Fundamentals of Artificial Intelligence
Assignment 1 - Follow the path

UMU-57205
Cyril Bos (cybo0001)
Dorian Cuquemelle (docu0002)
29/09/2017

Teachers: Thomas Johansson,
Ola Ringdahl,
Lennart Jern,
Jonathan Westin

Contents

This report presents the implementation and the results of a way to solve the path tracking problem described in the course, involving a robot able to move in a room.

1 General Information

This section presents general information about the project. Most of them are also written in the `README.md` file in the source code. The source code is located in the src directory of the zip archive.
The algorithms given in this report are written in a Python-ish language. The source code is included at the very end (starting at page 13).

1.1 Dependencies

The assignment was developed in Python 3(.6) and uses only standard library packages.

1.2 Usage

The following command will launch the main script controlling the robot using the default values written at the top of the `main.py` file :

```
> python main.py
```

Options can be issued as arguments of the script instead of modifying the file:

```
> python main.py path=paths/Path-around-table.json --obstacle --level=DEBUG
```

- The `--obstacle` option enables the obstacle detection algorithm by instantiating an `ObstacleController` instead of a `FixedController`.
- The `--level` option enables a more or less precise logging in the terminal. Defaults to `INFO`. Level `DEBUG` will provide more information such as the current position of the robot while travelling.
- Level `ERROR` will provide logs only when an exception is encountered.

2 Path-tracking theoretical explanation

This part will explain the method we chose from a mathematical and algorithmic point of view.

2.1 Mathematical approach

The used algorithm is `Pure Pursuit`, as described in Chapter 4 of *Barton's thesis*. It was chosen over `Follow-the-carrot` because it follows most paths more precisely by producing lesser heading errors. (as explained in `"Path tracking for mobile robots"` slides).

The robot navigates along the path through a succession of circles defined between the current position of the robot and the next position the robot is aiming for on the path, tangent to the current speed vector of the robot.

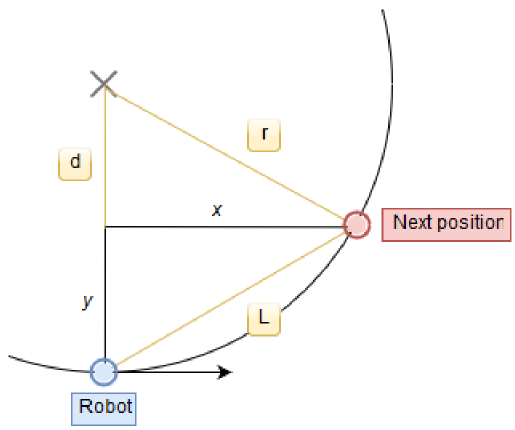

Figure 1: Pure Pursuit Principle

The angular speed ω can be deduced according the following formula :

$$\omega = \frac{2\pi}{T} = \frac{2\pi r}{Tr} = \frac{v}{r}$$

Given a constant linear speed V along this circle. R is the radius of the circle, and T is the period to perform a turn on the circle.

The robot needs both the linear and angular speed to move. We then need to find a way to express the radius with the robot's datas.
Using the Pythagorean theorem

$$x^2 + y^2 = L^2$$

$$d^2 + x^2 = r^2$$

We also have $y + d = r$.

Thus,

$$d = r - y$$

$$(r - y)^2 + x^2 = r^2$$

$$-2ry + y^2 + x^2 = 0$$

$$L^2 = 2ry$$

$$r = \frac{L^2}{2y}$$

Both L is (the Cartesian distance between the points) and Y can be computed, therefore the robot has every information needed to calculate the angular speed from its linear speed.

2.2 Two coordinate systems

Two different coordinate systems are used in the scripts:

- The World Coordinate System (WCS), used by the server for the 3D/physics engine
- The Robot Coordinate System (RCS), which is centered on the center of the robot

The coordinates given by the MRDS requests and the path files are expressed in the WCS. Yet, `Pure Pursuit` needs those coordinates relative to the RCS. Additionally, the angles given by the laser scan are expressed in the RCS, based on its current rotation in space. It is the needed to be able to convert the positions from one system to the other.
Since MRDS is able to provide the position of the robot and orientation of the robot, it is possible to calculate the heading vector of the robot starting from its center.

The rotation of the robot in space is given by a quaternion `Q`. Since conjugates of quaternions and rotation in space are equivalent, we can get the heading $\vec{h}$ like this :

$$\vec{h} = q.\ \vec{x}\ q^{-1}$$

with q^{-1} the conjugate of the robot's quaternion.

Then with trigonometry, $\alpha = atan(\frac{\vec{h}.y}{\vec{h}.x})$, angle between the heading of the robot (y axe in RCS) and the y axe in WCS.
Finally, using the rotation matrix to get the coordinates of a given point at $(x', y')_{RCS}$ from a point at $(x, y)_{WCS}$:

$$\begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} cos(\alpha) & -sin(\alpha) \\ sin(\alpha) & cos(\alpha) \end{bmatrix} \begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} x'\ cos(\alpha) - y'\ sin(\alpha) \\ x'\ sin(\alpha) + y'\ cos(\alpha) \end{bmatrix}$$

Given the position of the robot in (x_0, y_0) in WCS, we get :

$$(x - x_0) = x' cos(\alpha) - y' sin(\alpha)$$

$$(y - y_0) = x' sin(\alpha) + y' cos(\alpha)$$

Thus,

$$cos(\alpha)(x - x_0) + sin(\alpha)(y - y_0) = x'(cos^2(\alpha) + sin^2(\alpha)) - y' sin(\alpha)cos(\alpha) + y' cos(\alpha)sin(\alpha) = x'$$

$$cos(\alpha)(y - y_0) - sin(\alpha)(x - x_0) = x' sin(\alpha)cos(\alpha) - x' sin(\alpha)cos(\alpha) + y'(cos^2(\alpha) + sin^2(\alpha)) = y'$$

2.3 Algorithmic approach

Since it is possible to compute the speed from one point to another, the most basic path tracking algorithm would be to take every position on the path and compute those speeds to make very small rotating movements.
The linear speed of the robot is constant. Bigger values of it produce a bigger error, i.e the robot will derive further away from the tracked path. One of the implementation goals is to maximize this linear speed while keeping the distance to the track error acceptable.

```
def pure_pursuit(linear_speed: float):
    for target in path:
        position = robot_get_position(), orientation = robot_get_orientation()

        #Commmands the robot to apply the calculated angular_speed and linear_speed to its wheels until it reaches target
        robot_travel(target, linear_speed, calculate_angular_speed(target))

    robot_stop()

def calculate_angular_speed(target: Vector, linear_speed: float): -> float
    rcs_target = convert_to_rcs(target)
```

```
    L2 = ((pow(rcs_target.x, 2) + pow(rcs_target.y, 2))
    radius = L2 / (2 * rcs_target.y)
    return linear_speed / radius

def convert_to_rcs(position: Vector): -> Vector
    robot = robot_get_position()
     q = Quaternion(orientation.w, Vector(0, 0, orientation.z)).heading()
    angle = atan2(q.y, q.x)

    rcs_position = Vector()
    rcs_position.x = (position.x - robot.x) * cos(angle) + (position.y - robot.y) * sin(angle)
    rcs_position.y = -(position.x - robot.x) * sin(angle) + (position.y - robot.y) * cos(angle)
    return rcs_position

def robot_travel(target: Vector, linear: float, angular: float):
    position = robot_get_position()
    robot_apply_speed(linear, angular)
    while cartesian_distance(target, robot_get_position()) < DELTA:
        pass
```

The `DELTA` constant in `robot_travel()` can be equal up to 1 (as precised in the assignment instructions). A lower value means that the robot will follow the path more precisely. However a too small value would cause the robot to never reach the target and turn in a circle indefinitely.

3 Implementation and optimization

3.1 Code structure

The code is divided in two packages:

- `controller`, which contains the `Controller` super class, and the `FixedController` and `ObstacleController` subclasses. `Controller` offers methods to interact with the MRDS server (POST request for speed and GET requests of various information) and a travel method that monitors the movement of the robot till it reaches a position.
- `model`, which contains the Vector and Quaternion classes, and a pure_pursuit module. These two classes implement various methods needed to implement the path tracking algorithm.

Our implementation of quaternion related methods are tested against the ones provided by `lokarriaexample.py` in the module texttunit_tests.py. These files are located at the root of the project. The file `lokarriaexample.py` has been translated to Python 3 by the `2to3` linux command.
The main file contains several constants definitions, most of them can be redefined with script arguments (implemented using the standard library package `argparse`). A dictionary named `PARAMETERS` contains optimized parameters for both controllers for the paths in the `paths/` directory. When testing another path, it will use safer default values. It recognizes the paths by their filename.

3.2 Path optimization

A basic idea of path optimization using the pure pursuit algorithm is to increase L. In our case, this can be simply done by increasing the loop to more than one. We implemented this in the `FixedController` class.

```python
    def pure_pursuit(self, pos_path):
        # Travel through the path skipping "lookahead" positions every time
        for i in range(0, len(pos_path), self.__lookahead):
            cur_pos, cur_rot = self.get_pos_and_orientation()
            self.travel(cur_pos, pos_path[i], self._lin_spd,
                        pure_pursuit.get_ang_spd(cur_pos, cur_rot, pos_path[i],
                            self._lin_spd))

        self.stop()
```

Another similar idea is to fix the lookahead not in a fixed number of positions to skip on the path, but by a distance (which value depends on the simulation engine). This method would lead to better results if the distance between each position in the encoded path varies. Since we already got good results using a lookahead equal to a fixed number of positions, we instead tried to implement an improved version using the laser scan to detect obstacles.

3.2.1 Obstacle detection algorithm

The algorithm we implemented consists in testing if there is an obstacle between the current position of the robot and every position in the path. The robot will aim for the last position that is not blocked by an obstacle.
To detect this obstacle, the distance to the target is compared to the distance of the corresponding nearest laser. If the distance to the target is bigger, there is no obstacle between the current position and the targeted position. The following figure 2 illustrates the idea of the algorithm.

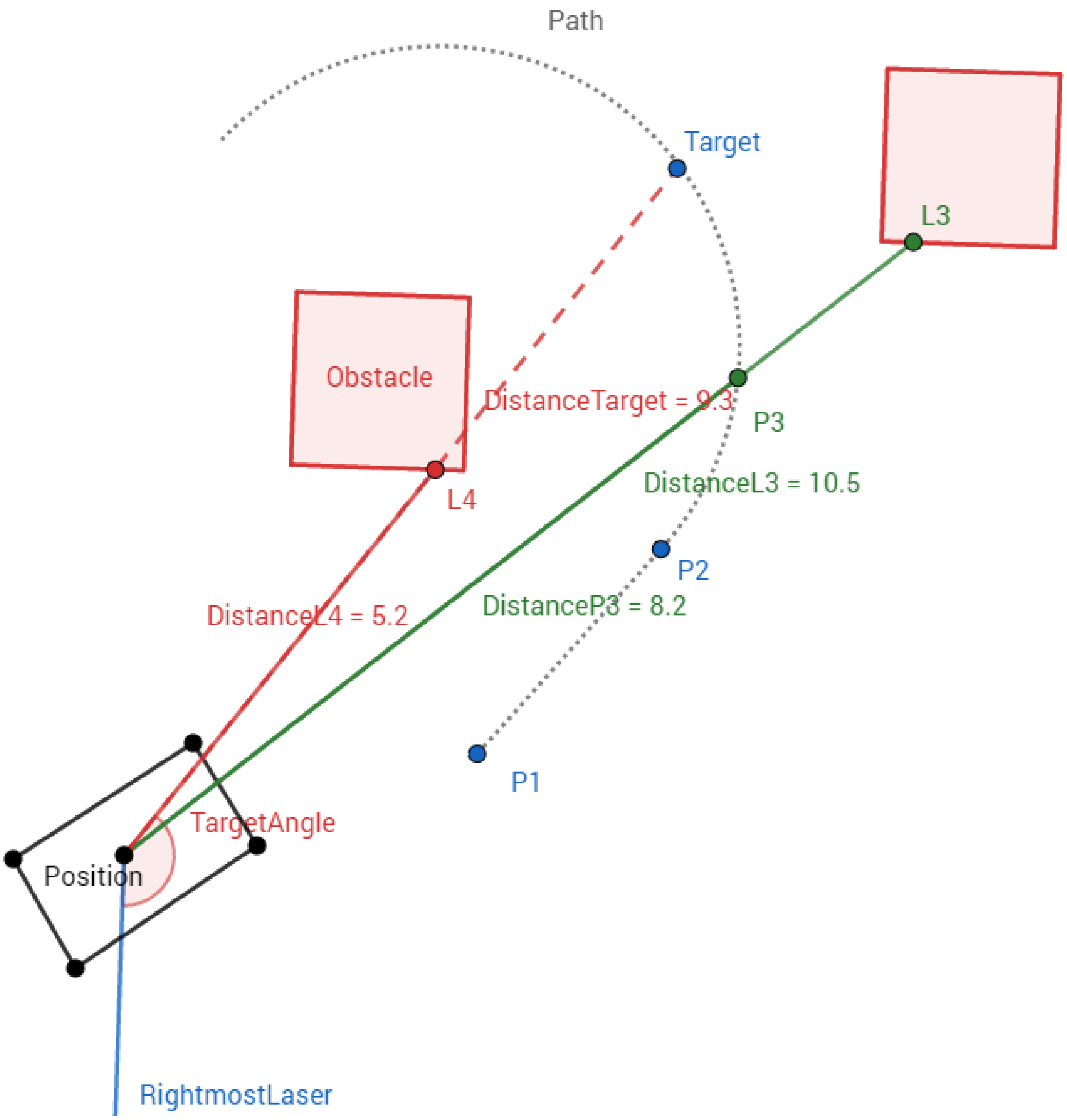

Figure 2: Visualisation of the obstacle detection process

From an algorithmic perspective, it could be written as:

```
def obstacle_pure_pursuit(path: List[Vector], linear_speed: float)

    i = 0
    while i < length(path):
        i = target_index_before_obstacle(robot_get_position(), path, i)
        target = path[i]
        robot_travel(target, linear_speed, get_angular_speed(target,
            linear_speed))


def target_index_before_obstacle(position: Vector, path: List[Vector],
```

```
    position_index: int): -> int
    lasers = robot_get_lasers()
    for i from position_index + 1 to length(path):
        rcs_target = convert_to_rcs(path[i])

        # current robot position in RCS
        rcs_origin = Vector(0, 0, 0)

        # compute angle between current robot position and aimed position
        target_angle = get_angle(rcs_origin, rcs_target)

        #get the laser which angle is the closest
        nearest_laser = get_nearest_laser(target_angle)

        # if the laser hit an obstacle
        if nearest_laser.distance < cartesian_distance(position, target):
            if i == cur_pos_index + 1:
                prevent_collision() #if first position failed, the robot could stop and rotate to prevent a collision
            return i - 1 #return the index of the last position that succeeded the test

    return max_index

def get_nearest_laser(lasers: List[Laser], angle: float): -> laser
    dist = angle - lasers[0]
    min_index = 0
    for i from 1 to length(lasers):
        dist = angle - lasers[i].angle
        min = dist if dist < min
        min_index = i

    return lasers[min_index]
```

We implemented this algorithm in the `ObstacleController` class, adding a texttttmax_lookahead parameter that will prevent the algorithm from aiming too far. The reason for this is that the pure pursuit algorithm works better for values of L (the distance to next target position) that are sufficiently high, but not too high (as shown in the last slide of "Path tracking for mobile robots").

4 Results, problems encountered and potential developments

The `FixedController` only works for small values of lookahead, equal to 5 with a linear speed of 1 for every path in directory `paths/` except 20 for the Path-around-table-and-back. These values also depend on the other two parameters values, and if the tracked paths use different distance between points it won't work as well as with a lookahead expressed in a world distance instead.

It is not possible to give accurate times using this simulation because of random frame drops that "slow down" the robot compared to the measured time. The obstacle detection still gives better times (same values of linear speed as the fixed version, but the lookahead is greatly improved).

If the path goes very close to an obstacle, the single laser may not detect a possible collision as shown on the figure 3 below.

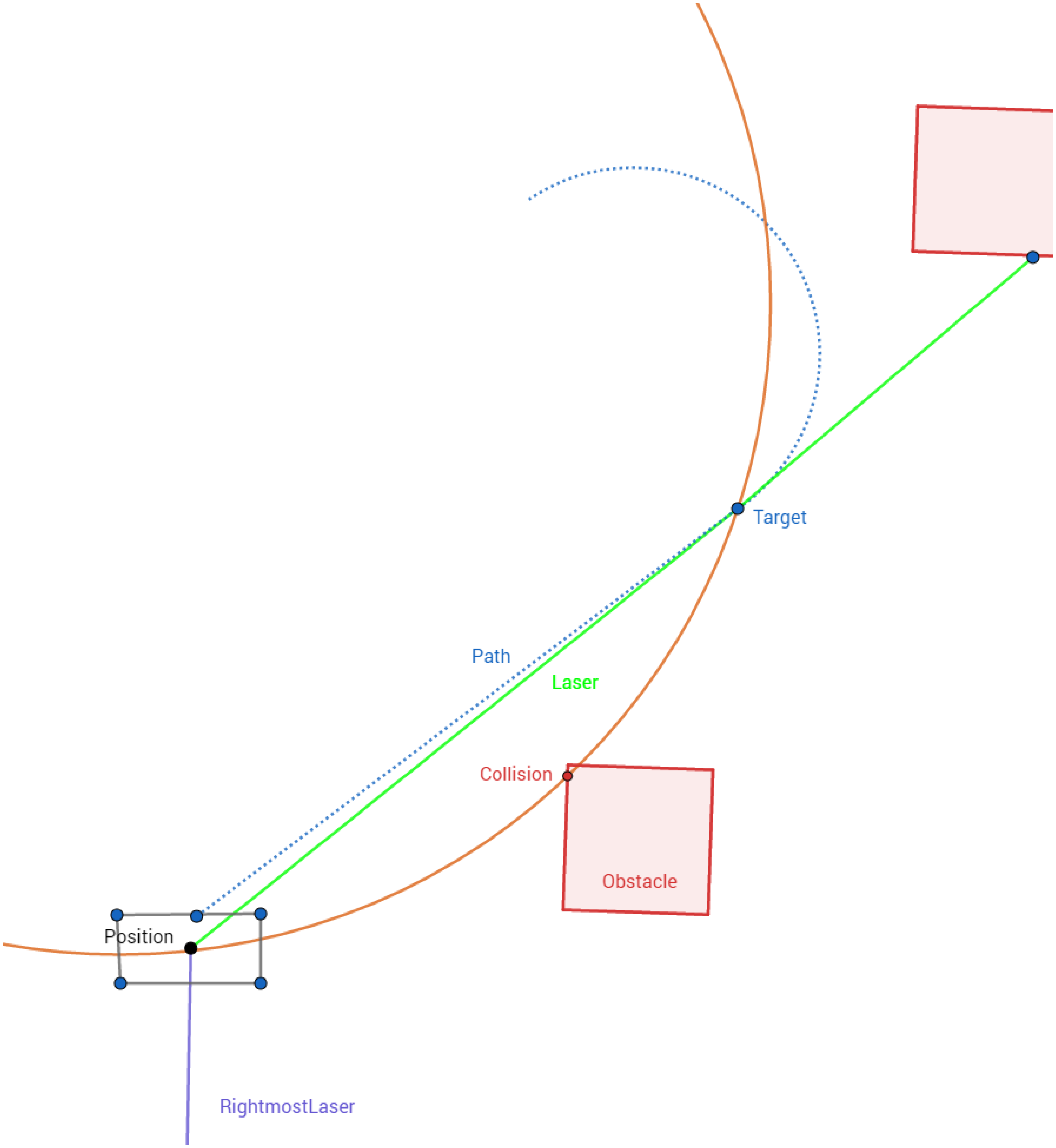

Figure 3: Visualization of a critical obstacle detection failure

By using a whole cone of lasers around Target, which angle would be computed by projecting the diagonal of the robot (or width, but the diagonal is safer because it is not possible to precisely virtualize the orientation of the robot on the calculated curve). This is illustrated by the figure 4 below.

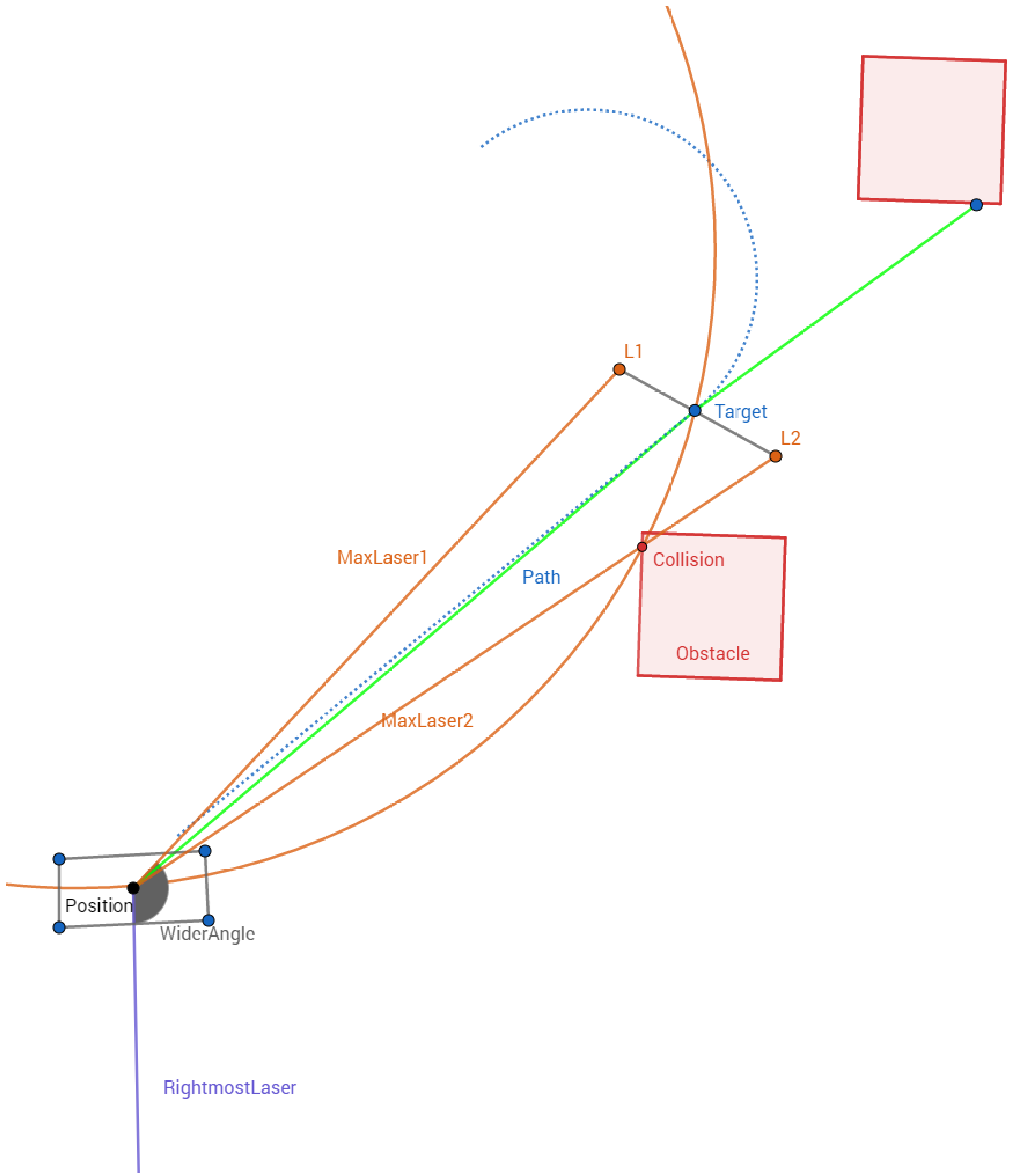

Figure 4: Visualization of an improved method for obstacle detection

We did not implement this because we were unable to find how to get the width and length of the robot in the MRDS/RobuBox documentation.

It could also be possible to detect some patterns in the paths, like straight lines (computing angle difference between every point until it differs too much). This way, the robot could totally rely on a single laser and cut through an "useless" curve, and help our version of the obstacle detection fix its bug. We were warned it is not authorized to do this, so we did not.

Another issue happens when the robot reaches a sub-target. There is a short duration where a possibly wrong angular speed is still applied to the wheels, until the next target is computed. This error is minimal and did not impact much our tests but is still present.

The use of multithreading (and even better multiprocessing) would reduce errors by determining the next target while travelling to the current target.

It was also cumbersome to have to relaunch the program to reset the position of the robot and the scene layout, especially when debugging but also when trying to find optimized parameters. The program also bugs out sometimes for some reason and fails to respond to the GET requests correctly, causing the tracking to fail. However, it rarely happened.

5 Source code

This section includes the source code of the latest revision of the code used during the presentation.

5.1 Controller package

```python
import http.client, json
from logging import getLogger
from math import atan2, cos, sin, pow, sqrt, pi
from time import sleep

from model import Vector, Quaternion
from model import pure_pursuit

logger = getLogger('controller')

# Headers sent with every POST speed requests
HEADERS = {"Content-type": "application/json", "Accept": "text/json"}


class Controller:
    """
    Controller base class which contains methods to send requests to the MRDS server and the travel monitoring
    routine.
    """

    class UnexpectedResponse(Exception):
        """
        Custom exception class raised when a HTTP request fails.
        """
        pass

    def __init__(self, mrds_url, lin_spd=1, delta_pos=0.75):
        """
        Initializes a new instance of Controller.
        :param mrds_url: url which the MRDS server listens on
        :type mrds_url: str
        :param lin_spd:
        :type lin_spd: float
        :param delta_pos:
        :type delta_pos: float

        """
        self.__mrds = http.client.HTTPConnection(mrds_url)
        self._lin_spd = lin_spd
        self._delta_pos = delta_pos

    def post_speed(self, angular_speed, linear_speed):
        """
        Sends a speed command to the MRDS server.

        :param angular_speed: value of angular speed
        :type angular_speed: float
        :param linear_speed: value of linear speed
        :type linear_speed: float
```

```
        """
        params = json.dumps({'TargetAngularSpeed': angular_speed, 'TargetLinearSpeed': linear_speed})
        self.__mrds.request('POST', '/lokarria/differentialdrive', params, HEADERS)
        response = self.__mrds.getresponse()
        status = response.status
        response.close()
        if status == 204:
            return response
        else:
            raise self.UnexpectedResponse(response)

    def get_pos(self):
        """
        Reads the current position from the MRDS and returns it as a Vector instance.
        """
        self.__mrds.request('GET', '/lokarria/localization')
        response = self.__mrds.getresponse()
        if response.status == 200:
            pos_data = json.loads(response.read())
            response.close()
            return Vector.from_dict(pos_data['Pose']['Position'])
        else:
            raise self.UnexpectedResponse(response)

    def get_pos_and_orientation(self):
        """
        Reads the current position and orientation from the MRDS server and returns it as a tuple (Vector, Quaternion).
        """
        self.__mrds.request('GET', '/lokarria/localization')
        response = self.__mrds.getresponse()
        if response.status == 200:
            pos_data = json.loads(response.read())
            response.close()
            return Vector.from_dict(pos_data['Pose']['Position']), Quaternion.from_dict(pos_data['Pose']['Orientation'])

        else:
            raise self.UnexpectedResponse(response)

    def get_laser_scan(self):
        """
        Requests the current laser scan from the MRDS server and parses it into a dict.
        """
        self.__mrds.request('GET', '/lokarria/laser/echoes')
        response = self.__mrds.getresponse()
        if response.status == 200:
            laser_data = response.read()
            response.close()
            return json.loads(laser_data)
        else:
            return self.UnexpectedResponse(response)
```

```
    def get_laser_scan_angles(self):
        """
        Requests the current laser properties from the MRDS server and returns a list of the laser angles.
        """
        self.__mrds.request('GET', '/lokarria/laser/properties')
        response = self.__mrds.getresponse()
        if response.status == 200:
            laser_data = response.read()
            response.close()
            properties = json.loads(laser_data)
            beamCount = int((properties['EndAngle'] - properties['StartAngle']) / properties['AngleIncrement'])
            a = properties['StartAngle']  # +properties['AngleIncrement']
            angles = []
            while a <= properties['EndAngle']:
                angles.append(a)
                a += pi / 180  # properties['AngleIncrement']
            # angles.append(properties['EndAngle']-properties['AngleIncrement']/2)
            return angles
        else:
            raise self.UnexpectedResponse(response)

    def travel(self, cur_pos, tar_pos, lin_spd, ang_spd):
        """
        Routine to travel to targeted position at given linear and angular speeds until close enough
        :param cur_pos: current position of the robot
        :type cur_pos: Vector
        :param tar_pos: targeted position to travel to
        :type tar_pos: Vector
        :param lin_spd: linear speed at which the robot should travel
        :type lin_spd: float
        :param ang_spd: angular speed at which the robot should travel
        :type ang_spd: float
        :param delta_pos: value which is the distance between current position.
                          Defaults to 1 (as precised in the assignment subject)
        :type delta_pos: float

        """
        logger.debug(
            'Traveling from {} to {}\n with linear speed={} and angular speed={}'.format(cur_pos, tar_pos, lin_spd,
                                                                                                                  a
        slp_dur = self._delta_pos / (lin_spd * 1000)  # unnecessary to monitor cur_pos more than this
        response = self.post_speed(ang_spd, lin_spd)
        sleep(slp_dur)
        try:
            while cur_pos.distance_to(tar_pos) > self._delta_pos:
                cur_pos = self.get_pos()
                logger.debug('[travel()] current position: {}'.format(cur_pos))
                sleep(slp_dur)

```

```
        except self.UnexpectedResponse as ex:
            print('Unexpected response from server when sending speed
               commands:', ex)

    def stop(self):
        self.post_speed(0, 0)

    def u_turn(self):
        self.stop()
        self.post_speed(-1, 0)
        sleep(1)
        self.stop()
```

```
from logging import getLogger

from .controller import Controller
from model import pure_pursuit

logger = getLogger('controller')


class FixedController(Controller):
    """
    Class that inherits from Controller and implements pure pursuit with a
       fixed lookahead.
    """

    def __init__(self, mrds_url, lin_spd=1, lookahead=5, delta_pos=0.75):
        """
        Initializes a new FixedController instance.
        :param mrds_url: url which the MRDS server listens on
        :type mrds_url: str
        :param lin_spd:
        :type lin_spd: float
        :param lookahead: fixed number of positions to skip on the path
        :type lookahead: int
        :param delta_pos: "close enough" distance. Minimum distance from the
           target position at which the robot
        considers it reached that position
        :type delta_pos: float

        """
        super(FixedController, self).__init__(mrds_url, lin_spd=lin_spd,
           delta_pos=delta_pos)
        self.__lookahead = lookahead
        logger.info(
            'Using {} with linear speed={}, lookahead={}, delta
               position={}'.format(self.__class__.__name__, lin_spd,
                                                                                                   lookah
                                                                                                      del

    def pure_pursuit(self, pos_path):
        """
        Implements the pure pursuit algorithm with a fixed lookahead. The robot
           aims for "self.__lookahead"
        positions ahead on the given path.
```

```python

        :param pos_path: list of Vector
        :type pos_path: list
        """
        # Travel through the path skipping "lookahead" positions every time
        for i in range(0, len(pos_path), self.__lookahead):
            cur_pos, cur_rot = self.get_pos_and_orientation()
            self.travel(cur_pos, pos_path[i], self._lin_spd,
                        pure_pursuit.get_ang_spd(cur_pos, cur_rot, pos_path[i],
                            self._lin_spd))

        self.stop()
```

```python
from logging import getLogger
from time import sleep

from .controller import Controller
from model import Quaternion, Vector, pure_pursuit

logger = getLogger('controller')


class ObstacleController(Controller):
    """
    Class that inherits from Controller and optimizes pure pursuit using
        obstacle detection.
    """

    def __init__(self, mrds_url, lin_spd, max_lookahead, delta_pos):
        """
        Initializes a new FixedController instance.
        :param mrds_url: url which the MRDS server listens on
        :type mrds_url: str
        :param lin_spd:
        :type lin_spd: float
        :param max_lookahead: maximum number of positions to skip on the path
        :type max_lookahead: int
        :param delta_pos: "close enough" distance. Minimum distance from the
            target position at which the robot
        considers it reached that position
        :type delta_pos: float

        """
        super(ObstacleController, self).__init__(mrds_url, lin_spd, delta_pos)
        self.__max_lookahead = max_lookahead
        logger.info(
            'Using {} with linear speed={}, max_lookahead={}, delta
                position={}'.format(self.__class__.__name__, lin_spd,
                                                                                  max_lo
                                                                                    del


    def pure_pursuit(self, pos_path):
        """
        Implements the pure pursuit algorithm using obstacle detection to aim
            for the furthest position possible.

```

```
        :param pos_path: path loaded into Vector
        :type pos_path: list
        """
        pos_index = -1
        last_pos_index = len(pos_path) - 1
        while pos_index < last_pos_index:
            cur_pos, cur_rot = self.get_pos_and_orientation()

            # aim for the position before the one that would cause a collision
            new_pos_index = self.next_optimized_waypoint(cur_pos, cur_rot, pos_path, pos_index)
            if new_pos_index == pos_index:
                pos_index = new_pos_index + 1
            else:
                pos_index = new_pos_index

            logger.info("Target position path index: {}".format(new_pos_index))

            self.travel(cur_pos, pos_path[pos_index], self._lin_spd,
                        pure_pursuit.get_ang_spd(cur_pos, cur_rot, pos_path[pos_index], self._lin_spd))


        self.stop()

    def next_optimized_waypoint(self, cur_pos, cur_rot, pos_path, cur_pos_index):
        """
        Returns the furthest point from path without an obstacle. Stops at the first position where the laser of
        nearest angle (laser angle ~= aimed position angle) detects an obstacle (laser distance < aimed position
        distance).
        :param cur_pos: current position of the robot
        :type cur_pos: Vector
        :param cur_rot: current orientation of the robot
        :type cur_rot: Quaternion
        :param pos_path: loaded path
        :type pos_path: list
        :param cur_pos_index: index of the current position in the path (-1
        :type cur_pos_index: int

        """
        lasers_angles = self.get_laser_scan_angles()
        lasers = self.get_laser_scan()['Echoes']

        max_lookahead_index = min(cur_pos_index + self.__max_lookahead - 1, len(pos_path) - 1)
        # Go through every position on the path starting at the position next to the current one and stopping at
        # max_lookahead_index positions further
        for i in range(cur_pos_index + 1, max_lookahead_index):
            tar_pos = pos_path[i]
            # convert potential aimed position to RCS
            rcs_tar_pos = pure_pursuit.convert_to_rcs(tar_pos, cur_pos, cur_rot)

```

```python
            # current robot position in RCS
            rcs_origin = Vector(0, 0, 0)
            # compute angle between current robot position and aimed position
            tar_angle = rcs_origin.get_angle(rcs_tar_pos)

            min_ind = 0
            min_dist = tar_angle - lasers_angles[0]
            # search for the nearest angle in laser_angles
            # could be simplified with a calculation instead of this iteration
            for j in range(1, len(lasers_angles)):
                dist = tar_angle - lasers_angles[j]
                if dist < min_dist:
                    min_dist = dist
                    min_ind = j

            # if the laser hits an obstacle, return the index of the previous
                position on the path
            if lasers[min_ind] < cur_pos.distance_to(tar_pos):
                if i == cur_pos_index + 1:
                    # if first position fails, the robot could stop and rotate
                    # does not work because of lack of laser precision (should
                        use a whole cone instead of just one)
                    pass
                return i - 1
        return max_lookahead_index
```

```python
import json
from model import Vector, Quaternion


class PathLoader:
    """
    Class that loads the path files into lists of Vector used by our program or
        dictionaries used by the unit tests.
    """

    def __init__(self, filename):
        # Load the path from a file and convert it into a list of coordinates
        self.loadPath(filename)
        self.vecPath = self.positionPath(dict=True)

    def loadPath(self, file_name):
        with open(file_name) as path_file:
            data = json.load(path_file)

        self.path = data

    def positionPath(self, dict=False):
        """
        Parses the positions in the loaded file into a list of either Vector
            instances or dictionaries depending on the
        value of the dict parameter.
        :param dict: if set to True, will parse into dictionaries instead of
            into Vector instances
        :type dict: bool
        """
```

```
        if dict:
            return [{'X': p['Pose']['Position']['X'],
                     'Y': p['Pose']['Position']['Y'],
                     'Z': p['Pose']['Position']['Z']}
                    for p in self.path]
        else:
            return [Vector.from_dict(p['Pose']['Position'])
                    for p in self.path]

    def orientationPath(self, dict=False):
        """
        Parses the orientations in the loaded file into a list of either
            Quaternion instances or dictionaries depending on the
        value of the dict parameter.
        :param dict: if set to True, will parse into dictionaries instead of
            into Quaternion instances
        :type dict: bool
        """
        if dict:
            return [{'W': p['Pose']['Orientation']['W'],
                     'X': p['Pose']['Orientation']['X'],
                     'Y': p['Pose']['Orientation']['Y'],
                     'Z': p['Pose']['Orientation']['Z']} for p in self.path]
        else:
            return [Quaternion(p['Pose']['Orientation']['W'],
                               Vector(p['Pose']['Orientation']['X'],
                                      p['Pose']['Orientation']['Y'],
                                      p['Pose']['Orientation']['Z']))
                    for p in self.path]
```

5.2 Model package

```
from math import atan2, sqrt, pow, cos, sin


class Vector:
    """
    Class that represents vectors with 3 float numbers.
    Implements useful functions to use these objects
    """

    def __init__(self, x, y, z):
        """
        Initialization of the Vector class given 3 floats
        :param x: x position
        :type x: float
        :param y: y position
        :type y: float
        :param z: z postion
        :type z: float
        """
        self._x = x
        self._y = y
        self._z = z

    def __eq__(self, other):
```

```
        """
        Returns whether the Vector object and another are equal or not
        :param other: the other vector for comparison
        :type other: Vector
        """
        return isinstance(other, Vector) and self.x == other.x and self.y == other.y and self.z == other.z

    def __str__(self):
        """
        Computes a string giving information about the vector
        """
        return "Vector<[{},{},{}]>".format(self.x, self.y, self.z)

    @property
    def x(self):
        return self._x

    @x.setter
    def x(self, value):
        self._x = value

    @property
    def y(self):
        return self._y

    @y.setter
    def y(self, value):
        self._y = value

    @property
    def z(self):
        return self._z

    @z.setter
    def z(self, value):
        self._z = value

    @staticmethod
    def from_dict(vec_dict):
        """
        Returns a new Vector instance from the dict representation used in the path files
        """
        return Vector(vec_dict['X'], vec_dict['Y'], vec_dict['Z'])

    @staticmethod
    def x_forward():
        """
        Returns the x-axis unit vector
        """
        return Vector(1.0, 0.0, 0.0)

    def get_angle(self, vec):
        """
        Returns the angle between self and another Vector
```

```
        :param vec: the other Vector
        :type vec: Vector
        """
        return atan2(vec.x - self.x, vec.y - self.y)

    def distance_to(self, vec):
        """
        Returns the angle between self and another Vector
        :param vec: the other Vector
        :type vec: Vector
        """
        return sqrt(pow(vec.x - self.x, 2) + pow(vec.y - self.y, 2) + pow(vec.z - self.z, 2))
```

```
import numpy

from .vector import Vector


class Quaternion:
    """
    Class that represents quaternions with 4 float numbers.
    Implements useful functions such as heading, rotations...
    """
    __w = 0

    def __init__(self, w, vector):
        """
        Initialization of the Quaternion class given a float and a Vector
        :param w: value of w to assign
        :type w: float
        :param vector: unit vector to assign
        :type vector: Vector
        """
        if not isinstance(vector, Vector):
            raise (TypeError('Parameter vector of init is not a vector'))
        self.__unit_vector = vector
        self.__w = w

    def __mul__(self, other):
        """
        multiplies the Quaternion with another, in that order
        :param other: the other quaternion for operation
        :type other: Quaternion
        """
        if isinstance(other, Quaternion):
            return Quaternion(self.w * other.w - self.x * other.x - self.y * other.y - self.z * other.z,
                              Vector(self.__w * other.x + self.x * other.w + self.y * other.z - self.z * other.y,
                                     self.__w * other.y - self.x * other.z + self.y * other.w + self.z * other.x,
                                     self.__w * other.z + self.x * other.y - self.y * other.x + self.z * other.w))
        else:
            raise TypeError('trying to multiply a quaternion by something else
```

```
                than a quaternion')

    def __str__(self):
        """
        Computes a string giving information about the quaternion
        """
        return "Quaternion<{}, [{},{},{}]>".format(self.w, self.x, self.y,
            self.z)

    @property
    def unit_vector(self):
        return self.__unit_vector

    @property
    def x(self):
        return self.__unit_vector.x

    @property
    def y(self):
        return self.__unit_vector.y

    @property
    def z(self):
        return self.__unit_vector.z

    @property
    def w(self):
        return self.__w

    @staticmethod
    def from_dict(dict):
        """
            Returns a new instance of Quaternion from the dict representation
                used in the path files.
        """
        return Quaternion(dict['W'], Vector(dict['X'], dict['Y'], dict['Z']))

    def normalize(self):
        """
        Returns the norm of quaternion
        """
        return (self.__w + self.x + self.y + self.z) / \
               numpy.sqrt(self.__w * self.__w + self.x * self.x + self.y *
                   self.y + self.z * self.z)

    def conjugate(self):
        """
        Returns the conjugate of quaternion
        """
        return Quaternion(self.w, Vector(-self.x, -self.y, -self.z))

    def heading(self):
        """
        Returns the heading of this quaternion from the X-axis
        """
        return self.rotate(Vector.x_forward())
```

```

    def rotate(self, v):
        """
        Returns the vector v rotated by the quaternion in a new Vector instance
        :param v: vector to rotate
        :type v: Vector
        """
        rotated = (self * Quaternion(0, Vector(v.x, v.y, v.z))) *
            self.conjugate()
        return Vector(rotated.x, rotated.y, rotated.z)
```

```
from math import atan2, cos, sin

from .vector import Vector
from .quaternion import Quaternion


def convert_to_rcs(tar_pos, cur_pos, cur_rot):
    """
    Computes the targeted position from the world coordinate system to the
        robot coordinate system
    :param tar_pos: targeted position to travel to
    :type tar_pos: Vector
    :param cur_pos: current position of the robot
    :type cur_pos: Vector
    :param cur_rot: current rotation of the robot
    :type cur_rot: Quaternion
    """
    q = Quaternion(cur_rot.w, Vector(0, 0, cur_rot.z)).heading()
    angle = atan2(q.y, q.x)
    rcs_pos = Vector(0, 0, tar_pos.z)

    rcs_pos.x = (tar_pos.x - cur_pos.x) * cos(angle) + (tar_pos.y - cur_pos.y)
        * sin(angle)
    rcs_pos.y = -(tar_pos.x - cur_pos.x) * sin(angle) + (tar_pos.y - cur_pos.y)
        * cos(angle)

    return rcs_pos

def get_ang_spd(cur_pos, cur_rot, tar_pos, lin_spd):
    """
    Computes the angular speed using pure pursuit formulas
    :param cur_pos: current position of the robot
    :type cur_pos: Vector
    :param cur_rot: current rotation of the robot
    :type cur_rot: Quaternion
    :param tar_pos: targeted position to travel to
    :type tar_pos: Vector
    :param lin_spd: linear speed of the robot
    :type lin_spd: float
    """
    rcs_tar_pos = convert_to_rcs(tar_pos, cur_pos, cur_rot)

    ang_spd = lin_spd / ((pow(rcs_tar_pos.x, 2) + pow(rcs_tar_pos.y, 2)) / (2 *
        rcs_tar_pos.y))

```

```
    return ang_spd
```

5.3 Main and unit tests

```
import logging
import argparse
import time

from model import Vector, Quaternion
from controller import FixedController, ObstacleController, PathLoader

# url which the MRDS server listens on
mrds_url = 'localhost:50000'

# filename of the path to load. Can be set by appending option --path=filename to this script
path_filepath = 'paths/Path-around-table-and-back.json'

# can be set to True by appending argument --obstacle to this script
# if set to True, instead of a fixed lookahead it will try to optimize as much as possible by detecting obstacles
# if set to False, the controller will skip positions with a fixed lookahead independent of the obstacle detection
obstacle_detection = False

# Optimized parameters for each path
# the lists respect the format [lin_spd, lookahead, delta_pos] when using fixed lookahead
# otherwise for the obstacle detection the format is [linear_speed, max_lookahead, delta_pos]
# lin_spd (linear speed) and delta_pos parameters are described in Controller.__init__()
# lookahead is described in FixedController.__init__() and used in FixedController.pure_pursuit()
# max_lookahead is described in ObstacleController.__init__() and used in ObstacleController.next_optimized_waypoint()
# if using another filename, will use the default values of Controller.__init__
PARAMETERS = {
    'fixed': {
        'Path-around-table-and-back': [1.5, 10, 0.75],
        'Path-around-table': [1, 5, 0.75],
        'Path-to-bed': [1, 5, 0.75],
        'Path-from-bed': [1, 5, 1],
    },
    'obstacle': {
        'Path-around-table-and-back': [1.5, 30, 0.75],
        'Path-around-table': [1, 60, 1],
        'Path-to-bed': [1, 8, 0.75],
        'Path-from-bed': [1, 10, 1],
        'exam2017': [1, 50, 0.75]
    }
}

# default values for Fixed
FIXED_DEFAULT_LIN_SPD = 1
FIXED_DEFAULT_LOOKAHEAD = 5
```

```python
FIXED_DEFAULT_DELTA_POS = 0.75

# default values for ObstacleController
OBSTACLE_DEFAULT_LIN_SPD = 0.75
OBSTACLE_DEFAULT_MAX_LOOKAHEAD = 5
OBSTACLE_DEFAULT_DELTA_POS = 0.75

#Used for logging level option
LOG_LEVEL_STRINGS = ['CRITICAL', 'ERROR', 'WARNING', 'INFO', 'DEBUG']
logging.basicConfig(level=logging.INFO)

if __name__ == '__main__':
    # Parsing arguments for script options
    parser = argparse.ArgumentParser()
    parser.add_argument('--path', type=str, help='filename of the path to load')
    parser.add_argument('--obstacle', action='store_true', default=False,
                        help='use obstacle detection to optimize the path')
    parser.add_argument('--level', type=str, help='Python logger level (ERROR, INFO, DEBUG). Defaults to info. \
                                                    Setting to debug will provide more information such as current position \
                                                     of the robot (among others). \
                                                    Setting to error will provide less information (only exception catching). ')

    args = parser.parse_args()

    logger = logging.getLogger(__name__)
    # Setting up depending on options values
    if args.obstacle:
        obstacle_detection = True
    if args.path:
        path_filepath = args.path
    if args.level:
        logger.setLevel(args.level)
        logging.getLogger('controller').setLevel(LOG_LEVEL_STRINGS.index(args.level))

    # if not placed in same folder, parses the filename of the filepath of the path
    # used to get the optimized parameters for each path
    if '/' in path_filepath:
        path_name = path_filepath[path_filepath.rindex('/') + 1:path_filepath.rindex('.')]
    elif '\\' in path_filepath:
        path_name = path_filepath[path_filepath.rindex('\\') + 1:path_filepath.rindex('.')]
    else:
        path_name = path_filepath
    logger.debug('Filename of path: ' + path_name)

    # Load the path
    try:
        logger.info('Loading path: {}'.format(path_filepath))
```

```
        path_loader = PathLoader(path_filepath)
    except Exception as ex:
        logger.error('Failed to load path {}:\n {}'.format(path_filepath, ex))
        exit()

    pos_path = path_loader.positionPath()

    logger.info('Sending commands to MRDS server listening at
       {}'.format(mrds_url))

    # Instantiate the chosen Controller with optimized parameters, or default
       ones
    if obstacle_detection:
        if path_name in PARAMETERS['obstacle']:
            controller = ObstacleController(mrds_url,
               PARAMETERS['obstacle'][path_name][0],
                                                 PARAMETERS['obstacle'][path_name][1],
                                                     PARAMETERS['obstacle'][path_name][2])
        else:
            controller = ObstacleController(mrds_url, OBSTACLE_DEFAULT_LIN_SPD,
               OBSTACLE_DEFAULT_MAX_LOOKAHEAD,
                                                 OBSTACLE_DEFAULT_DELTA_POS)
        logger.info('Starting obstacle optimized pure pursuit')
    else:
        if path_name in PARAMETERS['fixed']:
            controller = FixedController(mrds_url,
               PARAMETERS['fixed'][path_name][0],
               PARAMETERS['fixed'][path_name][1],
                                              PARAMETERS['fixed'][path_name][2])
        else:
            controller = FixedController(mrds_url, FIXED_DEFAULT_LIN_SPD,
               FIXED_DEFAULT_LOOKAHEAD,
                                              FIXED_DEFAULT_DELTA_POS)
        logger.info('Starting fixed lookahead pure pursuit')

    if path_name == "Path-from-bed":
        controller.u_turn()

    # Start stopwatch and start the controller logic sending instructions to
       the robot
    begin_time = time.time()
    controller.pure_pursuit(pos_path)
    end_time = time.time()

    logger.info('Path done in {}'.format(end_time - begin_time))
```

```
"""
Example demonstrating how to communicate with Microsoft Robotic Developer
Studio 4 via the Lokarria http interface.

Author: Erik Billing (billing@cs.umu.se)

Updated by Ola Ringdahl 204-09-11
"""

MRDS_URL = 'localhost:50000'
```

```python

import http.client, json, time
from math import sin, cos, pi, atan2

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


class UnexpectedResponse(Exception): pass


def postSpeed(angularSpeed, linearSpeed):
    """Sends a speed command to the MRDS server"""
    mrds = http.client.HTTPConnection(MRDS_URL)
    params = json.dumps({'TargetAngularSpeed': angularSpeed,
        'TargetLinearSpeed': linearSpeed})
    mrds.request('POST', '/lokarria/differentialdrive', params, HEADERS)
    response = mrds.getresponse()
    status = response.status
    # response.close()
    if status == 204:
        return response
    else:
        raise UnexpectedResponse(response)


def getLaser():
    """Requests the current laser scan from the MRDS server and parses it into
        a dict"""
    mrds = http.client.HTTPConnection(MRDS_URL)
    mrds.request('GET', '/lokarria/laser/echoes')
    response = mrds.getresponse()
    if (response.status == 200):
        laserData = response.read()
        response.close()
        return json.loads(laserData)
    else:
        return response


def getLaserAngles():
    """Requests the current laser properties from the MRDS server and parses it
        into a dict"""
    mrds = http.client.HTTPConnection(MRDS_URL)
    mrds.request('GET', '/lokarria/laser/properties')
    response = mrds.getresponse()
    if (response.status == 200):
        laserData = response.read()
        response.close()
        properties = json.loads(laserData)
        beamCount = int((properties['EndAngle'] - properties['StartAngle']) /
            properties['AngleIncrement'])
        a = properties['StartAngle']  # +properties['AngleIncrement']
        angles = []
        while a <= properties['EndAngle']:
            angles.append(a)
            a += pi / 180  # properties['AngleIncrement']
```

```
        # angles.append(properties['EndAngle']-properties['AngleIncrement']/2)
        return angles
    else:
        raise UnexpectedResponse(response)


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

def getHeading():
    """Returns the XY Orientation as a bearing unit vector"""
    return heading(getPose()['Pose']['Orientation'])


def heading(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"]
    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




if __name__ == '__main__':
    print('Sending commands to MRDS server', MRDS_URL)
    try:
        print('Telling the robot to go streight ahead.')
        response = postSpeed(0, 0.1)
        print('Waiting for a while...')
        time.sleep(3)
        print('Telling the robot to go in a circle.')
        response = postSpeed(0.4, 0.1)
    except UnexpectedResponse as ex:
        print('Unexpected response from server when sending speed commands:', ex)

    try:
        laser = getLaser()
        laserAngles = getLaserAngles()
        print('The rightmost laser bean has angle %.3f deg from x-axis (streight forward) and distance %.3f meters.' % (
            laserAngles[0], laser['Echoes'][0]
        ))
        print('Beam 1: %.3f Beam 269: %.3f Beam 270: %.3f' % (
        laserAngles[0] * 180 / pi, laserAngles[269] * 180 / pi, laserAngles[270] * 180 / pi))
    except UnexpectedResponse as ex:
        print('Unexpected response from server when reading laser data:', ex)

    try:
        pose = getPose()
        print('Current position: ', pose['Pose']['Position'])
        for t in range(30):
            print('Current heading vector: X:{X:.3}, Y:{Y:.3}'.format(**getHeading()))
            time.sleep(1)
    except UnexpectedResponse as ex:
        print('Unexpected response from server when reading position:', ex)
```

```
import unittest

from lokarriaexample import qmult, conjugate, rotate, heading
from controller import PathLoader


def are_vect_dict_equal(quat, quat_dict):
    return quat.x == quat_dict['X'] and quat.y == quat_dict['Y'] and quat.z ==
```

```python
quat_dict['Z']

def are_quat_dict_equal(quat, quat_dict):
    return quat.x == quat_dict['X'] and quat.y == quat_dict['Y'] and quat.z == quat_dict['Z'] and quat.w == quat_dict['W']

class TestMathsModule(unittest.TestCase):
    p = PathLoader('paths/Path-around-table-and-back.json')
    vect_dicts = p.positionPath(dict=True)
    vects = p.positionPath()
    quat_dicts = p.orientationPath(dict=True)
    quats = p.orientationPath()

    def test_loading(self):
        for i in range(len(self.quats)):
            self.assertTrue(are_quat_dict_equal(self.quats[i], self.quat_dicts[i]))
        for i in range(len(self.vects)):
            self.assertTrue(are_vect_dict_equal(self.vects[i], self.vect_dicts[i]))

    def test_conjugation(self):
        self.assertTrue(are_quat_dict_equal(self.quats[0].conjugate(), conjugate(self.quat_dicts[0])))

    def test_multplication(self):
        self.assertTrue(are_quat_dict_equal(self.quats[0] * self.quats[1], qmult(self.quat_dicts[0], self.quat_dicts[1])))

    def test_rotation(self):
        self.assertTrue(are_vect_dict_equal(self.quats[0].rotate(self.vects[0]), rotate(self.quat_dicts[0], self.vect_dicts[0])))

    def test_heading(self):
        self.assertTrue(are_vect_dict_equal(self.quats[0].heading(), heading(self.quat_dicts[0])))

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