Homework 6:
Wavefront Path Planning and Path Smoothing

Assigned: Tuesday, October 14
Due: Thursday, October 28, 2014 at the beginning of class (11:10AM)

In this assignment, you will write a path planner that makes use of wavefront propagation, smooth the resulting path (graduate students only), and then visualize the (smoothed) path by plotting it on the Stage map of the environment. In this assignment, the path is a set of *x*, *y* positions from the starting position to the goal position. Note that the orientation of the robot is not considered in this exercise. In this assignment, your robot does not have to execute the planned path – only derive it.

(Note: Player includes a Planner proxy. However, the purpose of this assignment is for you to gain experience writing your own planning software. So, you are not allowed to copy or make use of the existing planner proxy software. Instead, you must write your own wavefront planner using the algorithm outlined in class on September 30.)

Your software must operate as follows:

- Your code must accept the robot's goal position (in terms of *x*,*y* only) from the user using standard input. Your user interface should make it clear how the user should enter the goal (*x*,*y*) position. The exact format is up to you. You should do some modest error checking to make sure the goal position entered is numeric, and within the bounds of the robot's environment.
- Your code must input the map of the environment, and then create the configuration-space version of the map by growing the obstacles for the standard Pioneer robot (i.e., the robot you used in Homeworks 4 and 5).
- Your code must output the grown-obstacle map.
- Your code must execute a wavefront path planning method. The output of this function is a series of waypoints the robot should visit to reach the goal position.
- [Graduate Students Only]: Your code must smooth the resulting path.
- Your code must visualize the generated path by plotting it (as line segments) on the map of the environment (so that it is visible in the Stage simulation window.)

More details on these processes follow.

Inputting the map. Obviously, for the path planner to function correctly, it needs to load the map of the environment into memory. For this assignment, we'll use the "hospital_section" map. At present, we have a bitmap version of this map in "png" format. However this format makes use of data compression, and isn't the easiest for you to read in directly into memory for the purposes of this assignment. So, we'll convert the bitmap to another format. We have several to pick from; we'll choose the "pnm" format. Later in this document, I provide instructions on how to do this, as well as a function you can use to read in the map into a 2-dimensional binary occupancy grid array, in which 0's represent free space and 1's represent obstacles. Note that you will still use the "png" format for Stage display purposes; this "pnm" format is just for your code to have access to the map for planning purposes.

Growing the map for obstacles. You need to write a function that grows the obstacles in the grid map, so that your path planner can represent the robot as a point. This obstacle growth function must input a parameter representing the amount of growth desired, in units of the number of grid cells. Note that you will find there is a tradeoff between growing the obstacles too much and growing them too little. If you grow them too much, your path planner may not be able to plan a path to a reachable goal. If you grow them too little, the robot could come too close to obstacles. Output your grown obstacle map, to turn in.

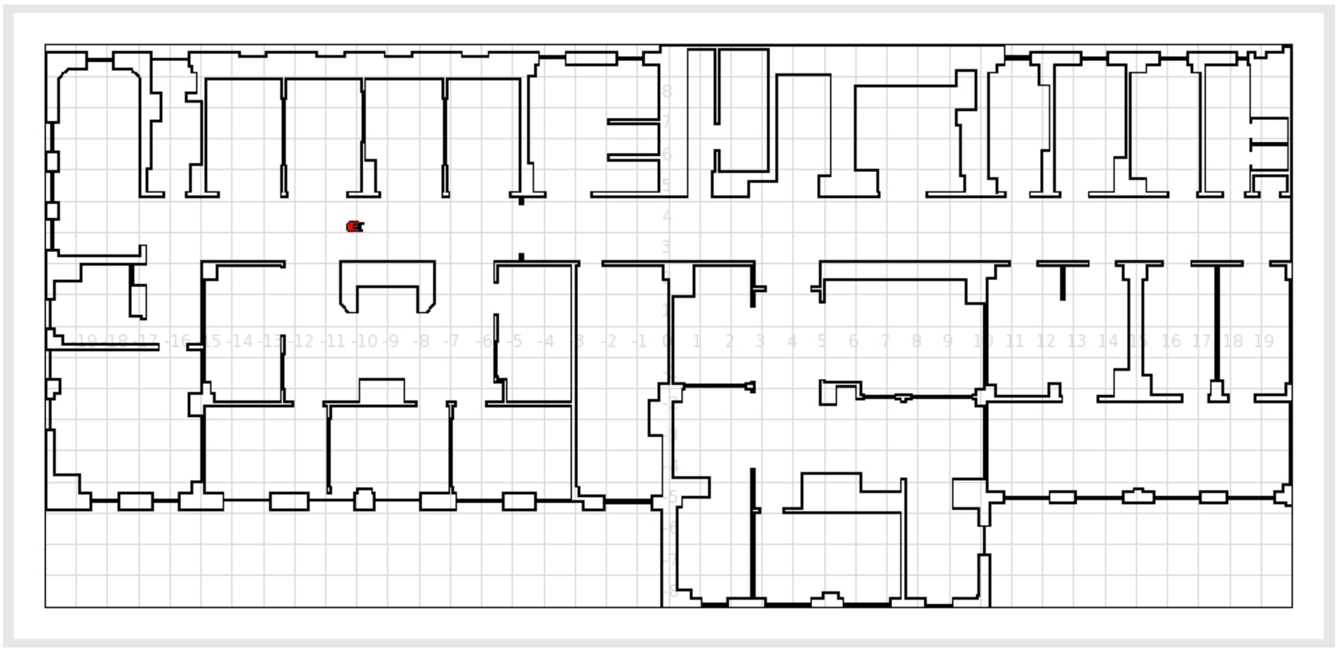
Wavefront path planner. Once you have your map in memory, and have grown the obstacles, you are ready to implement the path planner. Your path planner must make use of the wavefront path planning algorithm that we discussed in class on Sept. 30. You should start your wave propagation from the *goal* location, back to the start. (Later, after you get the single wave propagation working, you are welcome to implement a *dual wavefront* approach, where you propagate two waves from both the start and goal locations. However, this is not required. The point here is that if you want to implement the dual wavefront, that's terrific. But, if you don't, it isn't required; only the single wavefront is required.)

[*For Graduate Students*] Once you generate a path, you should smooth it using an approach described in class. The grid cells that remain on the robot path become waypoints. The output of your wavefront path planner should be this series of waypoints to be reached by the robot.

Note that if your path planner is given an unreachable goal position, it should state this fact and gracefully exit. A goal position could be unreachable because it is inside an obstacle, or it could be behind a wall, with no access. Your code should be smart enough to recognize this, and let the user know.

A note on doorways. A very tricky aspect of robot navigation in indoor environments is enabling the robot to move through doorways. You don't have a lot of room for error, and if your obstacle avoider is too sensitive, the robot may not be able to go through doorways, even though the robot should easily fit through the passage. This will be a tricky aspect of tuning your code so that the robot can follow the path generated by the path planner. Your code should work for any passage that is at least twice the width of the robot. Smaller passages may not work; you won't be penalized for that.

Setting up everything. In this exercise, you must use the bitmap called "hospital_section.png". The HW-6.cfg and HW-6.world files set up everything for you, so that when you start up player, your environment should look like the following:



Outputting the derived path: Your path planning code should also display the path generate by your planner on the Stage map. You can do this by making use of the Graphcs2dProxy. Here is some relevant code:

```
Graphics2dProxy gp(&robot, 0);
gp.Clear();
gp.Color(red);
player_point_2d plist[50];    //Will hold all your waypoints

//count = number of waypoints you generated
gp.DrawPolyline(plist,count);
```

Some documentation on the Graphics2dProxy is available here:
http://playerstage.sourceforge.net/doc/Player-2.0.0/player/classPlayerCc_1_1Graphics2dProxy.html

You'll be able to see the lines drawn in your Stage window if you select View->Data.

Testing your code: Your path planner should work for any goal position given; in the case of unreachable goal positions, as previously stated, your planner should recognize this and gracefully exit. Test your code for lots of different goal positions. Then, create screendumps of your robot's path to the following specific goal positions (i.e., after the robot follows the path generated by your wavefront planner):

- (7.5, 5.5)
- (8.5, -4.0)
- (-18, 7)
- (-9, -4)

WRITE UP THE FOLLOWING (written up in a single pdf file called *yourlastname*-HW-6.pdf):

a) A discussion of issues you ran into in getting your planner to work properly (about ½ page). Particularly, I'm interested in your experience with planning paths through doorways, and how you tuned your code to make it work. You should also briefly discuss any other issues you experienced related to the planner itself (i.e., don't write about trying to figure out how Stage works, or runtime errors, or figuring out the documentation, etc.). I only want planner-specific discussion.

b) 1 screenshot (or image) of your generated obstacle-grown map for the hospital environment.

c) 4 screenshots – one for each of the planned paths for the 4 goal points mentioned above. (Your robot does not have to follow the paths in this assignment; that is, your robot won't move.)

<u>SUBMITTING YOUR HOMEWORK:</u>

Place all your files in a single directory. These files should include:

- Your written answers to the questions above, plus screen dumps (images) as requested (i.e., the file "yourlastname-HW-6.pdf")
- Your configuration file, called "HW-6.cfg"
- Your world file, called "HW-6.world"
- Your makefile, called "makefile" or "Makefile"
- Your robot control code, called "yourlastname-HW-6.cc".
- Any additional include files you created (called whatever you want them to be called).

Remove all other unnecessary files, and tar or zip them up. Compress if needed. Submit to BlackBoard.

Homework 6 Appendix: Reading in Map Files

For this assignment, you need to read in the map file into your program's memory. Your path planner will use this map for finding the robot's path. Note that the bitmaps we have been using are stored in PNG format, which makes use of data compression techniques. While this is great for saving file space, it isn't the easiest format to use for path planning. So, we're going to convert the bitmap files to the PNM ("Portable Any Map") format, which is often used for exchanging between different graphics file formats. [Since the purpose of this class isn't to learn about graphics file formats, we're going to skip over the details. If you want to learn more about these file formats, there is lots of documentation on the web.]

Fortunately, we have handy utilities that can do this conversion for us. The utility we need is called, amazingly enough, "pngtopnm". This utility is so simple to use, that I'm going to let you convert the hospital_section.png file to the pnm version we'll call hospital_section.pnm. Simply issue the following linux command:

```
linux> pngtopnm hospital_section.png > hospital_section.pnm
```

The resulting PNM file has 3 header lines, followed by the pixels defining the map. You can safely ignore the first and third header lines in the PNM file. The 2nd header line has handy information on the width and height of the image, in pixels. The rest of the file has a character for each pixel, representing white pixels with –1 and black pixels with 0. We'll convert this to 0's for free space and 1's for obstacles in our grid map. So, to read in this map file, I've written a little utility for you. Creatively, it is called inputMap(). You can find the electronic source on the class web site, here:

http://web.eecs.utk.edu/~leparker/Courses/CS494-529-fall14/Homeworks/inputMap.cc

I've also attached a hard copy of the function at the end of this document.

Let me point out one more detail. Note that the map is scaled using the "SCALE_MAP" parameter. In this function, the effect of this parameter is to scale down the size of the map by a factor of 1/SCALE_MAP. This effectively sets the grid size of the map that your planner will use. To determine the size of the grid, you need to compare the metric size of the map (as defined in your world file) with the pixel size of the map (as defined in the PNM file). In this case, the metric size of the world map is 40 meters wide by 18 meters high. The size of the PNM file, in pixels, is 1086 wide by 443 high. We can therefore calculate the number of pixels per meter as 1086pixels/40m wide by 443pixels/18m high, giving us 27.15pixels/m and 24.61pixels/m. Inverting this, we get 3.68cm/pixel and 4.06 cm/pixel. We'll approximate this as 4 cm/pixel.

If we make one grid cell per pixel, then our grid resolution is 4x4 cm. This is probably a finer-grained resolution than we need. We can double the grid size to 8x8cm (thus halving the resolution). This seems like a reasonable resolution for our purposes. Thus, we scale the map by a factor of 2 (by setting SCALE_MAP to 2) in the inputMap() function.

The function is also set up to allow you to print out your scaled-down map to a separate file (called “scaled_hospital_section.pnm”). You can then use an image display program, such as *gimp*, to view the resulting file. You may find it handy to print out your map after you’ve grown obstacles. Then you can have a look at what your program has generated using the *gimp* utility. This should be helpful for debugging purposes.

```
#include <iostream>
#include <fstream>
#include <string>
using namespace std;

#define SCALE_MAP 2
#define GRID_ROWS 1000
#define GRID_COLS 1000

// You can change this to use dynamic memory, if you like.
float gridMap[GRID_ROWS][GRID_COLS];


/**********************************************************************
****
 *
*
 * Function inputMap converts the input map information into an
*
 * initial occupancy grid, which is stored in gridMap.
*
 *
 * (Here, the "output" parameter says whether to print out the scaled
*
 *  map; output = 1 ==> yes, print;  output = 0 ==> no, don't print.)
*

**********************************************************************
***/

void inputMap(int output)
{
    int i, j, m, n;
    char inputLine1[80], nextChar;
    int width, height, maxVal;

    ifstream inFile("hospital_section.pnm");

    /* Initialize map to 0's, meaning all free space */
    for (m=0; m<GRID_ROWS; m++)
        for (n=0; n<GRID_COLS; n++)
            gridMap[m][n] = 0.0;
    /* Read past first line */
    inFile.getline(inputLine1,80);

    /* Read in width, height, maxVal */
    inFile >> width >> height >> maxVal;
    cout << "Width = " << width << ", Height = " << height << endl;

    /* Read in map */
    for (i=0; i<height; i++)
        for (j=0; j<width; j++) {
       inFile >> nextChar;
       if (!nextChar)
         gridMap[i/SCALE_MAP][j/SCALE_MAP] = 1.0;
     }
    cout << "Map input complete.\n";

    if (output)  {
      ofstream outFile("scaled_hospital_section.pnm");
      outFile << inputLine1 << endl;
      outFile << width/SCALE_MAP << " " << height/SCALE_MAP << endl
           << maxVal << endl;

      for (i=0; i<height/SCALE_MAP; i++)
     for (j=0; j<width/SCALE_MAP; j++) {
       if (gridMap[i][j] == 1.0)
         outFile << (char) 0;
       else
         outFile << (char) -1;
     }
       cout << "Scaled map output to file.\n";
    }
}

int main(int argc, char *argv[])
{
  inputMap(1);  // Here, '1' means to print out the scaled map to a 
file;
                // If you don't want the printout, pass a parameter of 
'0'.
}
```