Massachusetts Institute of Technology

Robotics: Science and Systems I

Lab 7: Motion Planning and Global Navigation

Begins: Wednesday, 3/18/2009, 3pm
Checkpoint: Monday, 3/30/2009, 3pm
Due: Wednesday, 4/1/2009, 3pm

Objectives and Lab Overview

Your objective in this lab is to understand planning and global navigation. You will control your robot to a goal in a maze-like environment, using a motion planner to establish a sequence of waypoints within a given map, around obstacles, and to the goal. Given a map and the geometry of your robot, you will first compute the configuration space that allows you to reason about your robot as a moving point among complex obstacles. You will then develop a motion planning program that will search the configuration space for a path for the robot. This path will be transformed into waypoints in the physical space. Your robot will then follow the path defined by the computed waypoints to navigate from its starting configuration to the goal configuration.

This lab will give you the technical skills to incorporate motion planning and waypoint navigation algorithms into your robot. This lab will also enhance your knowledge of geometric algorithms, an important aspect of interfacing computation to the physical world.

Time Accounting and Self-Assessment:

Make a dated entry called "Start of Global Navigation Lab" on your Wiki's Self-Assessment page. Before doing any of the lab parts below, answer the following questions:

- **Programming**: How proficient are you at writing large programs in Java (as of the start of this lab)?
  (1=Not at all proficient; 2=slightly proficient; 3=reasonably proficient; 4=very proficient; 5=expert.)
- **Computational Geometry**: How proficient are you at working with computational geometry (representing polygons, computing intersections, etc)?
  (1=Not at all proficient; 2=slightly proficient; 3=reasonably proficient; 4=very proficient; 5=expert.)
- **Motion Planning**: How proficient are you at crafting robot motion planning algorithms?
  (1=Not at all proficient; 2=slightly proficient; 3=reasonably proficient; 4=very proficient; 5=expert.)
- **Waypoint Navigation**: How proficient are you at using the navigation software on your robot?
  (1=Not at all proficient; 2=slightly proficient; 3=reasonably proficient; 4=very proficient; 5=expert.)

To start the lab, you should have:

- This handout (7-A)
- Your notes from the Planning, Localization, and Mapping lectures

Physical Units

We remind you to use MKS units (meters, kilograms, seconds, radians, watts, etc.) throughout the course and this lab. In particular, this means that *whenever you state a physical quantity, you must state its units*. Also show units in your intermediate calculations.

Part 1: Team organization

This week's assignment has several parts that can be implemented in parallel provided you define appropriate interfaces (APIs) between the project components. To begin, read the project specifications carefully and work as a group to define your approach to solving the assignment. Consider the overall project goal and break it down into software modules. Develop an algorithm for each software module. Define the APIs between all the software modules. Now make programming assignments for each software module so that you can parallelize the implementation of this project. Develop a project timeline and be sure to include time for module development and testing, time for software integration, and time for project testing and evaluation.

Deliverables: Your report on the Wiki, named "GlobalNavigation Lab Report Group N". Include a description of each module, your proposed algorithm for it, the APIs between the modules, the task allocations within your team, and the proposed timeline.

Part 2: Programming Environment

We have supplied a number of Java classes and one map file (`global-nav-maze.map`, corresponding to the maze environment we have constructed) in the lab distribution directory. Copy them from `~/RSS-I-pub/labs/GlobalNavigation` and check them in, as usual, to your group repository.

Read the Javadoc for all the code we have provided.

As a warm-up exercise you will use the code we have provided to read in the map and display it in the GUI.

1. Create a new Java class called `GlobalNavigation`. Read the `PolygonMap` API to find out how to read a map file and how to access the components of the map. Also, look at the data-types returned in the `PolygonMap` accessor methods.
2. In `GlobalNavigation.main()`, create an instance of `PolygonMap`, reading in a map file whose name is passed on the command line, and store a reference to it in an instance field.
3. Implement a method `GlobalNavigation.displayMap()` which displays all the contents of the map in `MapGUI` (as usual, you will run the provided `MapGUI` as a separate process. Make sure that both your `GlobalNavigation` program and your `MapGUI` are configured to initialize with Carmen using the same `centralHost`, and that the Carmen daemons are running on that machine).

Hint: For reference, read and understand the code in `PolygonMap.main()`.

Now continue by using the code provided to you for the convex hull algorithm (in `GeomUtils`) to compute and draw the convex hull of a set of points. Recall that the convex hull of a set of points is the smallest convex polygon containing all the points. You can visualize the convex hull by imagining gift-wrapping the set of points.

1. Implement a method `GlobalNavigation.testConvexHull()`. In this method you will create instances of `java.awt.geom.Point2D.Double`s to test the convex hull algorithm given in GeomUtils.
2. Look at the API for the `GeomUtils` class and find the specifications for calling the convex hull method.
3. Plot out a (non-trivial!) set of test points. Then, in your code, create a corresponding array of Java points and test the convex hull algorithm.
4. Use `MapGUI` to visualize the input points and the polygon returned by the convex hull (temporarily disable display of the map).

Deliverables: Screen shots of both the map and convex hull in `MapGUI`. Be sure to include a list of the coordinates of the points whose convex hull you computed.

Part 3: Building a Configuration Space

In this part of the assignment, you will implement construction of the configuration space. Your C-space module will take as input a list of convex polygonal obstacles (`PolygonObstacle`), which will be specified by the map.

You will also have to specify the geometry of your robot as a parameter, as the computation of configuration space depends both on the geometry of the obstacles and the geometry of the robot. The goal is to compute the corresponding configuration space for the robot.

Recall that the configuration space of an object with respect to a robot allows you to reason about point robots by growing the obstacles in the map. The configuration space depends on the shape of your robot and its degrees of freedom. Your robots can rotate and translate in the plane. The true c-space will thus be a 3-dimensional object with algebraic surfaces. Although we have algorithms for computing such 3D c-spaces, in this assignment we will simplify the problem to include translations only. We can abstract out your robot's rotational degrees of freedom by growing the robot. Think of including the robot in a disk. Then the intersection between your robot and obstacles is independent of the orientation of the robot and thus we can omit considering the rotational degree of freedom of the disk robot in the c-space calculation. Now we will consider a further simplification. Although there are algorithms for computing c-spaces for objects with algebraic surfaces (a circle is an algebraic surface, not a polygon), the algorithm described in class works for the simplified case of polygonal-shaped robots. Thus we will do a further simplification to approximate the disk robot by a polygon. One option is to inscribe the disk in a square. Another is to do a polygonal approximation of n edges to the circle defining the robot.

Recall the algorithm for computing a c-space obstacle $C(O)$ relative to a robot R is

$$C(O) = ConvexHull(vert(O) \oplus (\ominus vert(R)))$$

In other words, after reflecting the robot you compute the convex hull of the Minkowski sum of the obstacle vertices and the reflected robot vertices. You will have to do this computation for each obstacle in the world. Remember, we have already provided Java code for computing the ConvexHull of a set of points.

Design and implement code which accomplishes the following:

1. Given two polygons, compute their Minkowski sum.
2. Given a polygon A and a reference point r, reflect A about r (note that the reflection of a square about its center point is the same square, and the same holds for any n-edge polygonal approximation of a circle about its center point when n is even. Thus, if you choose to implement either of those cases, you may omit the reflection code, but document your choice in your lab report.)
3. Given an obstacle polygon, a robot polygon, and the robot reference point, compute the configuration space obstacle for the polygon.
4. Given a map with several obstacles, compute the configuration space of this environment. You may or may not choose to include a representation of the map boundary in your configuration space. If you choose not to, you will have to handle it at a higher level (you must somehow guarantee that your robot will not drive into the maze walls).
5. Add code to your `GlobalNavigation` class to compute and display the configuration space of the map file specified on the command line. Display both the real obstacles and their corresponding configuration space obstacles. (disable your convex hull test code now).
6. Test this part of the assignment on the map we have provided, which corresponds to the maze set up for you in the Hangar. Then, test the algorithms on the two other environments (`practice-maze-01.map` and `practice-maze-02.map`). The format of the map file is documented in the Javadoc for the `PolygonMap` class.
7. **Extra Credit: Implement a more complete configuration space computation algorithm.** For example, consider a robot which can be an arbitrary convex polygon with an arbitrary reference point. Or, relax the constraint that the robot and obstacles must be specified as convex polygons.

Hint: Reflecting the robot about its center point or one of its vertices makes the results of the c-space computation easier to visualize.

What happens when you supply overlapping obstacles in your c-space computation algorithm? Consider an environment in which two adjacent convex obstacles form an L-shaped object. How does your algorithm handle this case?

Deliverables: Your wiki page should include a screenshot of the corresponding c-space obstacles for each of the three maps you tested and answers to all questions above.

Part 4: Motion Planning

Following Part 3, you will build a program that searches the configuration space for a collision-free path from the given start location to the desired goal location. Recall that the shortest path goes along the edges of the visibility graph. For this lab, you do not have to compute the *shortest* path—*any short* path which takes the robot from the start to the goal will suffice. Note that we said short—the robot's path should not be ridiculously longer than the shortest path. Also, your path planner should be *complete*: if a path exists, your planner should find it (in finite time). The result of this part will be a sequence of position waypoints $(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)$ where (x_1, y_1) is the initial robot position and (x_n, y_n) is the goal location. These points represent a collision-free path from start to goal.

For this part you can implement any motion planner of your choice. In lecture we described in detail an algorithm for *grid-based motion planning* with potential fields. Alternatively, you can implement a different planner such as the visibility graph motion planning algorithm, the Vornoi diagram motion planning algorithm, or other grid-based methods. Your notes describe these algorithms. Any standard algorithms text book will have some guidance on motion planning algorithms; you can also find extensive documentation for almost any such algorithm on the web.

Deliverables: Your wiki page should include a description of your motion planning algorithm (with pointers to the corresponding code), and the result of running your algorithm on the map that corresponds to the maze we constructed for you (write code to illustrate the results of the algorithm in the GUI, including the computed path from start to goal). You should also run the algorithm on the two additional environments, and include the results. What is the time complexity for your algorithm in big-O notation? How much space does your algorithm use?

Checkpoint – Demonstrate Your Motion Plan

For the checkpoint, you will need to demonstrate your motion planning algorithm in action. You should be prepared to show and explain:

- Your C-space map generated from the map provided with the lab.
- The path generated by your planning algorithm. The path should be drawn on the GUI.

Part 5: Waypoint Navigation

Waypoint robot navigation refers to controlling your robot to traverse a set of points in sequence.

The final part of this lab is to implement the motion plan you computed for the maze in the hangar: given the starting configuration for your robot and the desired goal location, execute the computed motion plan on your robot. Your robot will take the output of your planner which is a sequence of position waypoints $(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)$ where (x_1, y_1) is the initial robot position and (x_n, y_n) is the goal location. The robot will do waypoint navigation by traveling linearly from point (x_i, y_i) to point (x_{i+1}, y_{i+1}). Remember that the output of the planner contains points in c-space. How do you map these points to the physical space?

Deliverables: Your wiki page should include a description of your waypoint navigation implementation and tests on the three motion plans you computed in part and the answers to the questions above. Record a video of your robot traversing the maze. Using a ruler, measure the accuracy of your navigation by measuring how close to the desired goal location your robot reached. Also, numerically estimate the maximum amount of error you observe at any waypoint. Does the error accumulate? What do you think contributes most to the error: the translational or the rotational component of the robot motion? How many times did you try your waypoint navigation algorithm, and was the observed error at the goal location consistent across the runs? How many times did it work, once you stabilized the waypoint navigation code? What were the failure modes you observed on your robot, if any?