are reset.

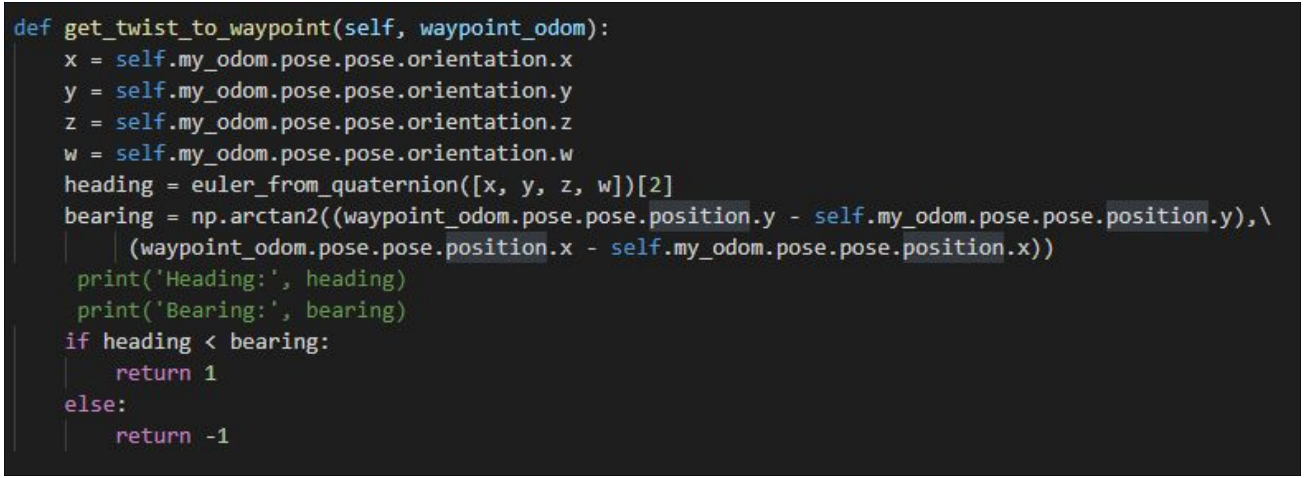
- If the Keeper blocks an incoming Chaser, the Game node announces a block and resets positions of that particular Chaser and Keeper.

- When a Beater collides with the opposing team’s Chaser, the Game node announces a collision. That Chaser’s team loses 10 points and the positions of that particular Chaser and Beater are reset.

- When a Seeker catches the Snitch, that Seeker’s Team scores 50 points, this is announced by the Game node and the match is over. If both Seekers get to the Snitch at the same time, both teams get 50 points and the match is decided by the overall score.

- The Team with the highest score wins the match. The score is maintained by the Game node.

Get_twist_to_waypoint()



```
def get_twist_to_waypoint(self, waypoint_odom):
    x = self.my_odom.pose.pose.orientation.x
    y = self.my_odom.pose.pose.orientation.y
    z = self.my_odom.pose.pose.orientation.z
    w = self.my_odom.pose.pose.orientation.w
    heading = euler_from_quaternion([x, y, z, w])[2]
    bearing = np.arctan2((waypoint_odom.pose.pose.position.y - self.my_odom.pose.pose.position.y),\
         (waypoint_odom.pose.pose.position.x - self.my_odom.pose.pose.position.x))
     print('Heading:', heading)
     print('Bearing:', bearing)
    if heading < bearing:
        return 1
    else:
        return -1
```

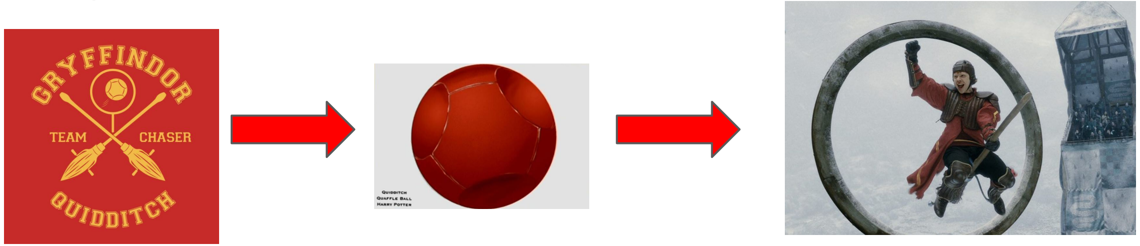
Chaser Node

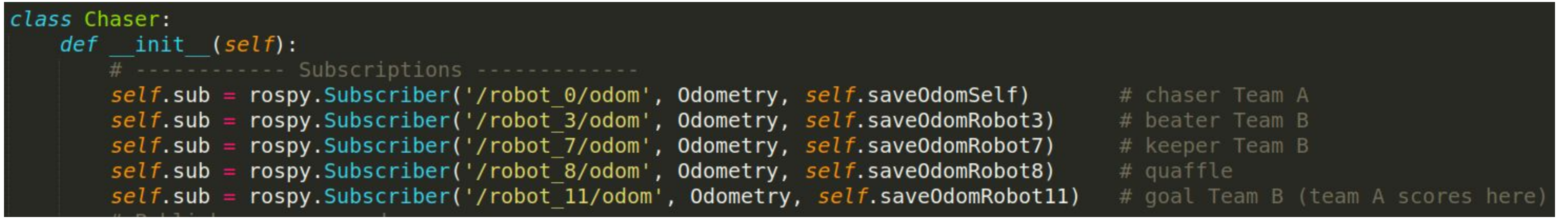
Chaser/Quaffle

The Chaser robot needs to first find (get within a certain distance of) the Quaffle. Then the Chaser will get the Quaffle to the opposite team’s Goal. Therefore, the Chaser will be trying to get to the same position as the Goal, while trying to avoid the Keeper and Beater of the opposite team.

Will also try to recover from any collisions during the run.

There is 1 chaser per team.

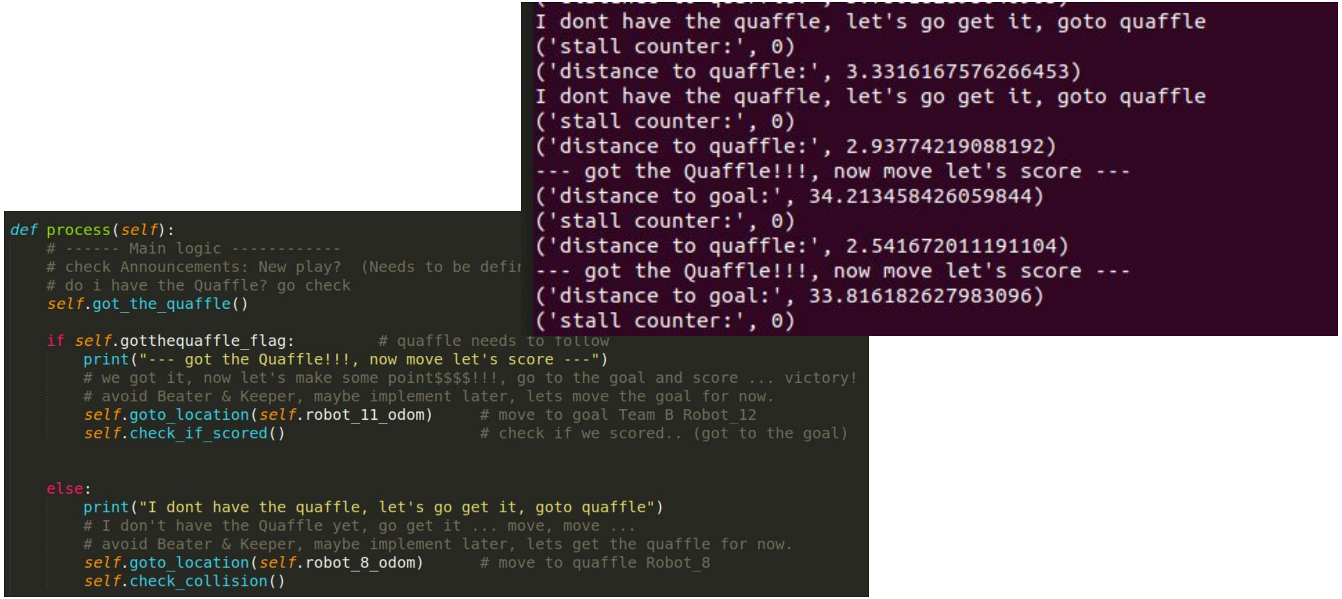




```
class Chaser:
    def __init__(self):
        # ------------ Subscriptions -------------
        self.sub = rospy.Subscriber('/robot_0/odom', Odometry, self.saveOdomSelf)       # chaser Team A
        self.sub = rospy.Subscriber('/robot_3/odom', Odometry, self.saveOdomRobot3)     # beater Team B
        self.sub = rospy.Subscriber('/robot_7/odom', Odometry, self.saveOdomRobot7)     # keeper Team B
        self.sub = rospy.Subscriber('/robot_8/odom', Odometry, self.saveOdomRobot8)     # quaffle
        self.sub = rospy.Subscriber('/robot_11/odom', Odometry, self.saveOdomRobot11)   # goal Team B (team A scores here)
```

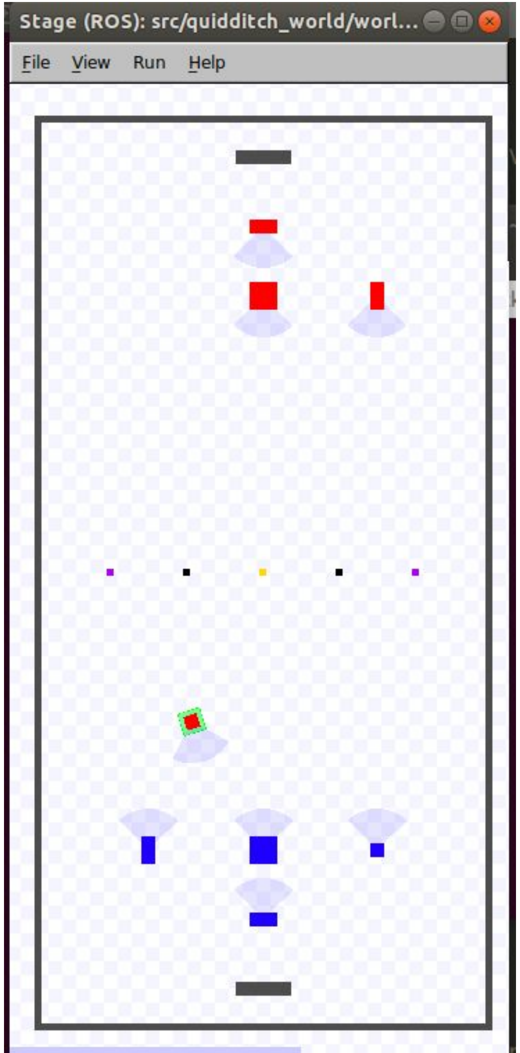
Chaser status announcements:

- Distance to the Quaffle
- Distance to the Opposite Goal
- Quaffle posesion status = Got the quaffle!!!
- Collision and recovery attempts
- Moving to target status: moving to the Quaffle, Moving to the Goal
- Goal !!!!! when it makes it to the opposite team Goal location



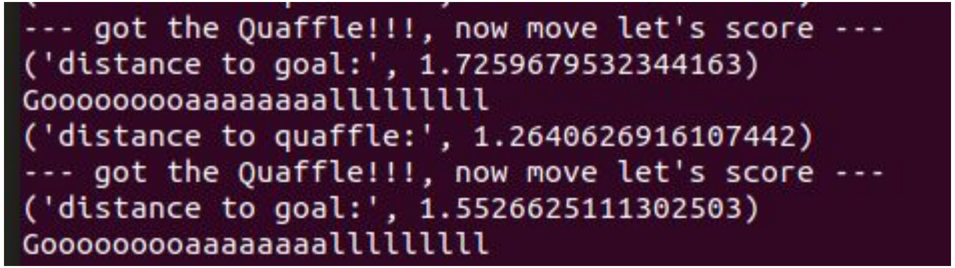
```
I dont have the quaffle, let's go get it, goto quaffle
('stall counter:', 0)
('distance to quaffle:', 3.3316167576266453)
I dont have the quaffle, let's go get it, goto quaffle
('stall counter:', 0)
('distance to quaffle:', 2.93774219088192)
--- got the Quaffle!!!, now move let's score ---
('distance to goal:', 34.213458426059844)
('stall counter:', 0)
('distance to quaffle:', 2.541672011191104)
--- got the Quaffle!!!, now move let's score ---
('distance to goal:', 33.816182627983096)
('stall counter:', 0)
```

```
def process(self):
    # ------ Main logic ------------
    # check Announcements: New play?  (Needs to be defir
    # do i have the Quaffle? go check
    self.got_the_quaffle()

    if self.gotthequaffle_flag:          # quaffle needs to follow
        print("--- got the Quaffle!!!, now move let's score ---")
        # we got it, now let's make some point$$$$!!!, go to the goal and score ... victory!
        # avoid Beater & Keeper, maybe implement later, lets move the goal for now.
        self.goto_location(self.robot_11_odom)      # move to goal Team B Robot_12
        self.check_if_scored()                      # check if we scored.. (got to the goal)


    else:
        print("I dont have the quaffle, let's go get it, goto quaffle")
        # I don't have the Quaffle yet, go get it ... move, move ...
        # avoid Beater & Keeper, maybe implement later, lets get the quaffle for now.
        self.goto_location(self.robot_8_odom)       # move to quaffle Robot_8
        self.check_collision()
```

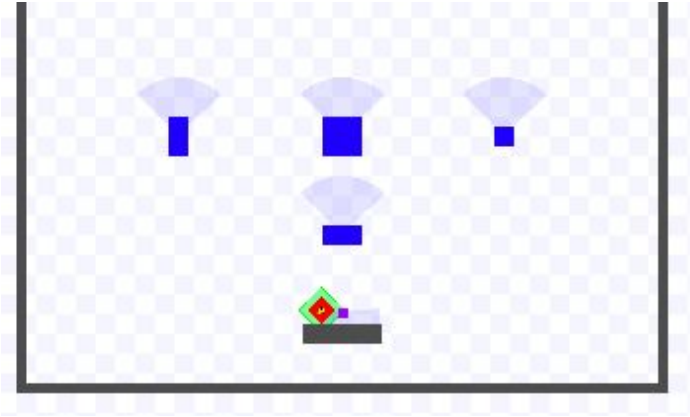


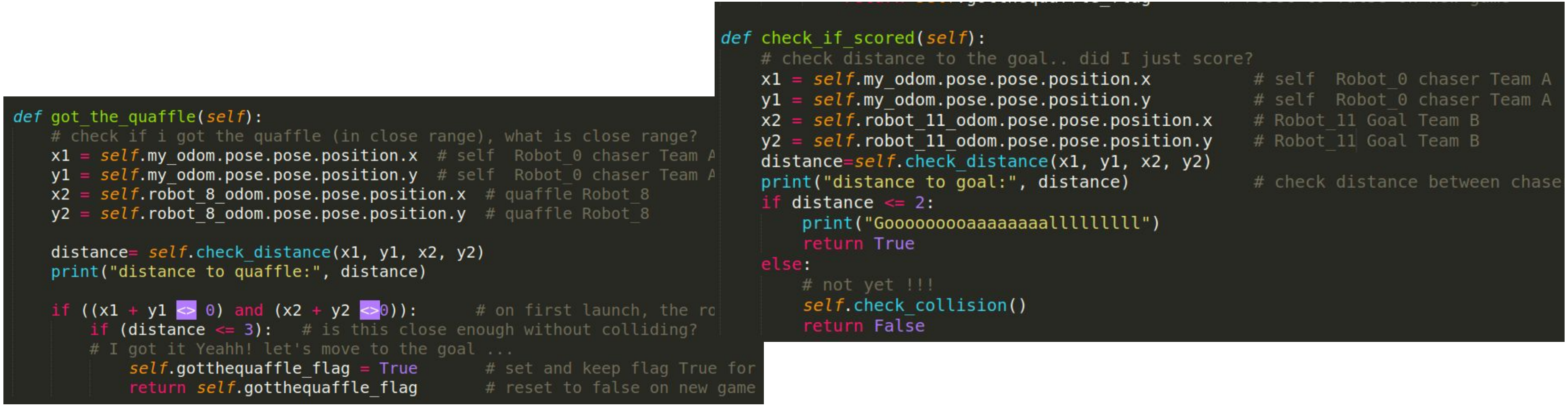
Chaser Objective: Score

Getting to the opposite team's Goal position



```
--- got the Quaffle!!!, now move let's score ---
('distance to goal:', 1.7259679532344163)
Goooooooaaaaaaaallllllll
('distance to quaffle:', 1.2640626916107442)
--- got the Quaffle!!!, now move let's score ---
('distance to goal:', 1.552662511302503)
Goooooooaaaaaaaallllllll
```





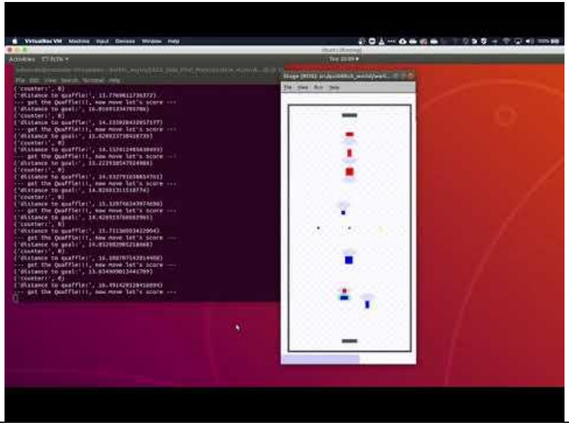
```
def check_if_scored(self):
    # check distance to the goal.. did I just score?
    x1 = self.my_odom.pose.pose.position.x          # self  Robot_0 chaser Team A
    y1 = self.my_odom.pose.pose.position.y          # self  Robot_0 chaser Team A
    x2 = self.robot_11_odom.pose.pose.position.x    # Robot_11 Goal Team B
    y2 = self.robot_11_odom.pose.pose.position.y    # Robot_11 Goal Team B
    distance=self.check_distance(x1, y1, x2, y2)
    print("distance to goal:", distance)            # check distance between chase
    if distance <= 2:
        print("Goooooooaaaaaaaallllllll")
        return True
    else:
        # not yet !!!
        self.check_collision()
        return False
```

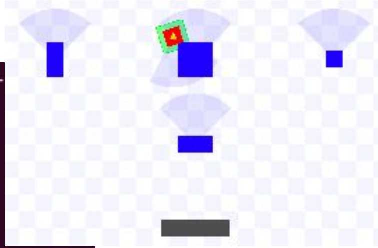
```
def got_the_quaffle(self):
    # check if i got the quaffle (in close range), what is close range?
    x1 = self.my_odom.pose.pose.position.x  # self  Robot_0 chaser Team A
    y1 = self.my_odom.pose.pose.position.y  # self  Robot_0 chaser Team A
    x2 = self.robot_8_odom.pose.pose.position.x  # quaffle Robot_8
    y2 = self.robot_8_odom.pose.pose.position.y  # quaffle Robot_8

    distance= self.check_distance(x1, y1, x2, y2)
    print("distance to quaffle:", distance)

    if ((x1 + y1 <> 0) and (x2 + y2 <>0)):       # on first launch, the ro
        if (distance <= 3):   # is this close enough without colliding?
        # I got it Yeahh! let's move to the goal ...
            self.gotthequaffle_flag = True       # set and keep flag True for
            return self.gotthequaffle_flag       # reset to false on new game
```

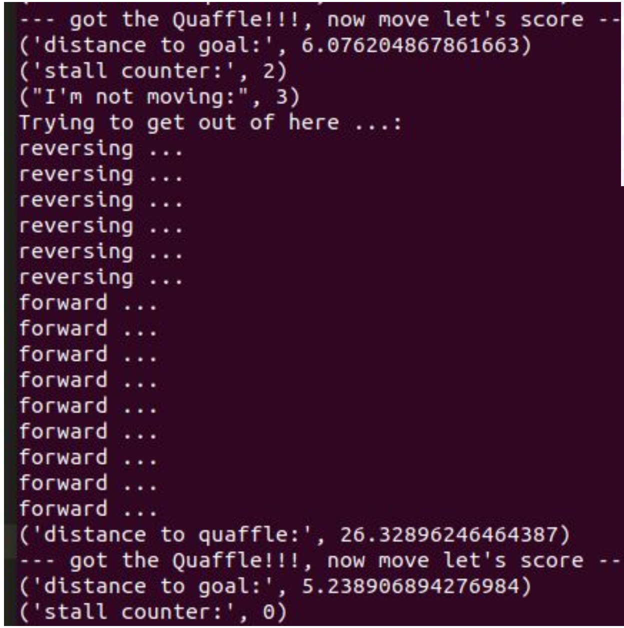
Collision recovery

- Implement method to recover when robot stops due to a collision.
- Chaser will find its way out of a stall, and continue moving to target.

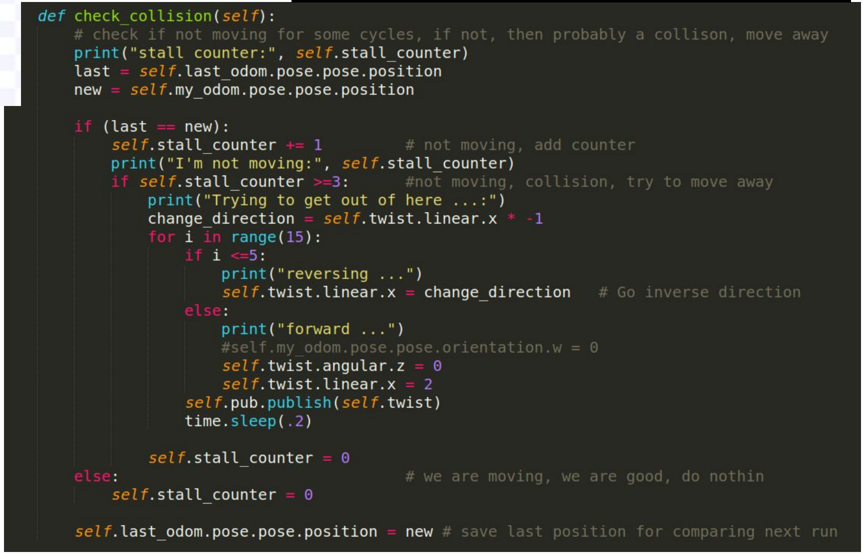
Demo video







```
--- got the Quaffle!!!, now move let's score ---
('distance to goal:', 6.076204867861663)
('stall counter:', 2)
("I'm not moving:", 3)
Trying to get out of here ...:
reversing ...
reversing ...
reversing ...
reversing ...
reversing ...
reversing ...
forward ...
forward ...
forward ...
forward ...
forward ...
forward ...
forward ...
forward ...
forward ...
('distance to quaffle:', 26.32896246464387)
--- got the Quaffle!!!, now move let's score ---
('distance to goal:', 5.238906894276984)
('stall counter:', 0)
```



```python
def check_collision(self):
    # check if not moving for some cycles, if not, then probably a collison, move away
    print("stall counter:", self.stall_counter)
    last = self.last_odom.pose.pose.position
    new = self.my_odom.pose.pose.position

    if (last == new):
        self.stall_counter += 1          # not moving, add counter
        print("I'm not moving:", self.stall_counter)
        if self.stall_counter >=3:       #not moving, collision, try to move away
            print("Trying to get out of here ...:")
            change_direction = self.twist.linear.x * -1
            for i in range(15):
                if i <=5:
                    print("reversing ...")
                    self.twist.linear.x = change_direction    # Go inverse direction
                else:
                    print("forward ...")
                    #self.my_odom.pose.pose.orientation.w = 0
                    self.twist.angular.z = 0
                    self.twist.linear.x = 2
                self.pub.publish(self.twist)
                time.sleep(.2)

            self.stall_counter = 0
    else:                                # we are moving, we are good, do nothin
        self.stall_counter = 0

    self.last_odom.pose.pose.position = new # save last position for comparing next run
```

Beater Node

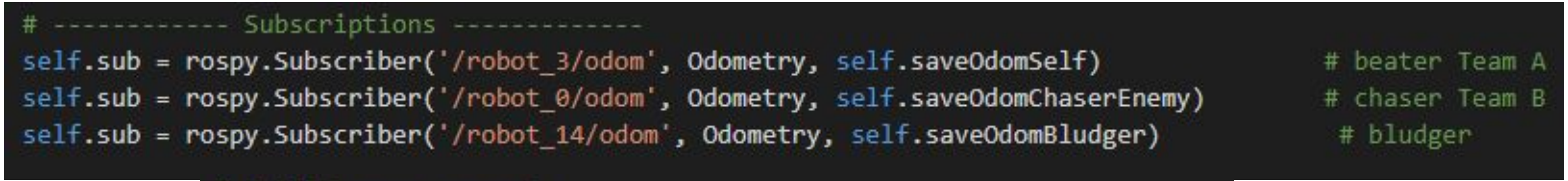
- It subscribes to the odometry topics for itself, the bludger, and the opposing chaser. It uses a Odometry message from the nav_msgs package.
- It publishes to its own cmd_vel topic using a Twist message type geometry_msgs package.

Odometer:

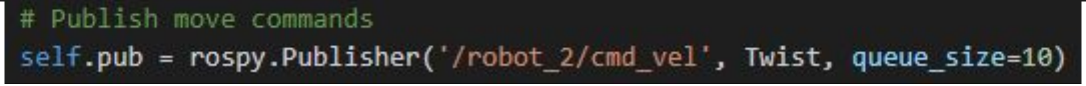
- Pose (x y z theta).
- Twist

Twist:

- Linear vector.
- Angular vector.

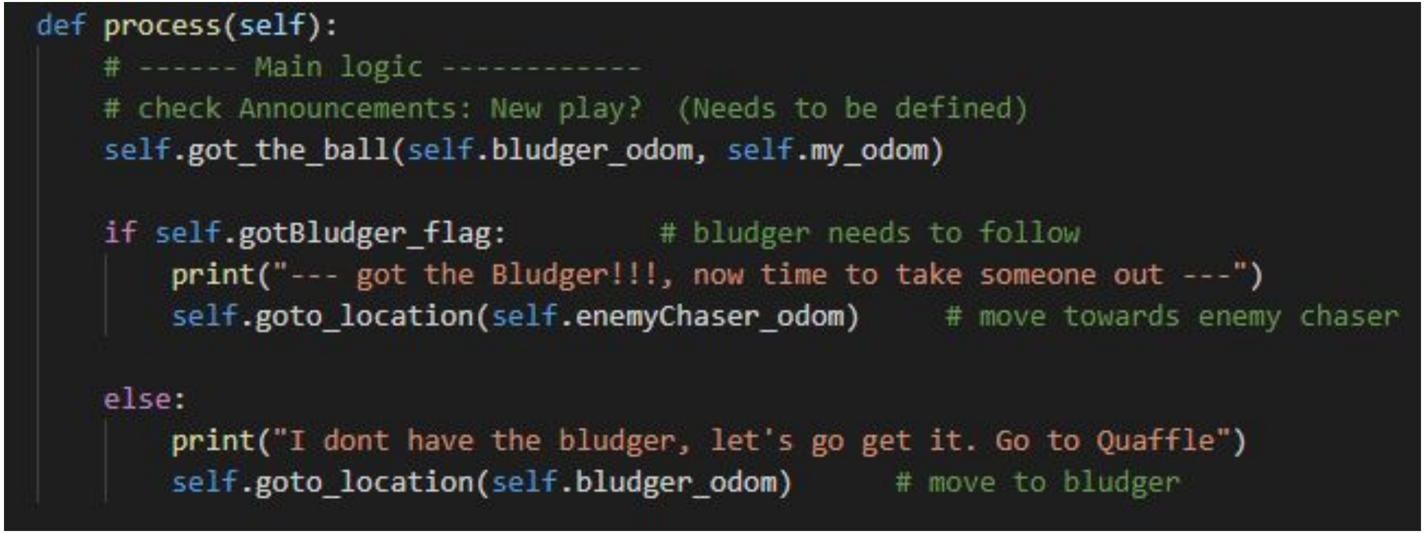


```
# ------------ Subscriptions -------------
self.sub = rospy.Subscriber('/robot_3/odom', Odometry, self.saveOdomSelf)               # beater Team A
self.sub = rospy.Subscriber('/robot_0/odom', Odometry, self.saveOdomChaserEnemy)        # chaser Team B
self.sub = rospy.Subscriber('/robot_14/odom', Odometry, self.saveOdomBludger)            # bludger
```



```
# Publish move commands
self.pub = rospy.Publisher('/robot_2/cmd_vel', Twist, queue_size=10)
```

Beater Node

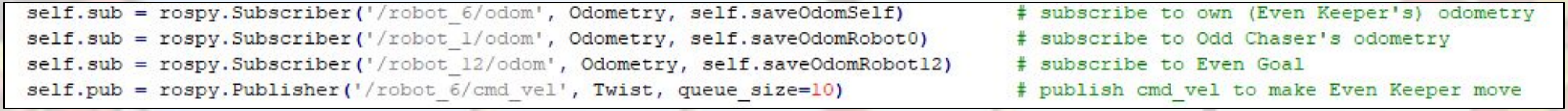


```
def process(self):
    # ------ Main logic ------------
    # check Announcements: New play?  (Needs to be defined)
    self.got_the_ball(self.bludger_odom, self.my_odom)

    if self.gotBludger_flag:          # bludger needs to follow
        print("--- got the Bludger!!!, now time to take someone out ---")
        self.goto_location(self.enemyChaser_odom)      # move towards enemy chaser

    else:
        print("I dont have the bludger, let's go get it. Go to Quaffle")
        self.goto_location(self.bludger_odom)       # move to bludger
```

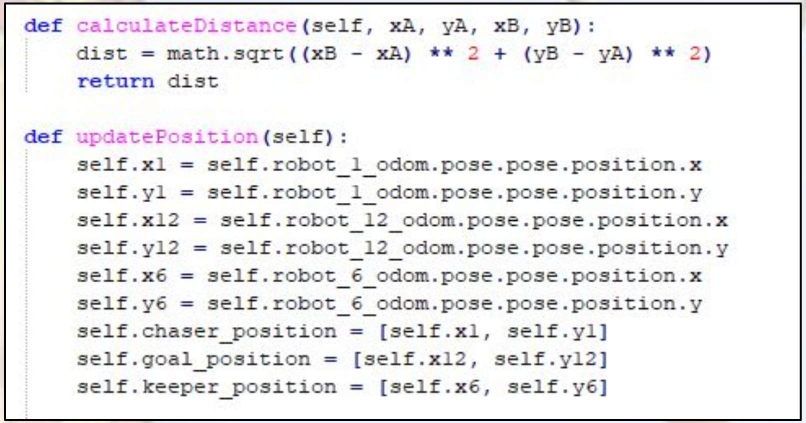
- It checks it position relative to the bludger in order to know if it is carrying it.
- If it doesn’t have the bludger it attempts to find it and capture it.
- If it does have the bludger it looks for the opposing Chaser and chases him in order to prevent him from scoring.

Keeper Node

- Guards Team Goal by monitoring opposing Chaser
- When Chaser gets within scoring distance, Keeper moves towards Chaser and tries to collide with him, thus blocking from scoring



```
self.sub = rospy.Subscriber('/robot_6/odom', Odometry, self.saveOdomSelf)          # subscribe to own (Even Keeper's) odometry
self.sub = rospy.Subscriber('/robot_1/odom', Odometry, self.saveOdomRobot0)        # subscribe to Odd Chaser's odometry
self.sub = rospy.Subscriber('/robot_12/odom', Odometry, self.saveOdomRobot12)      # subscribe to Even Goal
self.pub = rospy.Publisher('/robot_6/cmd_vel', Twist, queue_size=10)               # publish cmd_vel to make Even Keeper move
```

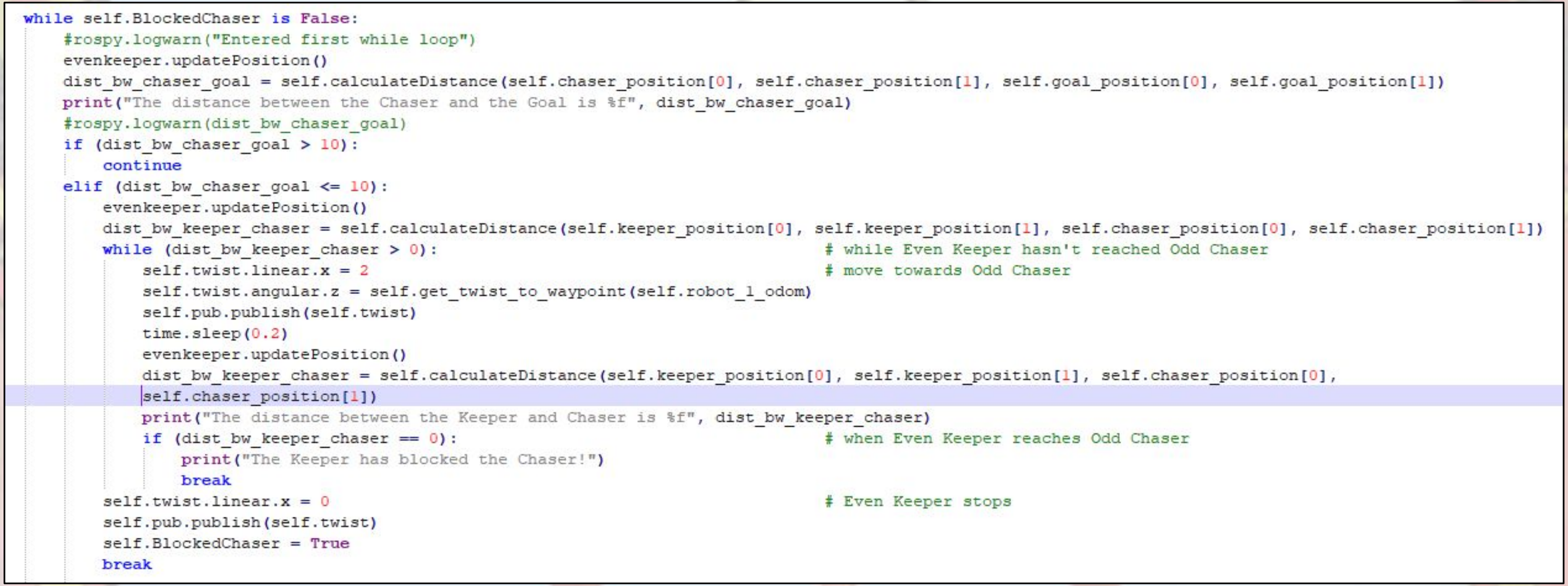


```
def calculateDistance(self, xA, yA, xB, yB):
    dist = math.sqrt((xB - xA) ** 2 + (yB - yA) ** 2)
    return dist

def updatePosition(self):
    self.x1 = self.robot_1_odom.pose.pose.position.x
    self.y1 = self.robot_1_odom.pose.pose.position.y
    self.x12 = self.robot_12_odom.pose.pose.position.x
    self.y12 = self.robot_12_odom.pose.pose.position.y
    self.x6 = self.robot_6_odom.pose.pose.position.x
    self.y6 = self.robot_6_odom.pose.pose.position.y
    self.chaser_position = [self.x1, self.y1]
    self.goal_position = [self.x12, self.y12]
    self.keeper_position = [self.x6, self.y6]
```

- Subscribed to opposing Chaser’s odometry, own odometry, own Goal’s odometry
- Publishing cmd_vel for movement

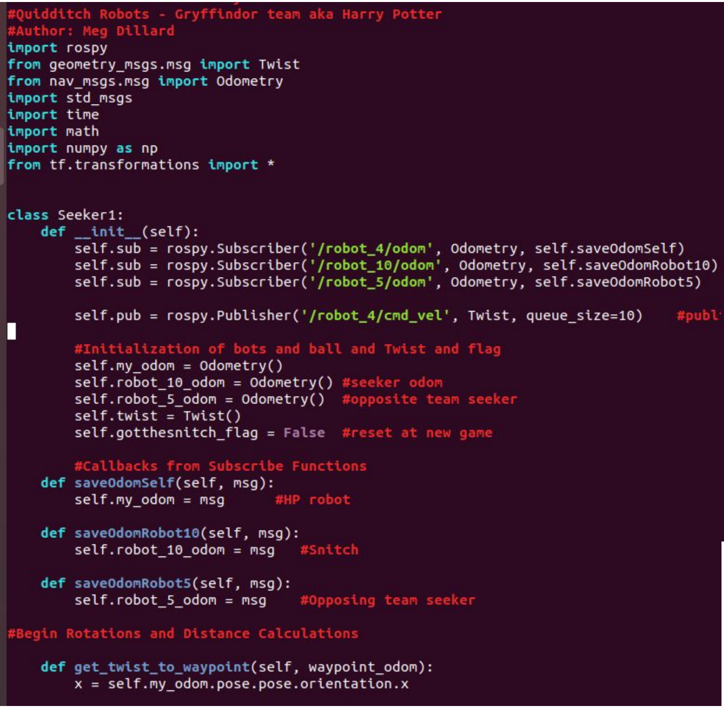
- Continuously updating position of Chaser to track
- Continuously updating position of self to detect collision/blocking of Chaser

- When Chaser within 10m of Goal, Keeper uses Twist to Waypoint function to turn and move towards incoming Chaser
- When distance between Keeper and Chaser drops to 0 => Chaser Blocked!

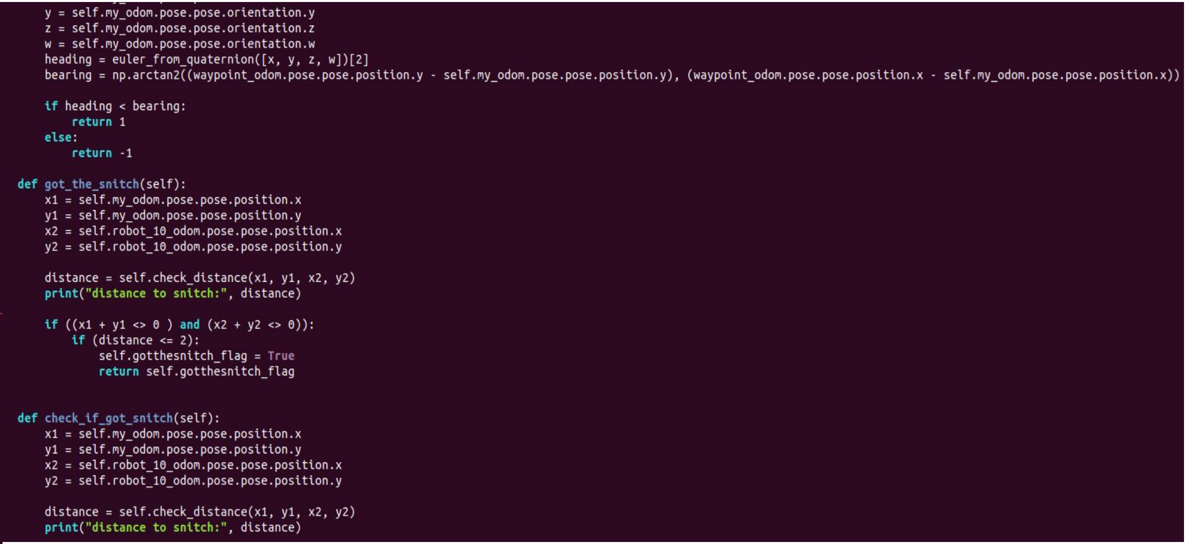


```
while self.BlockedChaser is False:
    #rospy.logwarn("Entered first while loop")
    evenkeeper.updatePosition()
    dist_bw_chaser_goal = self.calculateDistance(self.chaser_position[0], self.chaser_position[1], self.goal_position[0], self.goal_position[1])
    print("The distance between the Chaser and the Goal is %f", dist_bw_chaser_goal)
    #rospy.logwarn(dist_bw_chaser_goal)
    if (dist_bw_chaser_goal > 10):
        continue
    elif (dist_bw_chaser_goal <= 10):
        evenkeeper.updatePosition()
        dist_bw_keeper_chaser = self.calculateDistance(self.keeper_position[0], self.keeper_position[1], self.chaser_position[0], self.chaser_position[1])
        while (dist_bw_keeper_chaser > 0):                                       # while Even Keeper hasn't reached Odd Chaser
            self.twist.linear.x = 2                                              # move towards Odd Chaser
            self.twist.angular.z = self.get_twist_to_waypoint(self.robot_1_odom)
            self.pub.publish(self.twist)
            time.sleep(0.2)
            evenkeeper.updatePosition()
            dist_bw_keeper_chaser = self.calculateDistance(self.keeper_position[0], self.keeper_position[1], self.chaser_position[0],
            self.chaser_position[1])
            print("The distance between the Keeper and Chaser is %f", dist_bw_keeper_chaser)
            if (dist_bw_keeper_chaser == 0):                                     # when Even Keeper reaches Odd Chaser
                print("The Keeper has blocked the Chaser!")
                break
        self.twist.linear.x = 0                                                  # Even Keeper stops
        self.pub.publish(self.twist)
        self.BlockedChaser = True
        break
```

Seeker Node



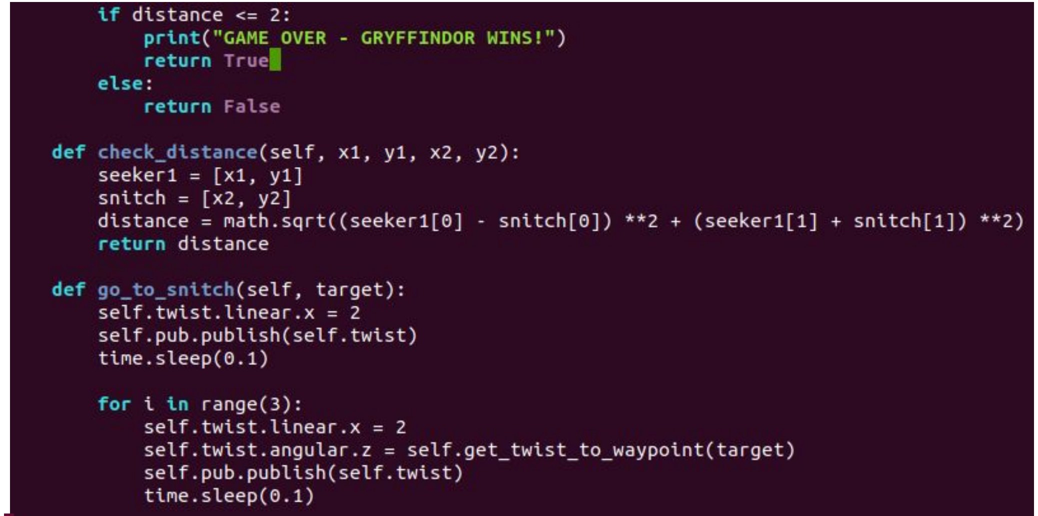
```
#Quidditch Robots - Gryffindor team aka Harry Potter
#Author: Meg Dillard
import rospy
from geometry_msgs.msg import Twist
from nav_msgs.msg import Odometry
import std_msgs
import time
import math
import numpy as np
from tf.transformations import *


class Seeker1:
    def __init__(self):
        self.sub = rospy.Subscriber('/robot_4/odom', Odometry, self.saveOdomSelf)
        self.sub = rospy.Subscriber('/robot_10/odom', Odometry, self.saveOdomRobot10)
        self.sub = rospy.Subscriber('/robot_5/odom', Odometry, self.saveOdomRobot5)

        self.pub = rospy.Publisher('/robot_4/cmd_vel', Twist, queue_size=10)    #publ

        #Initialization of bots and ball and Twist and flag
        self.my_odom = Odometry()
        self.robot_10_odom = Odometry() #seeker odom
        self.robot_5_odom = Odometry()  #opposite team seeker
        self.twist = Twist()
        self.gotthesnitch_flag = False  #reset at new game

        #Callbacks from Subscribe Functions
    def saveOdomSelf(self, msg):
        self.my_odom = msg      #HP robot

    def saveOdomRobot10(self, msg):
        self.robot_10_odom = msg   #Snitch

    def saveOdomRobot5(self, msg):
        self.robot_5_odom = msg    #Opposing team seeker

#Begin Rotations and Distance Calculations

    def get_twist_to_waypoint(self, waypoint_odom):
        x = self.my_odom.pose.pose.orientation.x
```



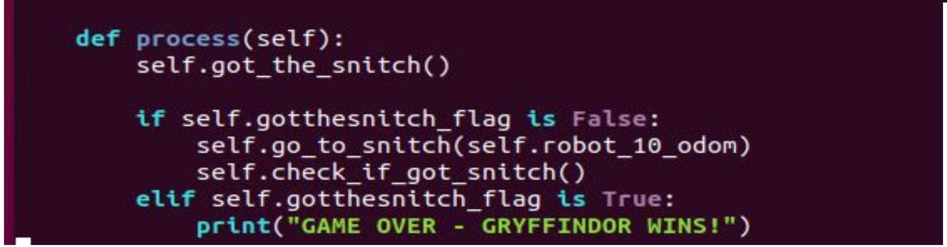
```
        y = self.my_odom.pose.pose.orientation.y
        z = self.my_odom.pose.pose.orientation.z
        w = self.my_odom.pose.pose.orientation.w
        heading = euler_from_quaternion([x, y, z, w])[2]
        bearing = np.arctan2((waypoint_odom.pose.pose.position.y - self.my_odom.pose.pose.position.y), (waypoint_odom.pose.pose.position.x - self.my_odom.pose.pose.position.x))

        if heading < bearing:
            return 1
        else:
            return -1

    def got_the_snitch(self):
        x1 = self.my_odom.pose.pose.position.x
        y1 = self.my_odom.pose.pose.position.y
        x2 = self.robot_10_odom.pose.pose.position.x
        y2 = self.robot_10_odom.pose.pose.position.y

        distance = self.check_distance(x1, y1, x2, y2)
        print("distance to snitch:", distance)

        if ((x1 + y1 <> 0 ) and (x2 + y2 <> 0)):
            if (distance <= 2):
                self.gotthesnitch_flag = True
                return self.gotthesnitch_flag


    def check_if_got_snitch(self):
        x1 = self.my_odom.pose.pose.position.x
        y1 = self.my_odom.pose.pose.position.y
        x2 = self.robot_10_odom.pose.pose.position.x
        y2 = self.robot_10_odom.pose.pose.position.y

        distance = self.check_distance(x1, y1, x2, y2)
        print("distance to snitch:", distance)
```

- Tracks the position of the snitch and the other seeker
- Bases its own position upon that of the current position of the snitch
- Will rotate and move until it gets close enough to “capture” the snitch and end the game

Seeker Node



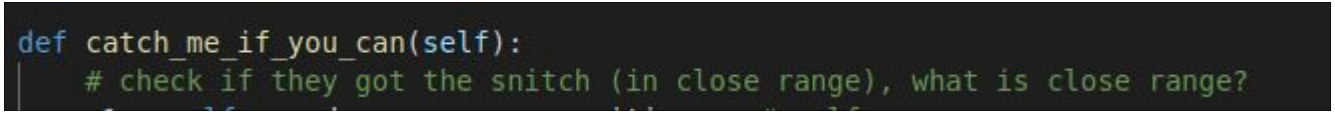
```
        if distance <= 2:
            print("GAME OVER - GRYFFINDOR WINS!")
            return True
        else:
            return False

    def check_distance(self, x1, y1, x2, y2):
        seeker1 = [x1, y1]
        snitch = [x2, y2]
        distance = math.sqrt((seeker1[0] - snitch[0]) **2 + (seeker1[1] + snitch[1]) **2)
        return distance

    def go_to_snitch(self, target):
        self.twist.linear.x = 2
        self.pub.publish(self.twist)
        time.sleep(0.1)

        for i in range(3):
            self.twist.linear.x = 2
            self.twist.angular.z = self.get_twist_to_waypoint(target)
            self.pub.publish(self.twist)
            time.sleep(0.1)
```



```
    def process(self):
        self.got_the_snitch()

        if self.gotthesnitch_flag is False:
            self.go_to_snitch(self.robot_10_odom)
            self.check_if_got_snitch()
        elif self.gotthesnitch_flag is True:
            print("GAME OVER - GRYFFINDOR WINS!")
```

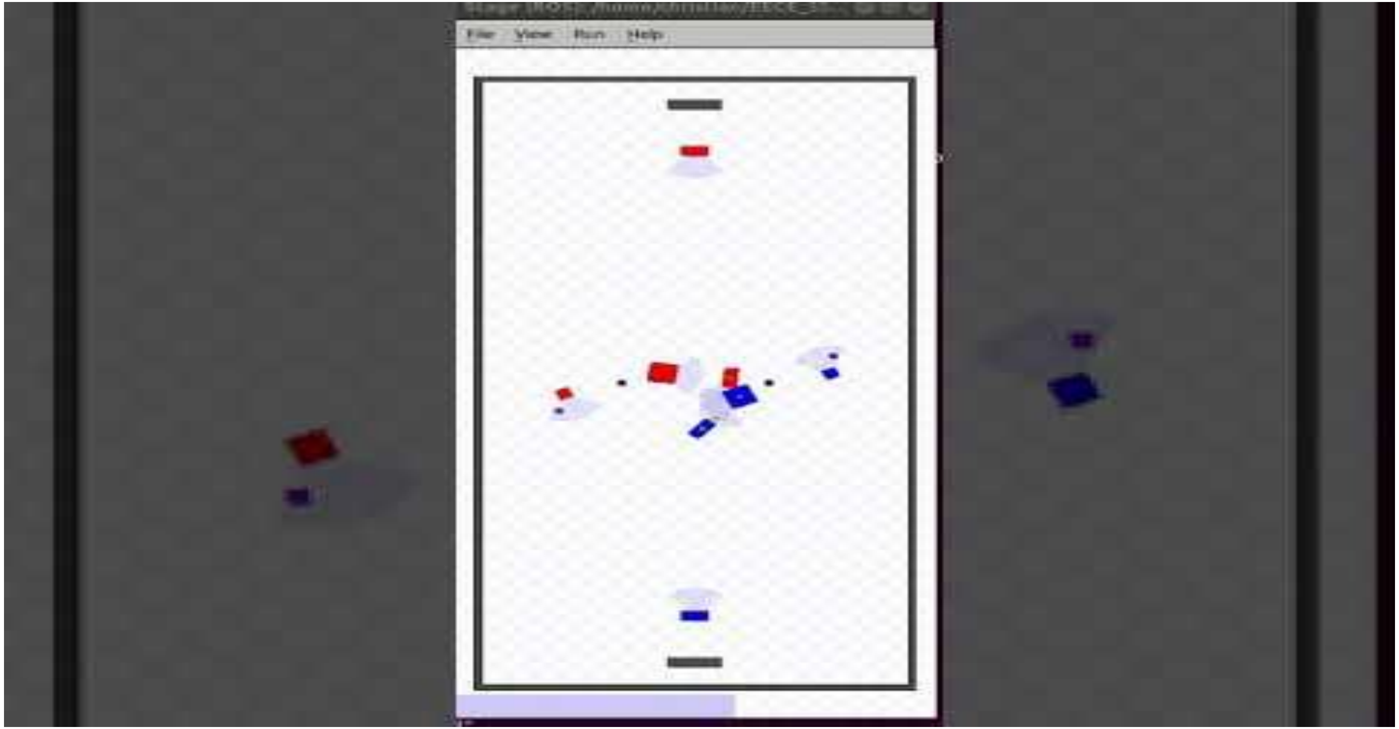
Snitch Node

- The Snitch will constantly try to avoid getting captured by the two Seeker robots. It’s sneaky
- If the Snitch robot has a Seeker robot within capture range, it will indicate that it was caught
- If two Seeker robots catch the Snitch at the same time, the Snitch will be responsible for breaking ties between two Seekers. The decision of the Snitch is final
- The Snitch will try very hard to not get caught. It will be faster than the seekers
- The Snitch will avoid collisions at any cost. This is the best time to catch it
- In the code implementation we keep track of the Seekers position at all times and make decisions based on this. We can either turn around, go in reverse, or increase or decrease velocity
- Future improvements



```
def catch_me_if_you_can(self):
    # check if they got the snitch (in close range), what is close range?
```

Demo



Future Modifications

- Improve object avoidance between most robots except those that are intended to collide and block each other.
- Trying 3D simulations with a more detailed representation of the Quidditch field and the robots potentially using a more modern, updated simulation package known as Gazebo.

- Instantiate more Chasers, Beaters on each team and implement an algorithm by which they work together to achieve their objective
- Implement more realistic gameplay with scoring/announcements

- Implement a reinforcement learning algorithm for each agent using a continuous timespace, linear function approximation (polynomial). An appropriate algorithm could be Actor Critic with Eligibility traces with a non greedy approach which encourages exploration and creates variability in the agent's movements.

Questions?