**Mobile Robotics Term Project Design Report and Documentation**

**Grid-map Construction**

First, we construct a grid for the given map. Essentially, we iterated through the entire map as if it were a 2D array with a certain precision. A precision of 5 for example, means that we iterate through the map using 0.2 unit distance increments (0.0, 0.2, 0.4, etc.). At every point, we check to see if it the section of the map is either inside of an obstacle, on top of an obstacle, or “near” an obstacle.

Since the robot is a circle, which means the distance from the centre of the robot to any point on the surface is the same (radius), we can condense the robot down to a single point. To compensate for this, we must “inflate” the obstacles. The radius of the robot in our case is 1cm, so to be safe, we gave a padding of 1.2cm to all the obstacles. This is what we meant by “near” an obstacle. We chose 0.2cm as the tolerance, because the stateGoal has a tolerance of ±0.2cm, so we decided that it is also an appropriate tolerance for the obstacle padding.

**Path-Planning Algorithm**

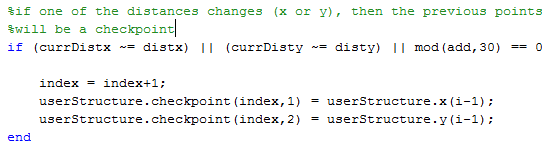
After we have our grid (made up of valid points that the robot can traverse), we construct our path from it. We use a variant of the Breadth First Search algorithm. Typically, a wave front algorithm is applied to a map, which simultaneously forms a grid-based map and assigns each grid a cost value. For us, we have decided to do the BFS and cost calculation simultaneously, as we believe it makes the code easier to understand and maintain, without the downside of consuming too much memory or slowing down the run time. Essentially, we implement a priority queue that keeps track of the current “least expensive path.” The BFS and path construction will continue until the goal is reached.

The cost of the path is determined by the summation of all the straight and diagonal movements. A straight movement (horizontal or vertical) costs 1 unit. We have decided on a heuristic of 1.5 for the diagonal cost. It makes no sense for diagonal to be lower than 1, since diagonal is longer than horizontal or vertical. It also doesn’t make sense for diagonal to be greater than 2, since diagonal movements would never be used in that case.

**Checkpoint Construction**

We use the path points stored in userStructure.x and userStructure.y to construct a set of checkpoints. The check points are helpful because now the robot knows exactly where to go and where to stop, and can afford to go at full speed without fear of veering off course or running into obstacles, thus greatly improving the time performance of the simulation. A point on the path is considered a checkpoint if the robot needs to turn (change direction) at that point. The startPoint and stateGoal must be checkpoints by default. We check for whether a point qualifies as a checkpoint by making a list of “direction” coordinates. We compare the vector between 2 consecutive points (by iterating through userStructure.x and userStructure.y). Whenever there is a change, we mark the current point as a check point. Also, to keep the robot from getting “overconfident,” we are also going to mark a checkpoint once every 30 units, so that if there is a long straight path, the robot will got go off trajectory.

For example, imagine we are going directly east. The vector is d=(x, y)=(1,0). Then, we decide to go north-east, which means d=(x,y)=(1,1). Since the vector changed from (1,0) to (1,1), that point will be marked as a checkpoint. Essentially, there should never be a checkpoint when the robot travels in a straight path.



**Motion Control Design**

We considered two approaches.

1. Imagine that the robot wants to go from point A to point B. We accelerate the robot with u=[a a]. When the robot reaches the midpoint, or (A+B)/2, we accelerate with u=[-a -a]. When the robot is close to B, the acceleration should be close to zero. To make sure this is the case, and to get rid of any precision errors, we simply set the acceleration to be proportional to the velocity. The exact same method is applied to turning. If the robot wants to turn by π radians, it would accelerate, then begin decelerating when it reaches π/2 radians. This is a simplified form of the proportional controller. However, due to the simplicity of this design, we must move and turn the robot separately, and the simultaneous action can throw the robot off its trajectory. Also, if the robot goes too far off trajectory, there isn’t an effective way to bring it back on track. As a result, we have decided to go with an actual PID controller.

2. We implemented a PD controller that uses a Kp value of… A Kp value that is too low results in very slow movement, and a Kp value that is too high results in instability in the form of overshooting, which occasionally causes the robot to crash into an obstacle. After much testing, we arrived at Kp values of 0.45 (for translation) and 1.75 (for turning), and a Kd value of 1. We did not bother implementing the integral controller because completely zeroing out the error takes too long for it to be viable. Also, occasionally the integral controller would cause the system to overshoot if the error ever goes from positive to negative, or from negative to positive.

