Dynamic Modeling, Control and Simulation of a Differential Drive Robot

MEE 4411/5411 Project 1, Phase 1

Due: Thursday, September 14th

1 System Description

Small ground robots are used all over the place for teaching, research, and commercial activities. Differential drive systems are the most common due to their small footprint, mechanical simplicity, and maneuverability. This project will explore the kinematics, control, and planning of differential drive robots.

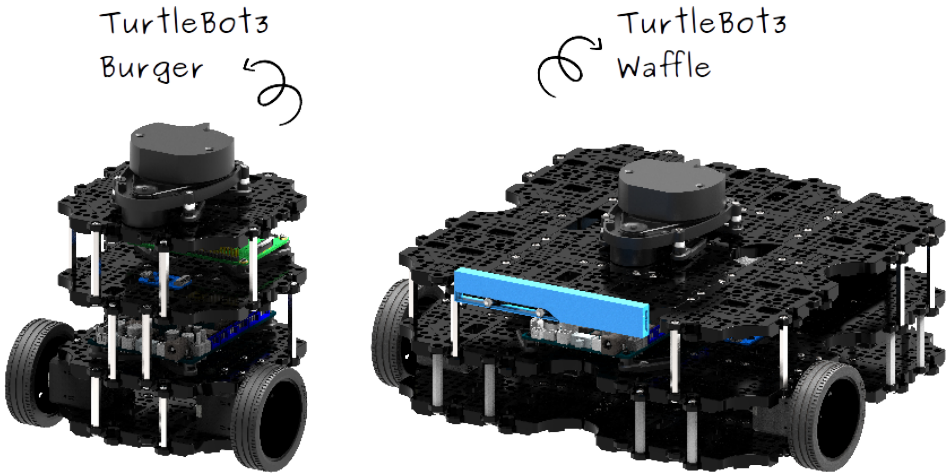

Figure 1: The TurtleBot3 robot models. This comes in 2 "flavors," the Burger and the Waffle, with the main difference being the size of the robots.

We will use the TurtleBot3 Burger robot. This is an open-source platform, both in terms of hardware and software, and was released in May 2017. The Burger has a maximum speed of 0.22 m/s and a maximum rotational velocity of 2.84 rad/s. The size of the robot is shown below in Figure 2.

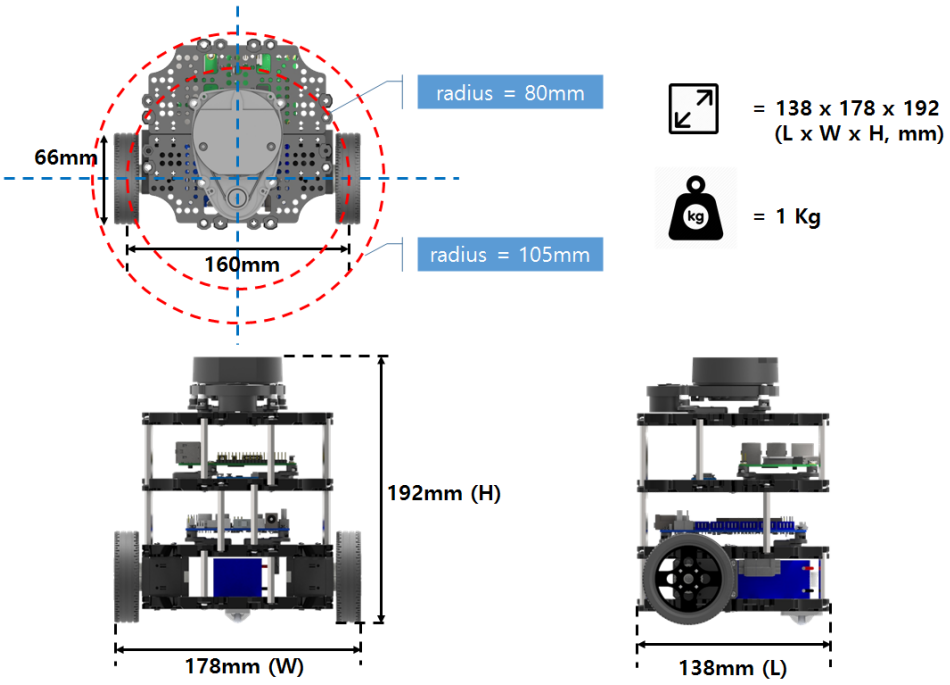

Figure 2: Specs of the TurtleBot3 Burger.

2 Controller

The goal of Phase 1 of Project 1 is to implement the pure pursuit controller[1] and use this to drive a robot through a sequence of waypoints in a simulated, obstacle-free environment. As explained in lecture, the basic idea of pure pursuit is very simple:

1. Find the current location of the robot
2. Find the point on the path that is closest to the robot
3. Find the goal point
4. Transform the goal point (in global coordiantes) to local vehicle coordinates
5. Calculate the radius (or curvature) of the arc connecting the robot to the goal point
6. Set the linear and angular velocity of the robot.

The rest of this section will describe these steps in greater detail.

[1] See http://www.ri.cmu.edu/pub_files/pub3/coulter_r_craig_1992_1/coulter_r_craig_1992_1.pdf and http://www8.cs.umu.se/kurser/TDBD17/VT06/utdelat/Assignment%20Papers/Path%20Tracking%20for%20a%20Miniature%20Robot.pdf for more details.

2.1 Find the Current Location

The pose of the robot is given by the simulation. The details of format of this data is given in the starter code provided with this handout.

2.2 Find the Closest Point

Finding the closest point on an arbitrary line to a specified point is a non-trivial task. To get around this we will make one major, simplifying assumption: that the path of the robot is specified by a sequence of straight line segments connecting waypoints. Let $\mathbf{p}_i = [x_i, y_i]^T$ be a point, then the path of a robot can be represented by a $2 \times n$ matrix $P = [\mathbf{p}_1, \mathbf{p}_2, \cdots, \mathbf{p}_n]$. The first segment is a straight line connecting $\mathbf{p}_1$ and $\mathbf{p}_2$, and, in general, the ith segment connects $\mathbf{p}_i$ and $\mathbf{p}_{i+1}$. So if there are n points, then there are $n-1$ segments.

Before we tackle the problem of finding the closest point on the path, let us first consider a single segment. There are three possibilities, a point can be closest to the a point along the line segment, it can be closest to the start of the segment, or closest to the end of the segment. For an example of this, see Figure 3. Given this, one way to find the closest point from a line segment to a point is to first project the point onto the line and check to see if the projection lies within the bounds of the segment. If it does, then the projected point is closest, and, if not, then one of the end points is closest. Hint: for a line going through point $\mathbf{p}$ and with a unit direction vector $\mathbf{v}$, then the projection of the point $\mathbf{x}$ onto the line is given by $\mathbf{p} + ((\mathbf{x} - \mathbf{p}) \cdot \mathbf{v})\,\mathbf{v}$.

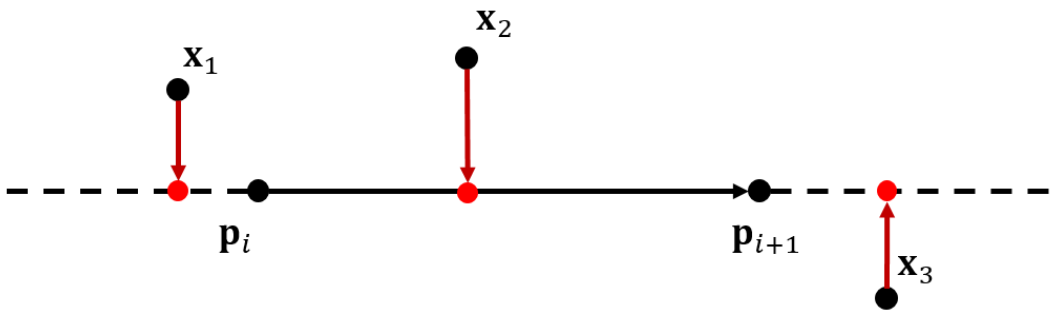

Figure 3: Consider segment i connecting points $\mathbf{p}_i$ and $\mathbf{p}_{i+1}$. The dashed line extends the line segment out to infinity. The red points indicate the projections of the three candidate points onto the line. Point $\mathbf{x}_2$ is closest to the projected point (shown in red) on the line. However, the projections of point $\mathbf{x}_1$ and $\mathbf{x}_3$ line beyond the boundaries of the line segment, so they are closest to $\mathbf{p}_i$ and $\mathbf{p}_{i+1}$, respectively.

Now that we have a method to find the closest point to a single segment, we can use this to find the closest point along the entire path. We simply have to iterate over each segment of the path and keep track of the segment that is closest to the robot. Once we have the point, it is simple to find the distance. We can simply use the length of the vector connecting the nearest point to the robot. You will implement these concepts in the function `find_closest_point`. The details of the inputs and outputs of this function are provided in the starter code. Also, if the robot passes through the same point twice (meaning the path intersects itself), then the robot should not jump ahead on the path. Hint: look at `persistent` and `global` variables in MATLAB.

2.3 Find the Goal Point

Once you have the closest point on the path to the robot, you must find the goal point. Recall, the lookahead point is a distance ℓ in front of the robot. So to find this point, you need to consider where a circle of radius ℓ centered at the robot first intersects the path ahead. See Figure 4 for an example of this.

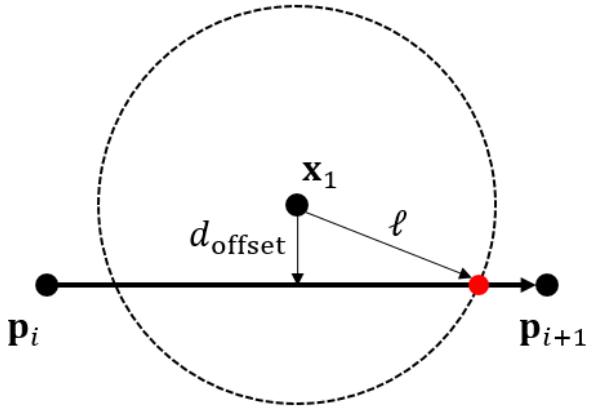

Figure 4: The goal point is set as the point where the path intersects a circle of radius ℓ centered at the robot's current position. The distance d_{offset} is the distance from the robot to the nearest point on the path.

To find this point, the robot should begin searching from the nearest point on the path. The robot should then traverse the path forward until the path leaves the circle. Note, if the nearest segment ends inside of the circle, the robot should then move on to the next segment and check that. Also, if the nearest point is greater than a distance ℓ from the path, the robot should drive towards the nearest point. This will not happen in general, since getting so far away from the path means that your controller did not do a good job of following the path.

2.4 Transform into Local Coordinates

All of these previous steps have taken place in the world coordinate frame. To find the wheel velocities, we must transform the goal point from the world frame to the vehicle frame. You can do this using the transformation equations discussed in the first week of lecture.

2.5 Calculate the Radius of the Arc to the Goal Point

We derived this relationship during lecture. The radius of the arc $R = \frac{\ell^2}{2y}$, where y is the y-coordinate of the goal point in the local robot frame and ℓ is the lookahead distance.

2.6 Set Robot Velocity

We also computed this in lecture. The relationship between the linear and angular velocity of the robot and the radius of the arc is $R = \frac{v}{\omega}$, where v is the linear velocity and ω is the angular velocity. This one relationship is not enough to completely determine both v and ω. To do this, we assume that we want the robot to move as quickly as possible. However, the problem is further complicated by the fact that we have limits on the wheel velocity, not on the robot velocity. Figuring out a method to find the wheel velocities given the actuator limits and the relationship that $R = \frac{v}{\omega}$

is left to you. Hint: you can multiply both the linear and angular velocity by a constant c and the radius of the arc will be unchanged $R = \frac{v}{\omega} = \frac{cv}{c\omega}$.

3 Project Work

3.1 Tasks

You will simulate the TurtleBot3 dynamics and control using the Matlab simulator posted on the course Piazza page. The simulator relies on the numerical solver `ode45`, details about this solver can be easily found online. You don't need to be an expert in numerical methods, but it would be beneficial if you know the basics of how ODE solvers work. Your tasks include:

1. **Trajectory Generator**
   You will first implement two trajectory generators that generate a circle and a zigzag trajectory. In order to specify the trajectory in the simulator, you will need to change the variable `trajhandle` in the function `runsim`. `trajhandle` is the name of a function that takes in no inputs and returns a matrix of the waypoints. The matrix should be $2 \times n$, where n is the number of waypoints.

2. **Find Closest Point**
   You will need to implement a function `find_closest_point` to find the closest point on a path to the position of a robot. This takes in the 2D position of a robot `position`, a matrix of waypoints `waypts`, and, optionally, a segment number `segment` and returns both the closest point `pt_min` and the distance from the robot to that point `dist_min`. When the optional segment number is specified, the point `pt_min` must lie on the specified segment of the path. This function is used in the simulator, and you are encouraged to use it within your controller.

3. **Controller**
   You will then implement a controller in file `controller.m` that makes sure that your robot follows the desired trajectory. This controller will be used again in the following phases. The controller takes in a struct `qd`, a matrix of waypoints `waypts`, the current time `t`, and robot parameters `params` and outputs the wheel velocity vector `u`.

4. **Simulation**
   Lastly, you need to fill in the appropriate variables for `trajhandle` and `controlhandle` in the script `runsim.m` for simulating your result.

3.2 Simulator

The robot simulator comes with the student code. Before implementing your own functions, you should first try running `runsim.m` in your matlab. If you see a robot sitting still then the simulator works on your computer and you may continue with other tasks. This is because the outputs of `controller.m` are all zeros, thus no motion is generated.

When you have the basic version of your trajectory generator and controller done, you will be able to see the robot moving in the space, leaving trails of desired and actual position behind. The desired position is color-coded blue and the actual position is red. After the simulation finishes, two plots of position will be generated to give you an overview of how well your trajectory generator and controller are doing.

3.3 Submission

When you are finished you may submit your code by putting it all into a single `.zip` file and emailing it to pdames@temple.edu with subject line "MEE4411 Project 1.1" (clicking the previous link will automatically start a correctly formatted email for you). Prof. Dames will test your submission and send you a list of items to fix. You are encouraged to update your code to fix any bugs, however each student is limited to 3 submissions per phase of the project.