# Weekly project – Python 3 for Robotics

##### Emmanouil Markodimitrakis - October 2021

I have spitted my code in two classes, one for the Controller (controller\_module.py) and one for the Robot (robot\_module.py), both have initialization function in order to initialize the properties. The Robot class has also setters and getters for every property. Below I describe the classes in more details. The main (main.py) module is the main script of the project that I initialize all the classes and implement the core functionality.

## Robot module

Contains the Robot class, setters and getters and a function to update the robot’s position.

### Properties, setters & getters

* **name (str)** – The name of the robot
* **max\_speed (float)** – The robot’s maximum speed, set by the initial function
* **initial\_pos ((float), (float))** – The initial position of the robot, set by the user and changes only on the initialization
* **current\_pos ((float), (float))** – Current position is firstly inisialized equal to the initial\_pos and it get updated constantly by the update\_position function
* **rotational\_speed (float)** – The rotational speed of the robot, set to 0 by default
* **forward\_speed (float)** – The forward speed of the robot, set to 0 by default

### Functions

* **update\_position(velocity)** – Is called by the main module, get as input the velocity and update the robot’s position by adding the velocity to the current position.
* def update\_position(self, vdt):
* """
* Update the robot's current position
* Parameters:
* velocity (float): Velocity
* """
* self.\_\_current\_pos = self.\_\_current\_pos + velocity

## Controller Module

Contains the Controller class, and some functions in order to implement the proportional control.

### Properties

* **forward\_speed\_gain (float)** – Initialized on the class construction with a constant that is used to calculate the proportional forward speed.
* **rotational\_speed\_gain (float)** - Initialized on the class construction with a constant that is used to calculate the proportional rotational speed.

### Functions

* **distance\_to\_target(current\_pos, target\_pos)** – Get as input the current of the robot and target position, calculate the Euclidean distance and return it.
* def distance\_to\_target(self, current\_pos, target\_pos):
* """
* Calculate the Euclidean distance between the points (X, Y) and (X', Y')
* Parameters:
* current\_pos ((float), (float)): (X, Y) - Current position
* targer\_pos ((float), (float)): (X', Y') - Target position
* Returns:
* euclidean\_distance (float): L2 Euclidean distance between the two points
* """
* current\_pos = np.array(current\_pos)
* target\_pos = np.array(target\_pos)
* euclidean\_distance = np.sqrt(np.power(current\_pos - target\_pos, 2).sum())
* return euclidean\_distance
* **calculate\_theta(current\_pos, target\_pos)** – Get as input the current of the robot and target position, calculate and return the following formula atan2( (Y’ – Y) , (X’ – X) ). This is the theta angle that we need to calculate the velocity.

Diagram

Description automatically generated with medium confidence

def calculate\_theta(self, current\_pos, target\_pos):

        theta = math.atan2( (target\_pos[1] - current\_pos[1]), (target\_pos[0] - current\_pos[0]) )

        return theta

* **calculate\_velocity(K, theta, dt)** – Where input K is the proportional constant that the user specifies, theta is the angle we calculate in the above function and dt is the iteration time. Using these variables it calculates the velocity with the following formula:

V ∆t= (Vx,Vy)∆t= (Vf.cos(θ),Vf.sin(θ))∆t= (Vf.cos(θk−1+ω∆t),Vf.sin(θk−1+ω∆t))∆t

* def calculate\_velocity(self, K, theta, dt):
* """
* Calculate forward speed and angular velocity
* Parameters:
* K (int): Proportional constant
* theta (float): Angle
* dt (float): Iteration time
* Returns:
* velocity (float): Velocity
* """
* velocity = np.array([self.forward\_speed \* math.cos(theta + self.rotational\_speed \* dt), self.forward\_speed \* math.sin(theta + self.rotational\_speed \* dt)])
* return velocity
* **control(theta\_error, K, dt, current\_pos, target\_pos) –** This is the main function of this module, it calculates two errors, distance\_error is used to update the forward speed and theta\_error to update the rotational speed. It follows the logic of a proportional controller, by correcting the robot’s orientation first with forward speed equals to 0, proportional to the theta error. Once the orientation is corrected, the robot starts moving with forward speed, proportional to the distance error.

def control(self, distance\_to\_target, current\_theta, dt, current\_pos, target\_pos):

        target\_theta = self.calculate\_theta(current\_pos, target\_pos)

        theta\_error = self.rotational\_speed\_gain \* (target\_theta - current\_theta)

        distance\_error = self.forward\_speed\_gain \* (distance\_to\_target)

        velocity = self.forward\_speed\_gain \* self.calculate\_velocity(target\_theta, dt, theta\_error, distance\_error)

        # 5 degrees = 0.08 rads

        if current\_theta == 0 or theta\_error >= 5:

            # Stop the robot and correct the angle again

            velocity = 0

            forward\_speed = 0

            print(f"theta\_error:{theta\_error} \t for\_speed:{forward\_speed} \t velocity{velocity}")

            return target\_theta, velocity, theta\_error, forward\_speed

        elif theta\_error < 5:

            # Forwad speed can be non-zero, while theta orentation still being corrected

            print(f"theta\_error:{theta\_error} \t for\_speed:{distance\_error} \t velocity{velocity}")

            return target\_theta, velocity, theta\_error, distance\_error

## Main Module

First of all, we initialize the attributes and the we initialize two object, one for the robot and another for the controller. Then we run the visualization loop and inside that loop we call the controller’s function move\_robot in order to calculate the velocity and change the robot’s position.

### Attributes

* **K = 1 –** The constant we adjust to change robot behaviour
* **iteration\_time\_sec = 0.001** – Iteration time 0.001 milisecond
* **target\_pos = np.array([1, 2])** – Initialize target position to X=1 and Y=2
* **current\_time = time.time()** – Initialize current time
* **theta\_error = 0**  - Initialize theta error to zero

### Objects

* **robot = Robot( "STUPIDO", 0.03, np.array([1, 1]))** 
  + robot\_name set to ‘STUPIDO’
  + max\_speed set to 0.03
  + initial\_pos set to point X=1, Y=1
* **controller = Controller(robot.forward\_speed, robot.rotational\_speed)**
  + forward\_speed set equal to robot’s forward\_speed
  + rotational\_speed set equal to robot’s rotational\_speed

### Functions

* **move\_robot()** – Is executed inside the visualization loop in order to compute the distance to target, call the controllers control function, and get as output the new theta, velocity, rotational speed and forward speed values.

Furthermore, this function updates the robot’s position by calling the update\_position function with velocity as input and set the new speeds to the robot object. The function returns False if the distance to target is lower that 0.01 and terminates the outer visualization loop.

def move\_robot():

    distance\_to\_target = controller.distance\_to\_target(robot.current\_pos, target\_pos)

    if distance\_to\_target < 0.01:

        print('Target reached!')

        return False

    else:

        # Compute forward and rotation speed with controller

        # set speed to robot

        global current\_time

        t\_k = current\_time

        time.sleep(iteration\_time\_sec)

        current\_time = time.time()

        dt = current\_time - t\_k

        # Update robot pos with t\_k and current\_time

        global theta

        theta, velocity, rotational\_speed, forward\_speed = controller.control(distance\_to\_target, theta, dt, robot.current\_pos, target\_pos)

        robot.rotational\_speed = rotational\_speed

        robot.forward\_speed = forward\_speed

        robot.update\_position(velocity)

        # print(f"distance\_to\_target: {distance\_to\_target} \t dt:{dt} \t  Current Pos:{robot.current\_pos} \t Theta error:{theta\_error} ")

        return True