Summary Homework 3

Lectures 1-14 (Max (useful) score: 110)

16-311: Introduction to Robotics (Fall 2017)

OUT: October 3, 2017, at 1:00am

DUE: October 13, 2017 at 9:00am - Available late days: 1

Instructions

Homework Policy

Homework is due on autolab by the posted deadline. As a general rule, you have a total of 8 late days. For this homework you cannot use more than 1 late day. No credit will be given for homework submitted after the late day. After your 8 late days have been used you will receive 20% off for each additional day late.

You can discuss the exercises with your classmates, but you should write up your own solutions. If you find a solution in any source other than the material provided in the course, you must mention the source.

Rules for grading: The homework consists of 6 Sections. Each section addresses one set of related topics. You have to answer (in a non-trivial way) to at *least one question per section*. If you don't answer the question from a section, your score will be decreased by the number of points of the question with the higher number of points in that Section. The grading scale is:
$[50, 60] \rightarrow D, \quad [61, 70] \rightarrow C, \quad [71, 90] \rightarrow B, \quad [91, 110) \rightarrow A, \quad [110, 228] \rightarrow A^{+}$.

In general, even if you don't answer a question, read it and try to understand the reasons why the question is of theoretical/practical importance, and try to figure out how you *would* answer it. This will help with the midterm exam. Moreover, keep in mind that some of the answered questions will be re-presented in another homework.

Submission

Create a tar archive of the folder with your ROS packages/nodes and submit it to Homework 3 on autolab. You should also have one PDF file in your archive, with an explaination of your results regarding the experiments with the robot, and the instructions for running the nodes, if any.

Contents

1 Holonomicity, degrees of freedom, maneuverability, ICR (25 points)

1.1 Questions mix (10 points)

Which ones of the following statements are true or false? Add a brief justification of the answer.

1. A bicycle robot can be found in any arbitrary pose $\begin{bmatrix} x & y & \theta \end{bmatrix}^T$ in the plane.

2. A locked bicycle, with steering angle γ locked to $\gamma = \pi/4$ has one degree of maneuverability, and can reach any arbitrary pose in its workspace.

3. A train robot (e.g., an automatic driving subway train) is non holonomic.

4. A tricycle has higher degrees of maneuverability of a unicycle.

5. A unicycle has two degrees of freedom.

6. An unsteered cart has two degrees of freedom in the plane (an unsteered cart is a car-like vehicle with all four fixed standard wheels, their wheel planes are aligned with each other).

7. The workspace of any planar robot that has less than or equal to 2 degrees of maneuverability is a circle.

8. A robot with 3 degrees of maneuverability is holonomic.

9. A robot with 3 degrees of maneuverability can achieve any path and any trajectory in its workspace.

10. An hovercraft with two parallel but not collinear propellers is an underactuated system.

11. A robot arm with 10 links is an overactuated system in the 3D workspace.

12. A unicycle is a fully actuated system.

13. A bicycle robot has full control over its ICR (i.e., it can place it anywhere in the plane).

1.2 Integration of kinematic constraints (6 points)

Let's consider the unsteered cart robot mentioned above. For simplicity, let's assume that in the map reference system `M` the pose of the cart is $\begin{bmatrix}0 & 0 & 0\end{bmatrix}^T$ (i.e., it is at the origin, heading in the X^+ direction). Prove that the robot is holonomic. You have to write down the kinematic constraints equations and show that it's possible to integrate them and reduce them to equations in the configuration variables only.

1.3 ICR of articulated vehicle (5 points)

Consider the articulated vehicle shown in the figure. All wheels are fixed standard ones, the vehicle steers by controlling the angle ϕ. Compute the coordinates of the ICR (as a function of a, b, ϕ) given that a vehicle pose $\boldsymbol{q}_M(t) = \begin{bmatrix}x & y & \theta\end{bmatrix}^T$ in `M` is measured at the middle of the virtual front axis.

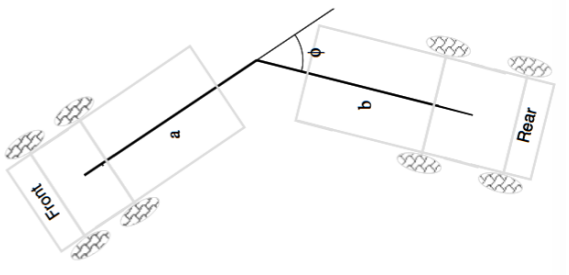

Figure 1: An articulated vehicle.

1.4 ICR of a multi-link robotic arm (4 points)

Consider the multibar system in the figure. Point A is stationary, joint B is prismatic. Find the relative position of the ICR with respect to the robot, and compute the angular velocities around the ICR for the mobile links $\overline{BC}$ and $\overline{CD}$ based on the lengths and angles shown in the figure and knowing that the instantaneous velocity of B is $v_B = 10$ m/s.

2 Forward kinematics (15 points)

2.1 Differential drive robot (9 points)

Let's consider a differential drive robot like our Turtlebot. It is a two-wheeled differential drive one with two castor wheels for stability. Each fixed standard wheel has a radius of $r = 0.0351$ m and the distance between wheels is $2\ell = 0.23$ m. The maximum speed is 0.7 m/s.

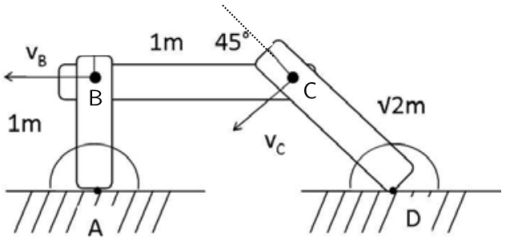

Figure 2: A multi-linkage system.

At time $t = 0$ robot's pose in $\{M\}$ is: $\boldsymbol{q}_M(0) = \begin{bmatrix} 0 & 0 & \pi/3 \end{bmatrix}^T$, and $\dot{\boldsymbol{q}}_M(0) = \begin{bmatrix} 0 & 0 & 0 \end{bmatrix}^T$.

Let's assume that velocities $v_l = 0.5$ m/s and $v_r = 0.75$ m/s are applied *instantaneously* to the left and right wheels respectively, and are then maintained constant.

1. Write down the equations and compute the value of the *instantaneous generalized velocity vector* $\dot{\boldsymbol{q}}_M$ in the `map` frame and in the `base_link` local robot frame.

2. Compute the coordinates of the ICR around which the robot is performing a circular motion and the angular velocity of rotation around the ICR.

3. Still assuming ideal conditions, compute the robot pose $\boldsymbol{q}_M(t)$ at time $t = 10$ seconds.

2.2 Car-like robot (6 points)

This time, let's consider a car-like robot, abstracted using the bicycle model such that the distance between the axels of the front and rear wheels is $L = 0.57$ m. At time $t = 0$ robot's pose in $\{M\}$ is: $\boldsymbol{q}_M(0) = \begin{bmatrix} 1 & 0 & \pi/4 \end{bmatrix}^T$, and the vehicle is initially at rest. Robot's configuration is defined by four parameters, $\boldsymbol{q}'(t) = \begin{bmatrix} x & y & \theta & \gamma \end{bmatrix}^T$, where γ is the steering angle of the front wheel, $\gamma(0) = 0$.

1. Let's assume that a linear (traction) velocity $v = 0.5$ m/s is applied *instantaneously* and the front wheels are steered *instantaneously* by an angle $\gamma = 30$ degrees. These controls are maintained for 2 seconds. At time $t = 2$, could the robot have reached pose $\boldsymbol{q}_M(0) = \begin{bmatrix} 2 & 2 & \pi/6 \end{bmatrix}^T$? You don't have to solve the kinematic equations in full (that would be too easy), but have to provide a few compelling arguments (with numbers) that show whether this could or could not be.

2. Consider the following velocity vector: $\dot{\boldsymbol{q}}_M = \begin{bmatrix} 2 & 3 & 0 \end{bmatrix}^T$. Would this be an admissible velocity at $t = 0$ for the robot given its pose and configuration at $t = 0$? If not, explain why (with equations) and tell what would be a parametric set of admissible velocities for the robot at $t = 0$.

3 Odometry for online pose estimation (15 points)

Let's consider our differential robot (parameters are as in question 2.1). Wheels' encoders are based on measuring 2578.33 ticks per wheel revolution. The encoder values are 16-bit numbers which means if they exceed 65535 they roll back to zero and restart counting. Encoder readings are published at a rate of 50Hz.

The following values are 11 (subsampled) measures from the right and left wheel encoders respectively, taken at consecutive times $t_i + \Delta t,\ i = 0, \ldots, 10, \quad \Delta t = 1s,\ t_0 = 0$.

Right Wheel:	0	2550	5450	11240	17873	23467	28890	30001	30013	30014	30017
Left Wheel:	0	2412	5722	11357	17025	22980	28720	31982	36201	39302	43112

At t_0 the robot was not moving and was at pose $\boldsymbol{q} = {}^M\begin{bmatrix} 0 & 0 & 0 \end{bmatrix}^T$ in the `map` fixed reference system `M`

The robot uses encoders information for *online pose estimation*.

1. (9 points) Compute robot pose at time t_{10} in `M` using the information from wheel encoders. This means that you have to compute robot odometry, and you have to use both an exact and an approximate way to perform numeric integration (the latter would likely be the one used by the robot in practice if the full 50 Hz stream of data is used, or if the robot has little computing power).

   Report the code used for computing both the odometry estimates and report the numerical results. Discuss the difference between the result obtained with the selected approximate approach and that from the exact numeric integration.

2. (6 points) Let's assume, ideally, that all wheels' commands are actuated in the real world with no errors or deviations from what expected. However, robot's encoders readings come with white noise Gaussian errors ϵ_L and ϵ_R, that are independent, have 0 mean and have the same stationary variance $\sigma_L^2 = \sigma_R^2 = 121$ ticks^2 (that corresponds to a standard deviation of 11 ticks, that in turn corresponds to an error of ± 0.001 m in terms of robot motion according to wheels' geometry).

   Compute the error in the odometry estimate for the position (x_t, y_t) of the robot in M at time $t = t_{10}$.

4 Inverse kinematics tasks in *open loop* (42 points)

Let's stick to a differential drive robot as described in the previous question (which is our Turtlebot, by the way) and let's consider a number of different scenarios.

4.1 Path to final pose (20 points)

At time $t = 0$ robot's pose is: $\boldsymbol{q}_M(0) = \begin{bmatrix} 5 & 0 & 0 \end{bmatrix}^T$, and from a dynamics point of view, the robot is at rest in the inertial map frame `M`. *No rotation in place* are accepted as part of the solutions below.

4.1.1 Use geometry (3 points)

Given a target pose defined as $\boldsymbol{q}_M(t > 0) = \begin{bmatrix} 5 & 15 & \pi \end{bmatrix}^T$ (in meters and degrees), compute an admissible *path* $\boldsymbol{q}(s),\ s \in [0, 1]$, in parametric form for the robot using *simple geometric considerations*. Also define the control commands in $v(t)$ and $\omega(t)$ such that the robot can reach the desired pose following the path.

4.1.2 Use inverse kinematic equations (3 points)

For the same task of the previous question, define the path by the *direct solution of the inverse kinematic equations*. Keep in mind that in this simple case the velocity profiles can be constant (until the final pose is reached, where velocities must go to zero).

4.1.3 Use differential flatness (10 points)

For the same task of the previous questions, compute an admissible *path* $\boldsymbol{q}(s),\ s \in [0, 1]$, by exploiting *differential flatness*. Use polynomials of degree higher or equal to 2 to define smooth space trajectories. Describe your choice for the polynomials and show the obtained paths for different choices of the free parameters that define the geometric speeds at the beginning and at the end of the path. Compute also the control commands in $v(t)$ and $\omega(t)$ such that the robot can reach the desired pose following the path.

4.1.4 Implement in ROS/Gazebo (4 points)

Implement your solution from the previous question in *Gazebo* and measure the error for reaching the desired final pose. Draw the expected and the achieved path.

☛ For dealing with this question and with a number of questions below, that require to know robot's state, you have two options for measuring the error (i.e., to assess the robot pose in `M` at a time t). One is realistic, the other is "idealistic". Realistic one: obtain the pose from `/odom` topic (be sure that the initial pose in

the `odometry` frame is the same as in the `map` frame, otherwise always apply a relative pose transformation). Idealistic one: get robot's ground truth directly from Gazebo. This second option won't be available when you will have to test your controller on the real robot, therefore, it might be a good idea to work directly with odometry (which is expected to be quite reliable on the Turtlebot).

4.2 Path to final pose under more challenging conditions (22 points)

4.2.1 Low friction, implement in ROS/Gazebo (5 points)

Implement your solution from the previous question in Gazebo as before, but now set the *friction coefficients of the ground plane* to very low values. In the `.sdf` or `.world` file describing the environment, look for the `ground_plane` model and change the values of the parameters `mu` and `mu2` to values near 1 (0 corresponds to a frictionless ground). You will experience significant deviations. Investigate how changing the degree of the polynomials and/or the free parameters affect robustness and stability.

4.2.2 Reverse pose (7 points)

The robot starts in the same initial pose, but now the target pose is $\boldsymbol{q}_M(t > 0) = \begin{bmatrix} 5 & 15 & 0 \end{bmatrix}^T$. First check whether the solution found answering question 4.1.3 works (of course, you need to recompute the paths considering the new goal pose). If not, change the degree of the polynomials and the free parameters to find a satisfactory solution. As before, show your results.

4.2.3 Reverse pose and low friction, implement in ROS/Gazebo (10 points)

Challenge yourself by trying out the new scenario in *Gazebo with low friction*. It will likely show a major deviation between the actual and the expected path. You need to *adjust the parameters of the path*, that in turn determines your velocity controls. Follow a procedure similar to that described for tuning a PID controller (maybe use twiddle): define an appropriate error metric, start with some guessing for the parameters, perform multiple experiments, iteratively adjust the parameters. Describe the process and show the results. Keep in mind that you're not doing feedback-based control, but you are only adjusting the response of your open loop control.

5 Inverse kinematics tasks in *closed loop* (89 points)

5.1 Wall following (31 points)

Tasks involving wall following can be stated in different ways, depending on the context, including:

1. "Keep at distance D from the wall", that assumes there is only one wall, otherwise it might be impossible to accomplish the task.
2. "Don't get closer than D to the/any wall", that says to keep a minimal clearance from all walls. This suffers from the same issues as the previous task in the case there are walls on both sides and their distance is less than $2D$. Therefore, an admissible request is "Don't get closer than D to the/any wall whenever feasible, otherwise, keep equal clearance from walls".
3. "Keep maximum clearance from all walls", which is equivalent to ask to follow a Voronoi path (we will see this later on in the course). When the distance between two walls on two opposite sides is too large this request would result in a quite inefficient path.

Let's check how different approaches work on these types of tasks.

5.1.1 Bang-bang controller (7 points)

Consider an environment like the one shown in the figure, Left (You don't need to replicate "exactly" the scenario, a good approximation is more than enough, and this applies to all scenarios sketched in figures). *Implement and tune a bang-bang controller with hysteresis* for performing task 1 in the previous list, with

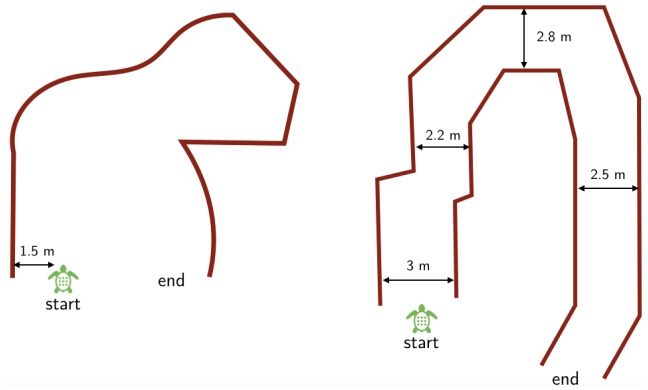

Figure 3: Wall-following scenarios. (Left) single wall on the left side. (Right) Corridor-like path, with two opposite-side walls.

$D = 1.5$ m. You can assume to use any virtual sensor that returns a precise measure of error (i.e., the current distance from the wall). Your robot should move with an average linear speed of 0.5 m/s. Apply some heuristic as described in the previous section for tuning bang-bang parameters over multiple runs. Describe the procedure and show your results (trajectory + quantification of the average error over time).

5.1.2 Bang-bang in ROS/Gazebo (5 points)

Build an equivalent environment in *Gazebo and set a low friction parameter*. This time the distance from the wall is returned by `/laserscan` measures. Repeat controller tuning process and experiments.

5.1.3 PID controller (9 points)

For the same task, design, implement, and tune a *PID controller*. For using and updating the PID, you can conveniently use the PID node provided by ROS `http://wiki.ros.org/pid`, or you can make your own implementation. Again, you will need to define appropriate metrics (e.g., sum of errors, oscillations, ...) for assessing how good/bad an experiment run is when evaluating a set of PID parameters.

5.1.4 PID controller for corridor navigation in ROS/Gazebo (10 points)

Consider now an environment equivalent to the one shown in the right part of the figure. Design a *PID controller that realizes task 2* in the list above for $D = 1.1$ m. Implement and tune the controller in Gazebo and report experiment results.

5.2 Move to a point or to a pose (25 points)

5.2.1 Move to distance point, implement in ROS/Gazebo (10 points)

This is similar to the wall following case, but the objective is to let the robot stopping smoothly at a certain distance from an obstacle. Let the robot start in $\boldsymbol{q}_M(0) = \begin{bmatrix}0 & 0 & 0\end{bmatrix}^T$ with $\dot{\boldsymbol{q}}_M(0) = \begin{bmatrix}0.7 & 0 & 0\end{bmatrix}^T$, and let a large regular obstacle be placed at a distance $d = 2$ m, as shown in the figure. Define a PID controller that let the robot smoothly and rapidly (in minimal time) arriving with zero velocity at precisely a distance $D = 0.5$ from the obstacle. Test the controller in Gazebo, assuming that the position of the obstacle in `M` is known. Study the effectiveness of the controller under two different conditions to get the relative distance $\delta_{rw}(t)$ of the robot with respect to the wall:

- $\delta_{rw}(t)$ is obtained from the measures from the on-board odometry (`/odom`);
- $\delta_{rw}(t)$ is obtained from the measures from the distance scanner (`/laserscan`).

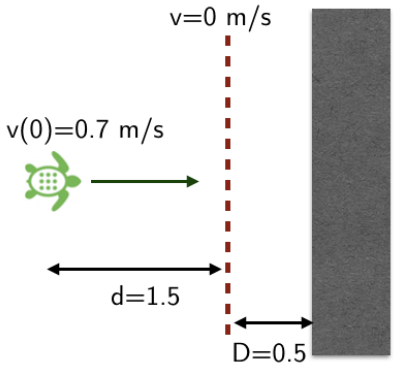

Figure 4: Scenario for the robot that has to smoothly and exactly stop at the desired distance from the obstacle.

5.2.2 Move to a pose, implement in ROS/Gazebo (15 points)

Let's solve the problem considered in question 4.2.2 with a feedback-based approach for online motion control. The robot starts in $\boldsymbol{q}_M(0) = \begin{bmatrix} 5 & 0 & 0 \end{bmatrix}$ and has to reach the target pose $\begin{bmatrix} 5 & 15 & 0 \end{bmatrix}$. Define a P controller, tune its parameters, and perform experiments in Gazebo. Based on a stability analysis on the linearized version of the system dynamics, discuss the stability of the system vs. gains values.

5.3 Path and trajectory following (33 points)

5.3.1 Follow a path, implement in ROS/Gazebo (10 points)

Define an elliptic path centered in $(0, 1.5)$, where the longer semi-axis (stretching long the x) has length 3 and the shorter 1.5 (see figure, left). The robot starts in pose $\boldsymbol{q}_M(0) = \begin{bmatrix} 5 & 0 & 0 \end{bmatrix}^T$. Define and tune a PI controller that makes the robot following the path. Implement and test it in Gazebo with a moderate friction level.

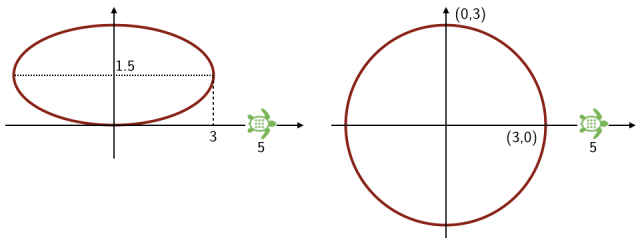

Figure 5: Left: Elliptic path that the turtlebot robot has to follow. Right: Circular trajectory to track.

5.3.2 Stability analysis (5 points)

Based on the linearization of the dynamic system for controlling the robot, discuss the asymptotic stability for the values used for the gains.

5.3.3 Follow a trajectory, stability analysis, implement in ROS/Gazebo (18 points)

Instead of an ellipsis, let's use a circular path of radius $r = 3$, centered in $(0, 0)$, as shown in the right part of the figure. Define a time parameter $t \in [0, 70]$, by time t the robot has reached point $(0, 3)$ on the circle and then follows the circular path by making one full revolution every 40 seconds (which roughly corresponds to a

tangential velocity of little less than 0.5 m/s). This means that the target trajectory is defined as follows:

$$\begin{cases} (x(t), y(t)) = (0, 3) & t \in [0, 10) \\ (x(t), y(t)) = \left(r\cos\left(\frac{2\pi}{40}t\right),\ r\sin\left(\frac{2\pi}{40}t\right) \right) & t \in [10, 70] \end{cases}$$

Make use of the property of differential flatness to compute the kinematic inputs (in v and ω) for the robot following the trajectory. However, to realize an effective trajectory tracking, feedback-based motion control is required. At this aim, consider the tracking error and linearize its dynamics in order to define a linear feedback law (a P controller).

Implement the linearized feedback controller and observe the result in Gazebo. Report the performed trajectory and a measure of the error in trajectory tracking (first, define an appropriate error measure).

Compute the eigenvalues of the linearized system and discuss the stability of the system in terms of the values assigned to the gains of the proportional controller (gains that you have to define through a trial-and-error procedure).

6 Navigation and online planning with obstacles (42 points)

Consider the cluttered environment shown in the figure, that features a start and a goal position, and a number of convex and concave obstacles. The robot has to navigate to reach the goal position using different approaches (and knowledge). The robot always knows where the goal is.

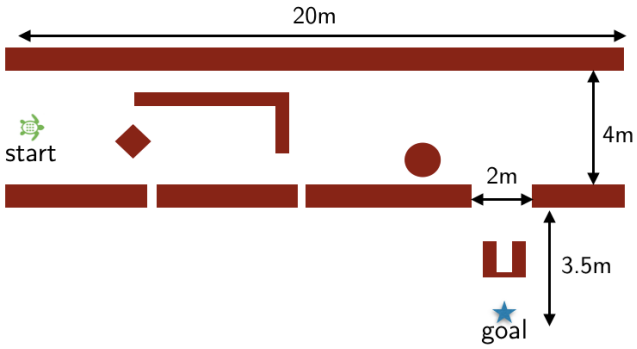

Figure 6: A complex cluttered environment for robot planning and navigation.

6.1 Dynamic Window Approach, implement in ROS/Gazebo (12 points)

The robot has a map of the environment. Navigate to the goal using a *Dynamic Window Approach*. The maximum speed is as usual 0.7 m/s. Discuss your design choices and implement the system in Gazebo. Show the performed trajectory.

6.2 Potential Fields (10 points)

The robot has a map of the environment. The objective is to navigate to the goal using a *Potential field* planning. Planning is done offline, meaning that first a potential field is computed for the scenario (in the continuous world), then a gradient descent algorithm is performed on the resulting force field to define the path. Make use of the force vectors to define the velocity vectors for the robot. Define and discuss all design choices.

Is the computed path letting the robot reaching the goal?

6.3 Randomized best-first, implement in ROS/Gazebo (10 points)

The robot makes use of its sensors (`/laserscan`) to make a *local obstacle map*. *Iteratively use potential fields to navigate towards the goal*. Implement it in Gazebo and show the resulting path.

Local minima are an issue and might make the robot getting stuck, or not making any actual progress toward the goal. Therefore, in order to overcome this (likely) issue, adopt a *randomized best-first* version of the potential field approach, that requires a discretization of the search space. Below a pseudo-code for the randomized best-first potential fields approach:

Randomized_BestFirst_PotentialFields(startState)
  $q \leftarrow$ startState
  $t \leftarrow 0$
  Repeat:
    If $U(q) == 0$:
      Return (Success, q)
    If too many iterations:
      Return (Failure)
    Else:
      Find the neighbor q_n of q with smallest $U(q_n)$
      If $(U(q_n) < U(q))$:
        $q \leftarrow q_n$
        $t \leftarrow 0$
      Else (q_n has not yet been visited) AND ($t <$ maxFillTime):
        $q \leftarrow q_n$ AND remember q_n
        visited[] $\leftarrow q_n$
        $t \leftarrow t + 1$
      Else:
        $k \leftarrow 0, \quad q_e \leftarrow q, \quad t \leftarrow 0$
        While ($k <$ maxEscapeSteps OR $U(q_e) < U(q)$):
          $q_e \leftarrow RandomWalk(q_e)$

6.4 Best-first path smoothing, implement in ROS/Gazebo (3 points)

In the previous solution, because of discretization, the path might result zigzagging. A way to overcome it is to perform some form of polynomial interpolation. Select one (e.g., cubic splines) and use it to have smooth trajectories. You can use the `scipy` library that comes with easy to use functions for interpolation, docs.scipy.org/doc/scipy/reference/tutorial/interpolate.html.

6.5 Bug algorithms, implement in ROS/Gazebo (7 points)

Navigate to the goal using a *bug algorithm*. Select one bug algorithm (justify your choice) and implement it in Gazebo. Obstacles are detected using data from `/laserscan`. Show the resulting path.