Homework 4

MacMillan, KyleA

October 26, 2018

Contents

List of Figures

1 Problem 5

1.1 Problem 5.2

The meat and potatoes of this problem is in bump and gps callbacks, the rest is filler but can be found here. The environment used is shown in Figure 1. The robot attempts to go to a point such as $(0, 20)$ which it cannot get to from within the box. In the homework submission a video was attached showing the basic motion bot attempting to reach the point $(0, 20)$. The box center is at $(0, 0)$ and has a height/width of 10. So the robot's goal is outside the box.

```
def bumpCallback(self, msg):
    hit_obj = False
    for i in range(len(msg.data)):
        hit = unpack('b', msg.data[i])[0]
        if hit != 0:
            hit_obj = True

    if hit_obj:
        # Turn right
        self.bumps += 1
        self.setVel(0.0, 2.0)
    else:
        self.bumps = 0
        self.setVel(2.0, 2.0)


def gpsCallback(self, msg):
    # we need to move to goal if we are not bumping a wall
    if self.bumps == 0:
        # Wraps deals with the robot if it spins around somehow
        wraps = np.abs(int(msg.theta / (2 * np.pi)))
        theta = np.fabs(msg.theta) - (wraps * 2 * np.pi)
        beta = np.arctan2(self.goal[1] -
                          msg.y, self.goal[0] -
                          msg.x)
        k = 0.25
        alpha = beta - theta
        w1 = 2.0 + k * alpha
        w2 = 2.0 - k * alpha
        self.setVel(w1, w2)
```

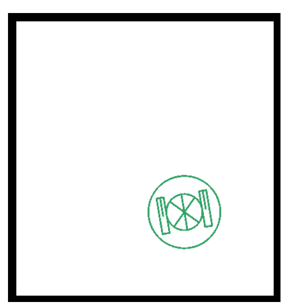

Figure 1: Problem 5.2 Stuck Robot

1.2 Problem 5.10a

The video of this problem is located here.

1.3 Problem 5.10b

The video of this problem is located here.

2 Problem 9

2.1 Problem 9.1

Figure 2 shows the required plot. The estimated robot location is:
$x = 882.079427083333$
$y = 443.766927083333$
$z = 521.18359375$

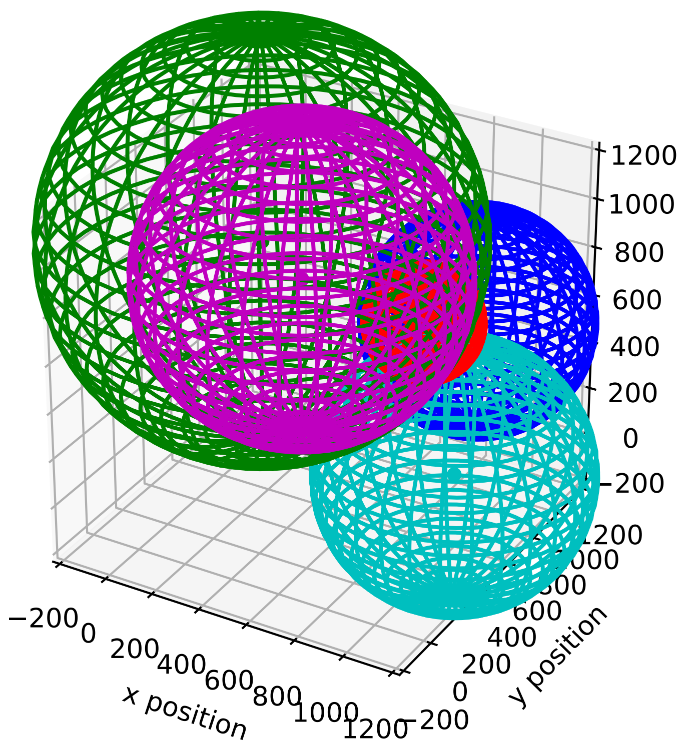

Figure 2: Problem 9.1

This was obtained with a BFS octant descent. Each octant containing an intersection with all beacon radii is added to a queue to be processed. If an octant is processed it will be chopped up into 8 more octants and all of those subdivisions will be processed. After we have either hit maximum depth (capped at $2 * log_2(max\_length)$) or we fail to find any octants containing all beacon radii, the queue will empty. After the queue empties we find the maximum depth obtained and find all points that made it to that depth. The centroid of those points is the likely location of the robot.

```python
def findRobot(self):
    best_pts = []
    best_depth = []
    self.q.put(((self.a, self.b), 0))
    while not self.q.empty():
        pts, depth = self.q.get()
        self.checkGrids(pts, depth)
        best_pts.append(pts)
        best_depth.append(depth)

    centroid = Point()

    # Find the index that made it deepest and calculate the centroid
    idx = best_depth.index(max(best_depth))
    count = 0
    for i in range(len(best_depth)):
        if best_depth[i] == best_depth[idx]:
            centroid += best_pts[i][0].midpoint(best_pts[i][1])
            count += 1

    if centroid != Point(0, 0, 0):
        centroid.x = centroid.x / count
        centroid.y = centroid.y / count
        centroid.z = centroid.z / count

        self.robot_loc = centroid
        print('\nRobot at: {}'.format(str(self.robot_loc)))
    else:
        print('\nRobot not found!')
```

```python
def checkGrids(self, pts, depth):
    # For clarity:
    a = pts[0]
    b = pts[1]
    octants = 8

    grid = [0] * octants
    x_mid = ((b.x + a.x) / 2.0)
    y_mid = ((b.y + a.y) / 2.0)
    z_mid = ((b.z + a.z) / 2.0)
    ab_list = []

    # The 8 octants of pts a & b
    ab_list.append((a, Point(x_mid, y_mid, z_mid)))
    ab_list.append((Point(a.x, y_mid, a.z), Point(x_mid, b.y, z_mid)))
    ab_list.append((Point(x_mid, a.y, a.z), Point(b.x, y_mid, z_mid)))
    ab_list.append((Point(x_mid, y_mid, a.z), Point(b.x, b.y, z_mid)))

    ab_list.append((Point(a.x, a.y, z_mid), Point(x_mid, y_mid, b.z)))
    ab_list.append((Point(a.x, y_mid, z_mid), Point(x_mid, b.y, b.z)))
    ab_list.append((Point(x_mid, a.y, z_mid), Point(b.x, y_mid, b.z)))
    ab_list.append((Point(x_mid, y_mid, z_mid), b))

    # Fill grid with hits
    for b in self.beacons:
        # Check each block
        for i in range(octants):
            if b.onBlock(ab_list[i][0], ab_list[i][1]):
                grid[i] += 1
    found = 0
    for i in range(self.b_len):
        if grid[i] == self.b_len:
            found += 1

    if found == octants:
        return

    for i in range(octants):
        if grid[i] == self.b_len and depth < self.max_depth:
            self.q.put((ab_list[i], depth + 1))
```

2.2 Problem 9.2

$\lambda = c * 10MHz$
$\lambda = 30\ meters$

Assuming phase shift $\theta = 10$ we can plug that into our formula to get

$$D' = L + \frac{\theta}{2\pi}\lambda$$

Therefore $D = \frac{D'}{2} = 0.833333333 + 15k$ where k denotes an integer interval. We make the assumption that L is arbitrarily small compared to the distance travel and is therefore set to 0. If the system has noise we will have to identify a range for $\frac{D'}{2}$, in this case it's $0.825\ to\ 0.841666667 + 15k$. In order to differentiate between 20 and 250 meters we would need a second system at a λ multiple that doesn't overlap before a distance of 250 meters.

3 Problem 10

3.1 Problem 10.1

$f = 0.8cm$
$b = 30cm$
$a = tan^{-}1(\frac{z}{b-x})$
$u = \frac{fx}{z}$

Given the above formulas we can say a is in the range of: $45 < a < 90$ and for u: $3 < u < 45$.

3.2 Probelm 10.2

$e = 10\%$
$v_1 = 0.2cm$
$v_2 = 0.3cm$
$z = \frac{fb}{v_1+v_2}$

Given the above formulas we can say that $f * b = 0.7 * 10 = 7$ but the range of z is dependent on $v_1 + v_2$, or: $\frac{7}{0.18+0.27} <= z <= \frac{7}{0.22+0.33}$

With zero error we would expect $z = 14$, on the low end we expect $z = 12.72$ with an error of 9.0909% and on the upper end we expect $z = 15.56$, leaving an error of 11.11111%.