CIS 32 Spring 2009, Project IV

1 Description

This final project has three parts to it. You will work on one of them, and then put your program together with that of two other groups to complete the project.

Overall, this project will allow your robot to explore a space and then find its way around within the map. The three parts are:

1. Create a map.
2. Use the map to do path planning.
3. Follow a plan.

To help in this task I wrote some code that allows occupancy grid maps to be input from a file, printed and output to a file, and code that allows a plan to be read from a file, printed, and output to a file. At the end of this document you can find some notes on the format of the map and the plan in my code.

You should use the bits of this code that are relevant to your task — this will help to ensure that your code works with that of the other groups.

2 Build a map

The task is to write a robot controller, based on the code at the end of this document, that will create an occupancy grid of the world that the robot is operating in.

An occupancy grid is a simple form of map that takes the form (as the name suggests) of a grid. Each square of the grid corresponds to a small area of the world, and the grid indicates if any portion of that area of the world has any obstacle in it, or whether that portion of the work is free space that the robot can move through.

As an example of an occupancy grid, see Figure 1(b). This is a map of the world in Figure 1(a), and looking at the two side by side shows that the occupancy grid gives a rather coarse representation of the world, that is a representation that loses a good deal of detail.

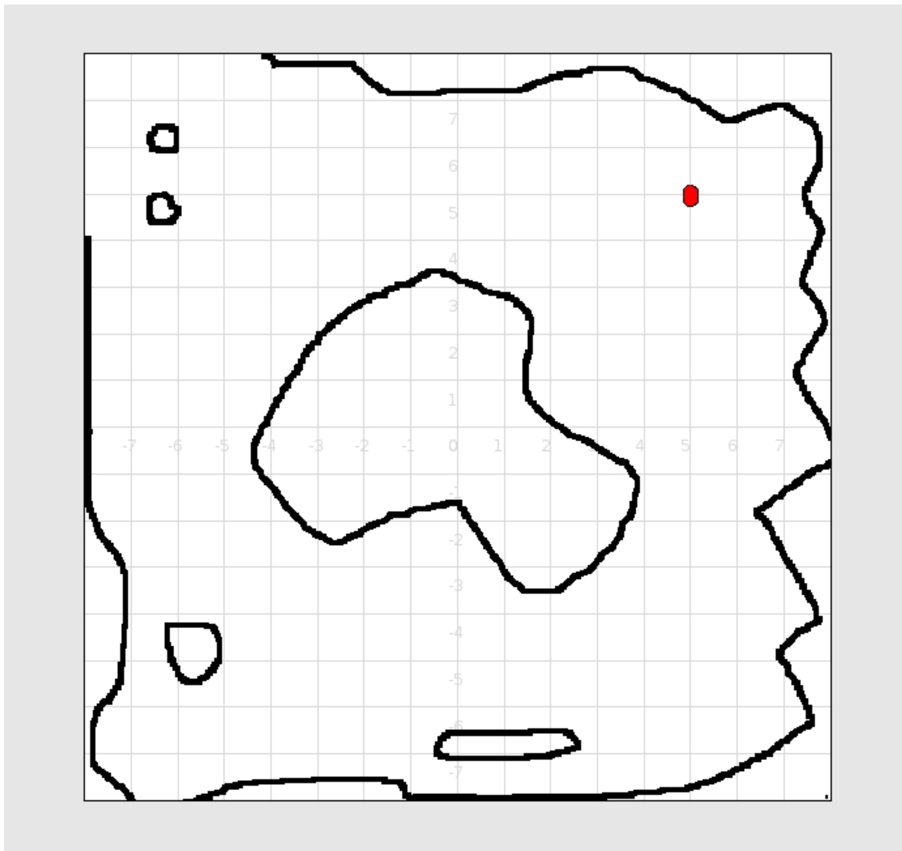
An occupancy grid should be represented in your code as an array of integers. Each element of the array represents one of the square of the grid, and the array should contain a 1 if that grid square is occupied, and a 0 otherwise.

To help with this task, you should use the code I provided to allow a robot controller to read an occupancy grid from a file, print an occuancy grid out on the screen, and write an occupancy grid to a file. Using this code will ensure that your code works with the code written by the groups who are creating a plan based on the map you create.

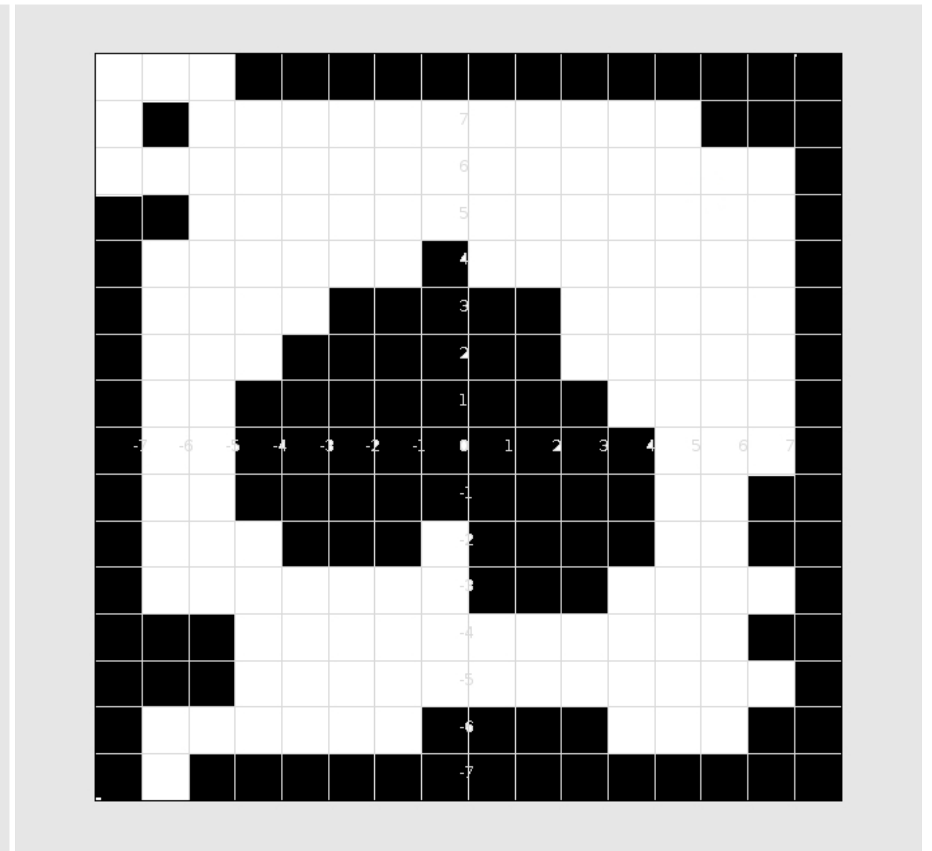
3 Use the map to plan

The task is to write a robot controller, based on the code at the end of this document, that can use an occupancy grid to plan a route through the world represented by the occupancy grid. This task assumes that you start with an occupancy grid, a set of coordinates that the robot starts at, and a set of coordinates that the robot has to end up at.

The task is then to generate a plan that is a sequence of sets of coordinates. The first of these sets of coordinates is the initial location of the robot, the last set of coordinates is the final location of the robot, and each intermediate point is a point that the robot should pass through on its way from the initial location to its final location.



(a) An example world in which the robot operates.



(b) An occupancy grid map.

Figure 1: A simulated world and the corresponding occupancy grid map

.

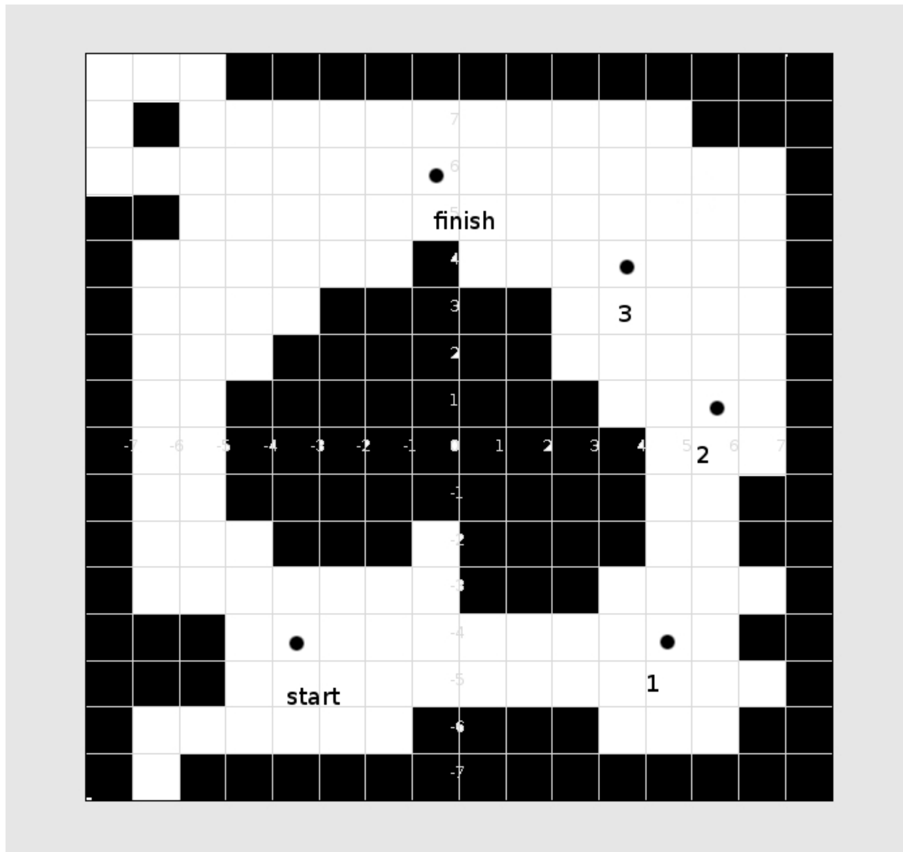


Figure 2: A plan for a robot

You need to choose the intermediate locations carefully so that the robot can move in a straight line from one to the next without encountering an obstacle.

An example plan, that works for a robot in the world illustrated in Figure 1(b) is given in Figure 2.

The code that you write to achieve this task should use the code I provided to read in an occupancy grid map, and also to output completed plan to a file. Using this code will ensure that your code works with the code written by the groups who are building the map and following the plan.

4 Follow a plan

The task is to take a plan that gives sequence of locations for the robot and make the robot move through this sequence of locations.

You should use the code I provided to read in a plan, and then, building on the code you wrote for Project 3, make the robot move from each location to the next, stopping when the robot gets close to the final location. While the plan that you are following should have been created in such a way that the robot can move in a straight line from one point to the next without hitting an obstacle, your code will have to include code for obstacle avoidance. (You never know when a faulty map of path-planning algorithm might have been used in creating the plan that your robot is following).

5 Map format

- An occupancy grid map is an array of integers.
- A 1 indicates that a sqaure is occupied, a 0 indicates that it is not occupied.
- I assume that the map is square, so there is only one parameter that controls the size of the map.
- In the example I gave you, each grid square is one meter on each side.
- The example map I gave you is for a map that is 16 meters on each side.

6 Plan format

- A plan is an integer followed by $2n$ doubles.
- The integer indicates how many coordinates there are in the plan (so it always has to be $2n$).
- Each pair of doubles is a set of coordinates, and x value and a y value, in that order..
- The coordinates in my example follow the coordinate system used by stage. This has the origin in the center, and has coordinates ranging from -8 to $+8$.

```
//------------------------------------------------------------------------
//
// pioneer-with-map.cc
//
// Example player libplayerc++ controller that controls a pioneer robot
//
// The robot controller is blank (as befits a program that is part of
// a class exercise, but it shows how to use the sonar proxy and the
// fake-localise feature of player/stage (which tells us where the
// robot is) and delivers commands to stage.
//
// The controller also has some functions that allow it to read and write
// occupancy grid maps, and read and write a navigation plan in the form of
// a set of x, y coordinates.
//
// Written by: Simon Parsons
// Modified  : 29th April 2009
//
// Usage:
//
// wall-follow [-h <host>] [-p <port>] [-i <index>] [-m]
//
//  -h <host> : connect to Player on this host
//  -p <port> : connect to Player on this TCP port
//  -i <index>: connect to this device (default 0)
//  -m        : turn on motors

//------------------------------------------------------------------------

#include <iostream>
#include <cmath>
#include <libplayerc++/playerc++.h>
#include <fstream>

#include "args.h" // Code that allows us to handle standard arguments

using namespace PlayerCc;

//------------------------------------------------------------------------
//
// Global constants

const int SIZE = 16; // The number of squares per side of the occupancy grid
                     // (which we assume to be square)

//------------------------------------------------------------------------
//
// Function headers

player_pose2d_t readPosition(LocalizeProxy&);
void printRobotData(player_pose2d_t, SonarProxy&);
double distanceToGoal(player_pose2d_t, double, double);
void readMap(int[SIZE][SIZE]);
void writeMap(int [SIZE][SIZE]);
```

```
void printMap(int [SIZE][SIZE]);
int  readPlanLength(void);
void readPlan(double *, int);
void printPlan(double *,int);
void writePlan(double *, int);

//-------------------------------------------------------------------
//
// main
//
// arg.h along with the arguments here handles command line arguments
// an provides the usage above.

int main(int argc, char *argv[])
{
  //-----------------------------------------------------------------
  //
  // Variables

  // For handling localization data

  player_pose2d_t          pose;

  // For setting speed

  player_pose2d speed;

  const  double stdSpeed = 0.4;
  double xSpeed;
  double ySpeed;
  double tSpeed;

  // Where we are going to:

  double goalX = -3.3;
  double goalY = -3.3;
  double distance;

  // The occupancy grid

  int oGrid[SIZE][SIZE];

  // The set of coordinates that makes up the plan

  int pLength;
  double *plan;

  //-----------------------------------------------------------------

  // Parse the arguments and set up the relevant variables.
  parse_args(argc,argv);

  // Use the arguments to connect to the specified device.
  PlayerClient robot (gHostname, gPort);
```

```
Position2dProxy pp (&robot, gIndex);
SonarProxy sp (&robot, gIndex);
LocalizeProxy lp (&robot, gIndex);

// Map handling
//
// The occupancy grid is a square array of integers, each side of
// which is SIZE elements, in which each element is either 1 or 0. A
// 1 indicates the square is occupied, an 0 indicates that it is
// free space.
readMap(oGrid);   // Read a map in from the file map.txt
printMap(oGrid);  // Print the map on the screen
writeMap(oGrid);  // Write a map out to the file map-out.txt

// Plan handling
//
// A plan is an integer, n, followed by n doubles (n has to be
// even). The first and second doubles are the initial x and y
// (respectively) coordinates of the robot, the third and fourth
// doubles give the first location that the robot should move to, and
// so on. The last pair of doubles give the point at which the robot
// should stop.
pLength = readPlanLength(); // Find out how long the plan is from plan.txt
plan = new double[pLength]; // Create enough space to store the plan
readPlan(plan, pLength);    // Read the plan from the file plan.txt.
printPlan(plan,pLength);    // Print the plan on the screen
writePlan(plan, pLength);   // Write the plan to the file plan-out.txt

// Control loop starts

while(!true)
{
  // Read values from the simulated robot. This reads new sensor
  // values into the sonar proxy sp, and new position data into the
  // localization proxy lp.
  robot.Read();

  // Load the position of the robot into the variable pose, which is
  // a struct with three parts --- x, y and angle.
  pose = readPosition(lp);

  // Print data on the robot to the terminal
  printRobotData(pose, sp);

  // How far are we from the goal?
  distance = distanceToGoal(pose, goalX, goalY);
  std::cout << "Distance to goal" << std::endl;
  std::cout << distance << std::endl;

  // Start with speed zero
  xSpeed = 0;
  ySpeed = 0;
  tSpeed = 0;
```

```
    //
    //
    // Write your controller in here
    //
    //

    // Setup speed values to send them to the robot.
    // speed.px is speed in the x direction
    // speed.py is speed in the y direction
    // speed.pa is rotational speed.
    speed.px = xSpeed;
    speed.py = ySpeed;
    speed.pa = tSpeed;

    // Now use those values to drive the robot
    //
    // This time around, setting ySpeed won't have any effect since the robot
    // is differential drive.
    pp.SetSpeed(speed);
  }

  // Control loop ends
}

//----------------------------------------------------------------------

// readPosition
//
// Read the position of the robot from the localization proxy.
//
// The localization proxy gives us a hypothesis, and from that we extract
// the mean, which is a pose.

player_pose2d_t readPosition(LocalizeProxy& lp)
{

  player_localize_hypoth_t hypothesis;
  player_pose2d_t          pose;
  uint32_t                 hCount;

  // Need some messing around to avoid a crash when the proxy is
  // starting up.

  hCount = lp.GetHypothCount();

  if(hCount > 0){
    hypothesis = lp.GetHypoth(0);
    pose       = hypothesis.mean;
  }

  return pose;
}

// printRobotData
```

```
//
// Print out data on the state of the sonar and the current location
// of the robot.

void printRobotData(player_pose2d_t pose, SonarProxy& sp)
{
  // Print out the sonar values. We can treat the sonar readings as
    // if they are an array of 16 values
    //
    // Left to right across the front.
    std::cout << "Sonar front" << std::endl;
    std::cout << sp[0] << std::endl << sp[1] << std::endl;
    std::cout << sp[2] << std::endl << sp[3] << std::endl;
    std::cout << sp[4] << std::endl << sp[5] << std::endl;
    std::cout << sp[6] << std::endl << sp[7] << std::endl << std::endl;
    //
    // Right to left across the back
    std::cout << "Sonar rear" << std::endl;
    std::cout << sp[8] << std::endl << sp[9] << std::endl;
    std::cout << sp[10] << std::endl << sp[11] << std::endl;
    std::cout << sp[12] << std::endl << sp[13] << std::endl;
    std::cout << sp[14] << std::endl << sp[15] << std::endl << std::endl;

    std::cout << "Current position" << std::endl;
    std::cout << "X: " << pose.px << std::endl;
    std::cout << "Y: " << pose.py << std::endl;
    std::cout << "A: " << pose.pa << std::endl;
}

// distanceToGoal
//
// Given the current pose and the goal coordinates, report the
// distance between the two.

double distanceToGoal(player_pose2d_t pose, double goalX, double goalY)
{
  double poseX = pose.px;
  double poseY = pose.py;
  double distance;

  distance = sqrt(pow((poseX - goalX), 2) + pow((poseY - goalY), 2));

  return distance;
}

// readMap
//
// Reads in the contents of the file map.txt into the array map
// in such a way that the first element of the last row of the
// file map.txt is in element [0][0].
//
// This means that whatever is in the file looks like the occupancy
// grid would if you drew it on paper.
```

```
void readMap(int map[SIZE][SIZE])
{
  std::ifstream mapFile;
  mapFile.open("map.txt");

  for(int i = SIZE - 1; i >= 0; i--){
    for(int j = 0; j < SIZE; j++)
      {
mapFile >> map[i][j];
      }
  }

  mapFile.close();
}

// printMap
//
// Print map[][] out on the screen. The first element to be printed
// is [SIZE][0] so that the result looks the same as the contents of
// map.txt

void printMap(int map[SIZE][SIZE])
{
  for(int i = SIZE -1; i >= 0; i--){
    for(int j = 0; j < SIZE; j++)
      {
std::cout << map[i][j] << " ";
      }
    std::cout << std::endl;
  }
}

// writeMap
//
// Write a map into map-out.txt in such a way that the [0][0] element
// ends up in the bottom left corner of the file (so that the contents
// of the file look like the relevant occupancy grid.

void writeMap(int map[SIZE][SIZE])
{
  std::ofstream mapFile;
  mapFile.open("map-out.txt");

  for(int i = SIZE - 1; i >= 0; i--){
    for(int j = 0; j < SIZE; j++)
      {
mapFile << map[i][j];
      }
    mapFile << std::endl;
  }

  mapFile.close();
}
```

```
// readPlanLength
//
// Open the file plan.txt and read the first element, which should be
// an even integer, and return it.

int readPlanLength(void)
{
  int length;

  std::ifstream planFile;
  planFile.open("plan.txt");

  planFile >> length;
  planFile.close();

  // Some minimal error checking
  if((length % 2) != 0){
    std::cout << "The plan has mismatched x and y coordinates" << std::endl;
    exit(1);
  }

  return length;
}

// readPlan
//
// Given the number of coordinates, read them in from plan.txt and put
// them in the array plan.

void readPlan(double *plan, int length)
{
  int skip;

  std::ifstream planFile;
  planFile.open("plan.txt");

  planFile >> skip;
  for(int i = 0; i < length; i++){
    planFile >> plan[i];
  }

  planFile.close();
}

// printPlan
//
// Print the plan on the screen, two coordinates to a line, x then y
// with a header to remind us which is which.

void printPlan(double *plan , int length)
{
  std::cout << "   x     y" << std::endl;
  for(int i = 0; i < length; i++){
    std::cout.width(5);
```

```
    std::cout << plan[i] << " ";
    if((i > 0) && ((i % 2) != 0)){
      std::cout << std::endl;
    }
  }
}

// writePlan
//
// Send the plan to the file plan-out.txt, preceeded by the information
// about how long it is.

void writePlan(double *plan , int length)
{
  std::ofstream planFile;
  planFile.open("plan-out.txt");

  planFile << length << " ";
  for(int i = 0; i < length; i++){
    planFile << plan[i] << " ";
  }

  planFile.close();
}
```