Umeå University
Department of Computing Science

Fundamentals of artificial intelligence
5DV121

Follow the path

Thomas Ranvier *Valentin Lecompte*

supervised by
Thomas Johannson
Carl-Anton Anserud
Anders Broberg
Adam Dahlgren Lindström
Jenny Nilsson
Ola Ringdahl

September 27, 2018

Abstract

The first assignment of this course was about developing a path tracking algorithm for a virtual robot. We implemented two algorithms, the first one is the known 'pure pursuit' algorithm. The second one is a simpler and more naive algorithm that we developed ourselves. After the implementation of those two algorithms we had various ideas of improvements.

An idea was to adapt the linear speed of the robot to the computed angular speed, meaning that the robot should slow down on sharp turns and go faster whenever the path is more straight. We managed to do it by using several mathematical functions that return an adapted linear speed given an angular speed.

We also had the idea to adapt the look-ahead distance to the linear speed, to do so a very simple modification was to set the look-ahead distance to the same value as the linear speed.

After numerous tests we concluded that:

- The fastest way to follow a path is using our algorithm with a function for the linear speed and the look-ahead distance equal to the linear speed.
- The most secured way to follow the path is by using the 'pure pursuit' algorithm with a function for the linear speed and the look-ahead distance set to 0.7.

Contents

Introduction

There are more and more robots in the world nowadays, that leads the developers and scientists to develop and create new algorithms that will make the robots work more efficiently in their environments.

Many of those robots need to move in their surroundings, to make the robot able to move by itself there are many solutions, the simplest of them all is to give that robot a path and develop an algorithm that will make the robot follow it. Those kinds of algorithms are called path tracking algorithms.

A more advance type of path tracking algorithm can detect the potentials obstacles on the way in order to make the robot avoid them.

Chapter 1

Context

The goal of the first assignment of this course was to develop a path tracking algorithm for a virtual robot. We first choose to implement the ‘pure pursuit’ algorithm, then we also created a simpler algorithm.

1.1 Pure pursuit

Pure pursuit is a path tracking algorithm. [Lun03, section 1.2] It computes the angular velocity that the robot needs to follow in order to reach some look-ahead point in front of itself. The linear speed is not computed by this algorithm, it can either stays constant or be modified according to the computed angular velocity. At every new step the algorithm selects a goal point based on the selected look-ahead distance until it reaches the last point of the path. The robot will then always follow a point that stays at a look-ahead distance in front of itself. The look-ahead distance can either be constant or be modified during the running process of the algorithm.

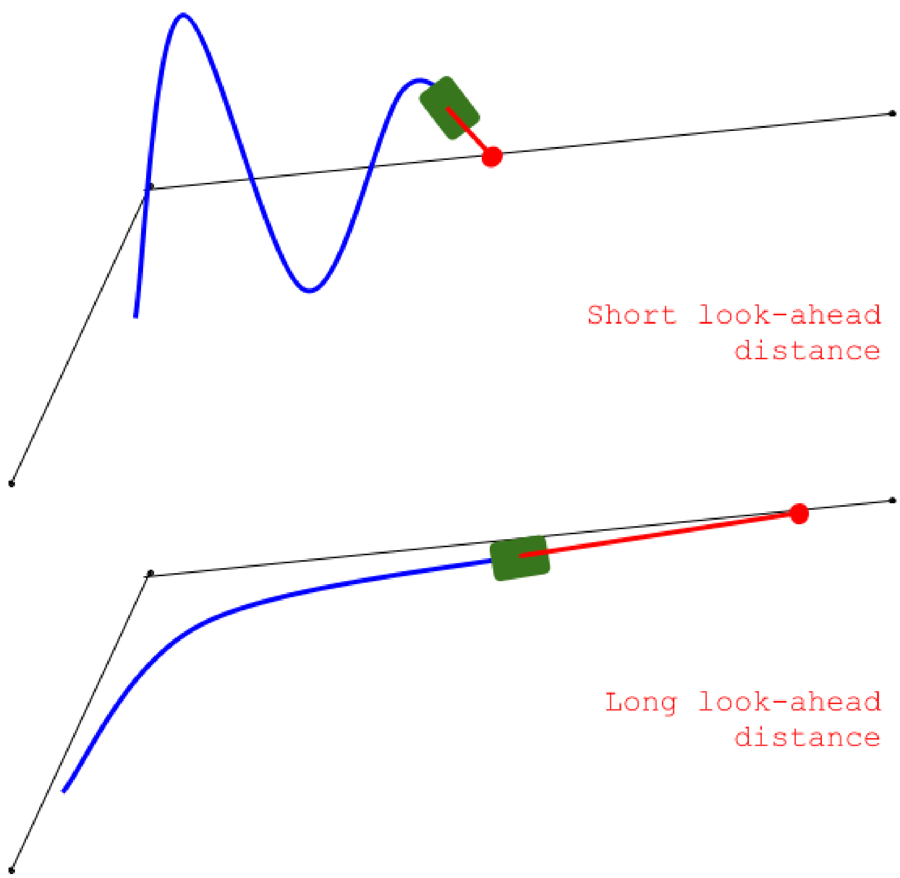
1.2 Look-ahead distance

The look-ahead distance is a very important value for the pure pursuit algorithm. This distance corresponds to how far along the path the robot looks from its current location. The algorithm then computes the angular velocity that the robot must have to orientate itself towards its goal point.

Changing that value changes the path that the robot will follow:

- A short look-ahead distance will make the robot stay close to the path but it will often oscillate around it.
- A long look-ahead distance will cause the robot to cut too much when the path turns a lot.

Figure 1: Look-ahead distance comparison



The look-ahead distance must be chosen so that the algorithm can follow the path as best as possible depending on the environment.

1.3 Our own algorithm

After the pure pursuit implementation we wondered if it was possible to create a simpler algorithm.

We came with a simple idea which is to define the robot angular speed like so:

– We compute the angle between the robot orientation and the goal point: Θ.

- We define a trigger value: x.
- If $abs(\Theta) >= x$ we define $angular\_speed$ as the $\pm max\_angular\_speed$ corresponding to the sign of Θ.
- Else we compute a weighted angular speed:

$$angular\_speed = \frac{max\_angular\_speed * \Theta}{x}$$

By implementing an algorithm as above and testing for numerous x values we managed to obtain results very similar to the pure pursuit algorithm and even better after more experimentation.

Chapter 2

Implementation on MRDS

2.1 Path tracking algorithms

2.1.1 Pure pursuit

We implemented the pure pursuit algorithm as follows:

Algorithm 1 Pure pursuit

```
procedure PURE_PURSUIT(robot_pos, goal_point)
    dist = compute the distance between the robot and the goal point
    robot_angle = compute the angle of the robot in the MRDS environment
    point_angle = compute the angle of the goal point using the robot as referential
    theta = point_angle − robot_angle
    delta_x = sin(theta)/dist
    return (2 ∗ delta_x)/(dist ∗ ∗2)
end procedure
```

After implementing this algorithm we have tested it using a linear speed of 1, which is the maximum linear speed for the robot. The result was already very convincing since the robot successfully followed the path without even hitting any obstacle.

After that we wanted to do some more testing, to do so we had three main ideas:

1. Develop a more simple and naive algorithm by ourselves so we can compare the pure pursuit algorithm to another one.

2. Adapt the linear speed of the robot to the computed angular speed. Meaning that the robot should slow down when there is a sharp turn and go as fast as possible otherwise.

3. Adapt the look-ahead distance to the linear speed, so the look-ahead is shorter when the linear speed decreases and longer conversely.

2.1.2 Our own algorithm

We implemented our algorithm as follows:

Algorithm 2 Our algorithm

procedure OUR_ALGO($robot\_pos, goal\_point$)
 $robot\_angle =$ compute the angle of the robot in the MRDS environment
 $point\_angle =$ compute the angle of the goal point with the robot as axis
 $theta = sin(point\_angle - robot\_angle)$
 $x = 0.3$ ▷ trigger value
 $max\_ang\_speed = 3$
 $ang\_speed = (max\_ang\_speed * theta)/x$
 return $min(max(ang\_speed, -max\_ang\_speed), max\_ang\_speed)$
end procedure

For this algorithm to work properly we have to chose a correct x value. At first we tried by setting a very low value: $x = 0.05$, we also computed $theta$ as follows: $theta = point\_angle - robot\_angle$, without the sinus. The result of such settings were a very oscillating path and without the sinus $theta$ were going from negative to positive, which made the robot briefly lose the path.

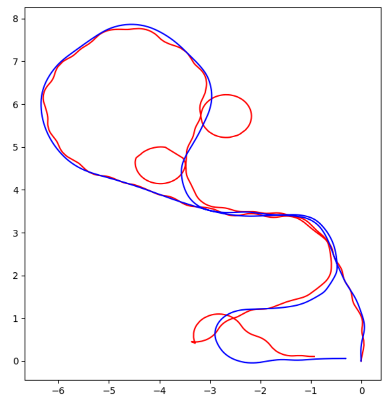


Figure 2: Our algorithm, $x = 0.05$, *theta* without the sinus

After seeing that result we figured out a way of finding a good x value and added the sinus to *theta*.

To find the x value we have displayed the *theta* value during a run and we have figured out that when the robot is on a good trajectory to follow the path the $abs(theta)$ value is usually bellow 0.5. We then tried to run our algorithm using different x values around 0.5 and the value which works the best is $x = 0.3$. With that value the algorithm works perfectly fine, we obtained results sensibly similar to a pure pursuit run.

2.2 Diverse improvements

2.2.1 Compute the linear speed depending on the angular speed

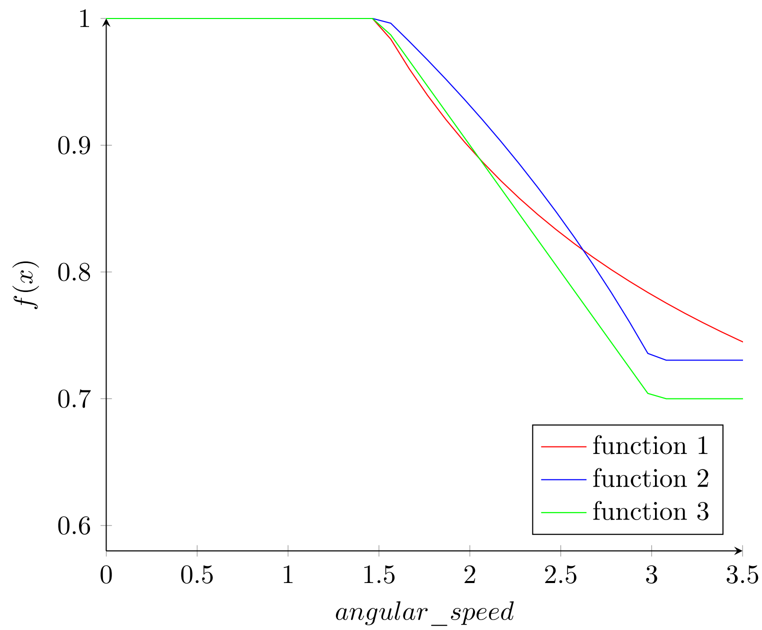
After the implementation of those two algorithms we wanted to try out to adapt the linear speed of the robot to the computed angular speed. Since the maximum linear speed of the robot is 1 we tried to find different functions that would maximize the linear speed when the angular speed is around 0 and minimize it when the angular speed is around 3.

We have found three functions that correspond to our needs:

$$min(1, 1/log_{10}(6 * abs(x) + 1)) \tag{1}$$

$$min(1, log_{10}(-(min(3, abs(x)) - 4.7)) + 0.5) \tag{2}$$

$$min(1, -0.2 * min(3, abs(x)) + 1.3) \tag{3}$$



2.2.2 Adapt the look-ahead distance to the linear speed

After many tests with different functions to adapt the angular speed we figured out that adapting the look-ahead distance to the linear speed would probably be a good idea. The idea is to reduce the look-ahead distance when the robot slows down, and increase it conversely.

Using our functions the maximum linear speed is 1 and the minimum is around 0.7, we decided to try to set the look-ahead distance at the same value as the linear speed. By doing that we obtained some very good results using our own algorithm, on the other hand the robot always fails to follow the path using the pure pursuit algorithm.

2.3 How to run our program

To run our program you need to have python2 installed on your machine, then you just need to enter the following command with the parameters you want:

```
python2 path_tracking.py path.json X Y
```

$$X = \begin{cases} 1\text{: pure pursuit algorithm} \\ 2\text{: our own algorithm} \end{cases} \quad Y = \begin{cases} 1\text{: } linear\_speed = 1 \\ 2\text{: apply function 1 to } linear\_speed \\ 3\text{: apply function 2 to } linear\_speed \\ 4\text{: apply function 3 to } linear\_speed \end{cases}$$

If the optional parameter `--`*ahead* is set, the *look_ahead* will be equal to the *linear_speed*, 0.7 otherwise. You can also set the optional parameter `--`*plot* to display the path of the robot. If the ip address of the MRDS server if not *localhost*, you can set it with the `--`*ip* parameter.

Chapter 3

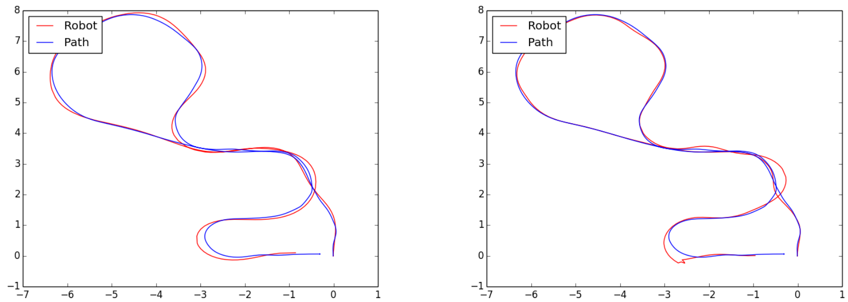
Experimentation and analysis

In this chapter we will present the results that we obtained after all our tests.

On the following table are the times in seconds obtained for the path 'around the table and back' with different settings:

Algorithm	$linear\_speed$	$look\_ahead = 0.7$	$look\_ahead = linear\_speed$
Pure pursuit	1	29.39	Fail
	function 1	30.62	Fail
	function 2	29.99	Fail
	function 3	32.76	Fail
Our algorithm	1	29.57	29.48
	function 1	29.20	27.54
	function 2	29.64	27.56
	function 3	29.83	27.79

We can see that with a linear speed set at 1 and a look-ahead distance at 0.7 the pure pursuit and our algorithm both have very similar times. But when we analyze the paths that the robot followed in both cases we can see that the linear speed is too high for our algorithm, causing a crash on the last sharp turn:



(a) Pure pursuit (b) Our algorithm

Figure 3: $linear\_speed = 1$, $look\_ahead = 0.7$

We can see that with that look ahead distance, both algorithms shifted away from the path in the sharp turns.

An interesting result is to see how the fact that the robot slows down in sharp angles changes its path, in the following image we can see that the robot stays very close to the path the entire time:

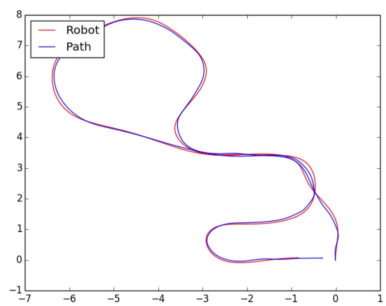


Figure 4: Pure pursuit, $linear\_speed =$ function 1, $look\_ahead = 0.7$

The faster result that we obtained was using our own algorithm, with the function 2 and a look-ahead distance equals to the linear speed. With those settings the robot cuts inside every turn but stays very close to the path as soon as it's more straight. It was able to follow the path in 27.54 seconds.

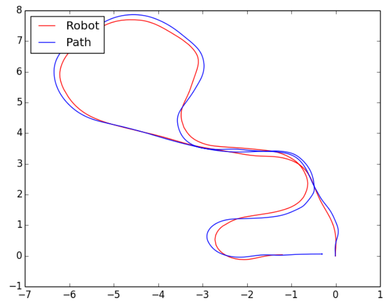


Figure 5: Our algorithm, $linear\_speed$ = function 1, $look\_ahead$ = $linear\_speed$

Of course with that kind of settings if the path was going closer to obstacles in the inside of sharp turns the robot would most likely hit them and fail to follow the path.

By analyzing those results we can conclude that:

- The safest way to follow a path is to use the pure pursuit algorithm, using the function 1 and a look-ahead distance set at 0.7.
- The fastest way to follow a path is to use our algorithm, using the function 1 and a look-ahead distance equal to the linear speed.

Conclusion

For this assignment we have implemented a pure pursuit algorithm and developed another path tracking algorithm.

On the day of the examination we first tried a run using the fastest of our settings, meaning our algorithm with a logarithmic function on the linear speed and the look-ahead distance adapted to that linear speed. The result of that run was a crash on the turn behind the sofa, indeed as concluded in chapter 3 those settings make the robot cut the turns, since the sofa was very close in the inside of the turn the robot hit it and failed the run.

For the second run we have used our safest settings, being the pure pursuit algorithm with a logarithmic function on the linear speed and a constant look-ahead distance of 0.7. This second turn was as expected successful, the robot staying very close to the path didn't hit any obstacles. With those settings the robot was able to follow the path of the examination in 44 seconds.

To go further and minimize the time of travel we could use the 'lasers' of the robot to detect when it is too close from an obstacle and so adapt the linear speed and the look-ahead distance depending on the distance.

List of Algorithms

List of Figures

Bibliography

[Con92] Craig Conlter. Implementation of the pure pursuit path 'hcking algorithm. Master's thesis, Camegie Mellon University, 1992. https://www.ri.cmu.edu/pub_files/pub3/coulter_r_craig_1992_1/coulter_r_craig_1992_1.pdf.

[Lun03] Martin Lundgren. Path tracking for a miniature robot. Master's thesis, Umeå University, 2003. http://www8.cs.umu.se/kurser/TDBD17/VT06/utdelat/Assignment%20Papers/Path%20Tracking%20for%20a%20Miniature%20Robot.pdf.

Appendix

Source code

```
import httplib, json, time, sys
import matplotlib.pyplot as plt
import argparse
from math import *

HEADERS = {"Content-type": "application/json", "Accept": "text/json"}

parser = argparse.ArgumentParser()
parser.add_argument("path", help="The file containing the path (jsonFile).")
parser.add_argument("algo", type=int, help="id of the algo to use:\n\t1. Pure pursuit\n\t2. Our algo")
parser.add_argument("func", type=int, help="id of the function to use to calculate the linear speed:\n\t1. constant=1\n\t2. 1/log10(6*abs(angle)+1)\n\t3. log(-abs(angle)+0.5\n\t4. -0.2*abs(angle)+1.3")
parser.add_argument("--plot", action="store_true", help="dislay the path to follow and the robot's path")
parser.add_argument("--ahead", action="store_true", help="If set, the look ahead is equal to the speed, 0.7 otherwise")
parser.add_argument("--ip", default="localhost", help="Ip address of the MRDS server. Default : localhost")

args = parser.parse_args()

MRDS_URL = args.ip + ":50000"

class UnexpectedResponse(Exception): pass

def post_speed(angular_speed, linear_speed):
```

```
    """Sends a speed command to the MRDS server"""
    mrds = httplib.HTTPConnection(MRDS_URL)
    params = json.dumps({'TargetAngularSpeed':
                angular_speed, 'TargetLinearSpeed':
                linear_speed})
    mrds.request('POST', '/lokarria/differentialdrive',
                params, HEADERS)
    response = mrds.getresponse()
    status = response.status
    if status == 204:
        return response
    else:
        raise UnexpectedResponse(response)

def get_pose():
    """Reads the current position and orientation from the
                MRDS"""
    mrds = httplib.HTTPConnection(MRDS_URL)
    mrds.request('GET', '/lokarria/localization')
    response = mrds.getresponse()
    if response.status == 200:
        pose_data = response.read()
        response.close()
        return json.loads(pose_data)
    else:
        raise UnexpectedResponse(response)

def bearing(q):
    return rotate(q, {'X': 1.0, 'Y': 0.0, "Z": 0.0})

def rotate(q, v):
    return vector(qmult(qmult(q, quaternion(v)), conjugate(
                q)))

def quaternion(v):
    q = v.copy()
    q['W'] = 0.0
    return q

def vector(q):
    v = {}
    v["X"] = q["X"]
    v["Y"] = q["Y"]
    v["Z"] = q["Z"]
    return v

def conjugate(q):
    qc = q.copy()
    qc["X"] = -q["X"]
    qc["Y"] = -q["Y"]
```

```python
    qc["Z"] = -q["Z"]
    return qc

def qmult(q1, q2):
    q = {}
    q["W"] = q1["W"] * q2["W"] - q1["X"] * q2["X"] - q1["Y"] * q2["Y"] - q1["Z"] * q2["Z"]
    q["X"] = q1["W"] * q2["X"] + q1["X"] * q2["W"] + q1["Y"] * q2["Z"] - q1["Z"] * q2["Y"]
    q["Y"] = q1["W"] * q2["Y"] - q1["X"] * q2["Z"] + q1["Y"] * q2["W"] + q1["Z"] * q2["X"]
    q["Z"] = q1["W"] * q2["Z"] + q1["X"] * q2["Y"] - q1["Y"] * q2["X"] + q1["Z"] * q2["W"]
    return q

def get_heading():
    """Returns the XY Orientation as a bearing unit vector"""
    return bearing(get_pose()['Pose']['Orientation'])

def get_position():
    """Returns the XYZ position"""
    return get_pose()['Pose']['Position']

def pythagora_hypotenus(x, y):
    """Pythagoras theorem"""
    return sqrt((x ** 2) + (y ** 2))

def make_path():
    """Add all coordinates of the path to a stack"""
    stack = []
    with open(args.pat) as path_file:
        json_path = json.load(path_file)
        for i in range (len(json_path)):
            stack.append(json_path[i]['Pose']['Position'])
        stack.reverse()
        return stack

def get_point(path, pos, look_ahead, update_path = True):
    """Select the next goal point using the robot's position and a look-ahead distance

    Args:
        path (array of points): The path that the robot must follow
        pos: The actual position of the robot
        look_ahead (float): The look-ahead distance
        update_path (boolean): At true it deletes the past points, at false it doesn't

```

```
    Return the selected goal point
    """
    if path:
        for i in range(len(path)):
            point = path[len(path) - (1 if update_path else i)]
            dx = point['X'] - pos['X']
            dy = point['Y'] - pos['Y']

            dist = pythagora_hypotenus(dx, dy)

            if dist < look_ahead:
                if update_path:
                    path.pop()
            else:
                return point
    else:
        print ("Stack failed")

def pure_pursuit(robot_pos, point):
    """compute the angular speed that the robot must have to follow the path using the pure pursuit algorithm

    Args:
        robot_pos: the actual position of the robot
        point: the coordinates of the goal point

    Return the computed angular speed
    """
    dx = point['X'] - robot_pos['X']
    dy = point['Y'] - robot_pos['Y']
    dist = pythagora_hypotenus(dx, dy)

    robot_heading = get_heading()
    hx = robot_heading['X']
    hy = robot_heading['Y']
    robot_angle = atan2(hy, hx)
    point_angle = atan2(point['Y'] - robot_pos['Y'], point['X'] -  robot_pos['X'])
    teta = point_angle - robot_angle
    delta_x = sin(teta) / dist

    return (2 * delta_x) / (dist ** 2)

def our_algo(robot_pos, point):
    """compute the angular speed that the robot must have to follow the path using our own algorithm

    Args:
```

```
        robot_pos: the actual position of the robot
        point: the coordinates of the goal point

    Return the computed angular speed
    """
    robot_heading = get_heading()
    hx = robot_heading['X']
    hy = robot_heading['Y']
    robot_angle = atan2(hy, hx)
    point_angle = atan2(point['Y'] - robot_pos['Y'], point['X'] - robot_pos['X'])

    teta = sin(point_angle - robot_angle)
    x = 0.3
    max_ang_speed = 3

    ang_speed = (max_ang_speed * teta) / x

    return min(max(ang_speed, -max_ang_speed), max_ang_speed)

def run_algo(algo, pos, goal_point):
    """Run the algorithm that the user choose

    Args:
        algo: The parameter value that the user gave
        pos: The actual position of the robot
        goal_point: The coordinates of the goal point

    Return the angular speed returned by the corresponding algorithm
    """
    ang_speed = 0
    if algo == 1:
        ang_speed = pure_pursuit(pos, goal_point)
    else:
        ang_speed = our_algo(pos, goal_point)

    return ang_speed

def compute_linear_speed(func, ang_speed):
    """Compute the linear speed adapted to the angular speed, using the function that the user picked

    Args:
        func: The parameter value that the user gave
        ang_speed: The angular speed of the robot

    Return the computed linear speed
```

```
    """
    max_linear_speed = 1
    linear_speed = max_linear_speed

    if func == 2:
        linear_speed = min(max_linear_speed, 1 / log10(6 *
        abs(ang_speed) + 1))
    elif func == 3:
        linear_speed = min(max_linear_speed, log10(-(min(3,
        abs(ang_speed)) - 4.7)) + 0.5)
    elif func == 4:
        linear_speed = min(max_linear_speed, -0.2 * min(3,
        abs(ang_speed)) + 1.3)

    return linear_speed

if __name__ == '__main__':
    path = make_path()
    positions_x, positions_y, path_x, path_y = ([] for i in
                range(4))
    for point in path:
        path_x.append(point['X'])
        path_y.append(-point['Y'])

    angular_constant = 0.4
    look_ahead = 0.7

    algo = args.algo
    func = args.func

    start_time = time.time()
    while path:
        pos = get_position()
        positions_x.append(pos['X'])
        positions_y.append(-pos['Y'])
        goal_point = get_point(path, pos, look_ahead)
        if goal_point:
            ang_speed = run_algo(algo, pos, goal_point)
            linear_speed = compute_linear_speed(func,
                        ang_speed)
            if args.ahead:
                look_ahead = linear_speed

            response = post_speed(ang_speed *
                        angular_constant,
            linear_speed)
            time.sleep(0.01)
    response = post_speed(0,0)

    end_time = time.time()
```

```
    run_time = end_time - start_time

    if args.plot:
        plt.plot(positions_y, positions_x, 'r', label="
            Robot")
        plt.plot(path_y, path_x, 'b', label="Path")
        plt.legend(loc='upper left')
        plt.show()

    print("end of run")
    print("The robot finished the path in:", run_time, "
        seconds")
```