

CS26410 - Occupancy Grid Mapping Report

Samuel Jackson
slj11@aber.ac.uk

March 3, 2013

1 Player and Stage

The Player program together with the Stage plug-in provide a convenient controller and simulator pair for use in the field of robotics. Player acts as a wrapper to the hardware specifics of a robot and its sensors and provides an interface for an application to communicate with a physical robot. This is extremely useful, as it allows the developer to easily port code between different types and models of robots with different configurations and sensors without worrying too much about the underlying hardware. It also allows the developer to jump right into developing a robot controller, rather than dealing with code to interface with the system.

The Stage program is a plug-in for Player which is used to simulate a robot. Stage takes instructions from the Player application (which in turn is controlled by the developers code) and actuates a model of a robot within a simulated environment. In response, Stage also simulates sensor data relative to the position, speed, angle etc. of the robot and feeds it back into Player. This is useful as development of robot controllers can be made and testing without having access to a physical robot which can be both expensive and delicate. A simulation program such as Stage allows the core development of robot controllers on a smaller budget and without risk of damaging the robot through unforeseen bugs. The controllers can then be tested, tweaked and improved on a physical robot.

2 Occupancy Grid Mapping

Occupancy grid mapping is a robot mapping technique where the world is represented as a discrete number of equally sized, square chunks called cells. Each of these cells represents a position in a Cartesian coordinate system. Each of these cells can be labelled according to what a robot senses is at this position on the grid (such as if the cell is occupied by an object or not).

2.1 Representing the world

In my presented solution, I have split the world into a collection of cells which are 60x60 cm spanning from position (0,0) to an unspecified height and width using a 2D vector representation. Initially, only a grid of 6 x 6 cells are created, but as the robot explores the grid is able to dynamically resize to ensure that the robot never "falls off the map".

Resizing the only occurs when a measured point is found to lay outside of the existing grid. When resizing in a negative direction, we must also shift all data elements forwards (or up) in order to keep the grid correctly positioned relative to the robot's position.

Listing 1: C++ code for dynamically resizing the grid when a point (x,y) falls outside the current grid's size

```

if((x < 0 || y < 0) || (x >= grid_width || y >= grid_height)) {

    x_expand = (x < 0 || x >= grid_width) ? EXPANSION_SIZE : 0;
    y_expand = (y < 0 || y >= grid_height) ? EXPANSION_SIZE : 0;

    new_width = grid_width + x_expand;
    new_height = grid_height + y_expand;

    //resize grid to new dimensions
    grid.resize(h);
    for (int i = 0; i < h; ++i) {
        grid[i].resize(w);
    }

    //if we did a negative resize, shift data
    if(x < 0 || y < 0) {
        robot_x += (x_expand*MAP_SCALE);
        robot_y += (y_expand*MAP_SCALE);

        x += EXPANSION_SIZE;
        y += EXPANSION_SIZE;

        for (int i = (grid_width-1); i >= 0; i--) {
            for (int j = (grid_height-1); j >= 0; j--) {
                int value = GetCell(i,j);
                SetCell(i+x_expand,j+y_expand,value);
                SetCell(i,j,0);
            }
        }
    }

    grid_width = new_width;
    grid_height = new_height;
}

```

Real world values for the robots change in position are fed into the mapping system and the robots position in the grid is updated relative to its position in the grid's coordinate system rather than Stage's coordinate system. This is because we require the index of each of cells to be positive so they can be applied to the 2D vector.

2.2 Sensing the world

As the Pioneer robot moves though the world (or the simulator) it can sample data from each of it's sensors and feed the data into the mapping system for evaluation. The sonar sensors used by the Pioneer robots can only give us information on the range (between 0-5 metres) of how far away something is relative to the position of the robot.

However, as we have access to the position of the robot, the angle it's facing and the distance to the sensed object, simple trigonometry can be used to estimate the likely location of the sensed point.

$$x_{sensed} = x_{robot} + \cos(\theta) \times range$$

$$y_{sensed} = y_{robot} + \sin(\theta) \times range$$

Where x_{sensed} and y_{sensed} are the position of the sensed point, x_{robot} and y_{robot} are the robot's current position, θ is the range of the sensor is currently pointing relative to 0 and range is the range the sensor measured too.

Listing 2: C++ code used for calculating the new point given the current position, angle and range

```

void OccupancyGrid::SensorUpdate(double range, double angle) {
    double sensor_x, sensor_y;

```

```

if (range < MAX_RANGE) {
    //new point hit by sensor
    sensor_x = robot_x + (cos(angle) * range);
    sensor_y = robot_y + (sin(angle) * range);

    grid_x = ScaleToGrid(sensor_x);
    grid_y = ScaleToGrid(sensor_y);

    double max_grid_r = (MAX_RANGE/MAP_SCALE);
    double range_prob = (max_grid_r-range)/max_grid_r;

    SetCell(grid_x, grid_y, GetCell(grid_x, grid_y) + range_prob);
}
}

```

Once this new position has been obtained, we can scale and round it to the nearest cell in the grid and mark it as being occupied. In my controller, I am adding a value to the cell that represents the probability of the cell being occupied, which is inversely proportional to the range measured, as this cell is measured more and more and at closer ranges, the value increases proportionally to the certainty we have that the cell is occupied. Using this technique we can move through the world, make readings using the robot's sensors and build up a map of the world around us which could later be used for navigation.

In my example controller, I am simply using a random wander technique to move through the world and collect information about the system. This involves using simple reactive techniques to prevent the Pioneer from crashing into obstacles while still encouraging exploration of the environment. Below I provide a listing of the code I have used to get the robot moving around the simulated environment and avoiding collisions.

Listing 3: C++ code used to make control the robot to reactively wander through the environment

```

for (;;) {
    robot.Read();

    //do simple collision avoidance
    if(sp[3] < 0.6 || sp[4] < 0.6) {
        int direction = (sp[3]<sp[4]) ? -1 : 1;
    } else if((sp[0] + sp[1]) < (sp[6] + sp[7])) {
        turnrate = dtor(-10);
    } else {
        turnrate = dtor(10);
    }

    if(sp[3] < 0.6 || sp[4] < 0.6) {
        speed = 0;
    } else {
        speed = 0.150;
    }

    //command the motors
    pp.SetSpeed(speed, turnrate);
}

```

The above system works by trying to keep the robot a balanced distance between the side a further front sensors. If the distance on one side becomes to large then it turns back towards the other direction. If the distance in front of the robot becomes too small then it reacts by turning on the spot until it has enough room to move off forward again. The robot will continue mapping in this way indefinitely.

Once we have mapped the environment, we can begin to analyse the quality of the map sensed and use a threshold to remove any values that appear to be incorrectly identified as walls and obstacles (more on why this occurs in section 2.3)

2.3 Issues and Potential Improvements