Name: Zeyad Shureih	

Names of people you worked with:
Miguel Ruiz

Websites you used:

Approximately how many hours did it take you to complete this assignment (to nearest whole number)?	72? I lost track

The Rules: Everything you do for this lab should be your own (or partner's) work. You can look up general information on the web, but no copying code you find there. Read the code, close the browser, then write your own code.

By writing or typing your name below you affirm that all of the work contained herein is your own, and was not copied or copied and altered.

Note: Failure to sign this page will result in a 50 percent penalty. Failure to list people you worked with may result in no grade for this lab. Failure to fill out hours approximation will result in a 10-percent penalty.

Turn this file into Gradescope (Final project). Turn in all ROS files to gradescope (Final project code).

BEFORE YOU BEGIN

1) Make sure you can load the slam map from the last assignment.

Learning Objectives:

You should be able to do the following:

Use A* to plan a path between two points in the map
Use your Lab 2 code to move the robot along the path
Use a global planning to determine where to visit next
Use ROS's slam to build up a map that you explore

Lab Guidelines:

1) There is no "right" answer to these problems
2) We will provide other maps to test your robot planner on… Some of them are not as "nice" as the one you've been using.

The instructions for this lab are deliberately more vague than the labs.

Part 1: A* in ROS

For this part, just load the map you already made. Hard-code a point on the map to plan a path to. Draw that path in RViz.

Step 1: Copy/move your code over to your ROS install and put in the slam_mapping directory.

Step 2: After loading the map, figure out where the robot is (which pixel) and plan a path to your pre-defined location.

Step 3: Draw the path as a set of markers in RViz.

This requires four things: How to map from pixels to the robot coordinates (and back), how to draw markers, and how to convert the occupancy grid message to an array similar to an image array.

An example of how to plot points in Rviz (which you should read/use) is in the file final_project/src/points_to_rviz.py in the git repository (https://github.com/OSUrobotics/ROB456and514Intros)

RViz image with path

```
% Copy and paste your code here
def mark_path_to_goal(self):
        if self.odom_pos == None or self.goal_pos == None or
self.map_data == None:
            return

        points = self.get_path()
        # convert points back into rviz units
        points = [(self.x_array_to_rviz(b),
self.y_array_to_rviz(a))
                  for a, b in points]

        new_marker = Marker()
        # Marker header specifies what (and when) it is drawn
relative to
        new_marker.header.frame_id = "map"
        new_marker.header.stamp = rospy.Time.now()
        # uint8 POINTS=8
        new_marker.type = 8
        # Disappear after 1sec. Comment this line out to make them
persist indefinitely
```

```
        # new_marker.lifetime = rospy.rostime.Duration(1, 0)
        # Set marker visual properties
        new_marker.color.b = 1.0
        new_marker.color.a = 1.0
        new_marker.scale.x = 0.1
        new_marker.scale.y = 0.1
        for (x, y) in points:
            # set current point
            p = Point()
            p.x = x
            p.y = y
            p.z = 0.1
            new_marker.points.append(p)
        self.mark_pub.publish(new_marker)

    def get_path(self):
        pos = self.odom_pos[:2]
        goal = self.goal_pos
        map_grid = self.map_data

        map_w = map_grid.info.width
        map_h = map_grid.info.height
        map_res = map_grid.info.resolution

        # convert current pos into map_grid coordinates
        current_y = self.x_rviz_to_array(pos[0])
        current_x = self.y_rviz_to_array(pos[1])

        # convert goal pos into map_grid coordinates
        goal_y = self.x_rviz_to_array(goal[0])
        goal_x = self.y_rviz_to_array(goal[1])

        # build cost map (flood fill)
        self.cost_map = np.full(self.map_array.shape,
fill_value=np.inf)
        if self.flood_fill(current_x, current_y, goal_x, goal_y):
            # get path
            return self.find_path(current_x, current_y, goal_x,
goal_y)
        else:
```

```
            return []

    def near_wall(self, img, x, y):
        width = 5

        start = np.array([x, y])
        start_x = x - width
        start_y = y - width

        end_x = x + width
        end_y = y + width

        for i in range(start_x, end_x):
            for j in range(start_y, end_y):
                curr = np.array([i, j])
                if np.linalg.norm(start - curr) <= width:  # circle
radius width
                    if img[i][j] == 100:
                        return True

        # print("not near wall")
        return False

    def find_path(self, x_0, y_0, x_1, y_1):
        cost_map = self.cost_map
        img = np.flipud(self.map_array.copy())

        # work backwards towards goal
        node = cost_map[x_1][y_1]  # node = distance
        c_x = x_1
        c_y = y_1

        path = [(c_x, c_y)]
        while node:
            print("{}, {}".format(c_x, c_y), node)
            # get neighbors
            neighbors = self.get_neighbors(img, c_x, c_y)
            print(neighbors)

            # find the neighbor with the lowest distance value
```

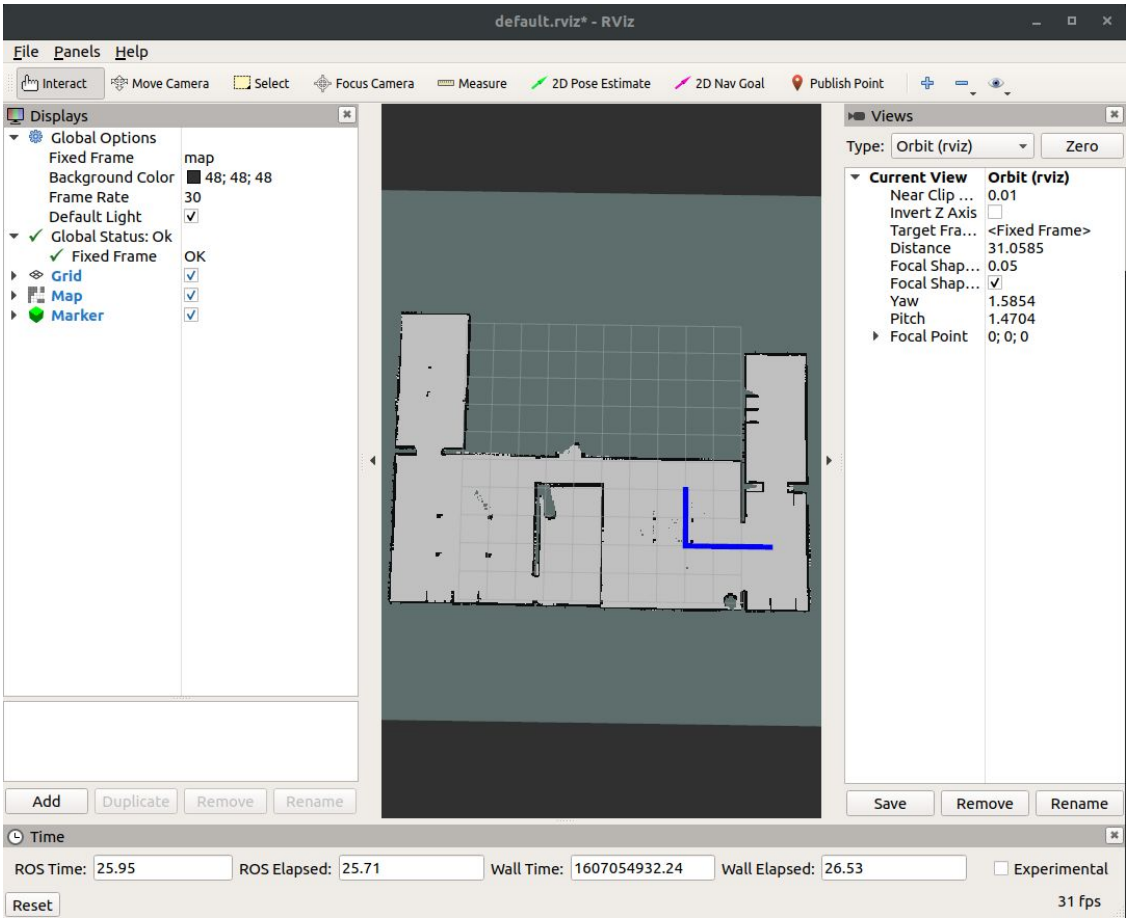
```
            # shortest_distance = np.inf
            for neighbor in neighbors:
                n_x = neighbor[0]
                n_y = neighbor[1]

                if cost_map[n_x][n_y] <= node and not (n_x, n_y) in
path:
                    c_x = n_x
                    c_y = n_y
                    node = cost_map[n_x][n_y]

            path.append((c_x, c_y))

            if node == 0:
                print("we found the path")
                path.append((x_0, y_0))
                return path

        print("no path")
        return []

    def x_rviz_to_array(self, x_pos):
        x = round((x_pos - self.map_data.info.origin.position.x) /
                  self.map_data.info.resolution)
        return int(x)

    def y_rviz_to_array(self, y_pos):
        y = round(self.map_data.info.height - ((y_pos -

self.map_data.info.origin.position.y) /
self.map_data.info.resolution))
        return int(y)

    def x_array_to_rviz(self, x_pos):
        x = (x_pos * self.map_data.info.resolution) + \
            self.map_data.info.origin.position.x
        return x

    def y_array_to_rviz(self, y_pos):
        y = (-1 * self.map_data.info.resolution * (y_pos -
```

```
self.map_data.info.height)) + self.map_data.info.origin.position.y
        return y

    def flood_fill(self, x, y, final_x, final_y):
        map_img = np.flipud(self.map_array.copy())
        unknown_spaces = np.where(map_img == -1)  # -1 means
unknown
        map_img[unknown_spaces] = 100  # turn everything unknown
into a wall
        # plt.imshow(map_img, cmap='gray', vmin=-1, vmax=100)
        # plt.show(True)
        # quit()

        nodes = [(0, (x, y))]
        heapify(nodes)

        while nodes:
            # pop the node
            node = heapq.heappop(nodes)
            print(node)
            # get its values
            cur_p = node[0]
            cur_x, cur_y = node[1]

            # update the map object
            self.cost_map[cur_x][cur_y] = cur_p

            # # if the current node is near the goal, but goal is
near a wall, end early
            # if np.linalg.norm(np.array([cur_x, cur_y]) -
np.array([final_x, final_y])) <= 5 and self.near_wall(map_img,
final_x, final_y):
            #     # set the end of the path to the current node
            #     print("found shortened path")
            #     return True

            # get neighbors
            neighbors = self.get_neighbors(map_img, cur_x, cur_y)
```

```
            # already skipping occupied spaces
            for neighbor in neighbors:
                n_x = neighbor[0]
                n_y = neighbor[1]
                n = np.array([n_x, n_y])

                # make the path distance the priority
                new_p = (cur_p + np.linalg.norm(np.array([cur_x,
cur_y]) - n))

                if (n_x, n_y) == (final_x, final_y):
                    self.cost_map[n_x][n_y] = new_p
                    print("found the goal")
                    return True

                # if self.near_wall(map_img, n_x, n_y):
                #     new_p *= new_p  # square the distance to
disuade path's near walls

                # if the block is already seen, possibly update
priority
                if self.cost_map[n_x][n_y] != np.inf:
                    # get the p value that matches the node in the
heap
                    old_p = [a for a, b in nodes if (n_x, n_y) ==
b]
                    if len(old_p) > 1:
                        print("Error - quiting")
                        quit()

                    if new_p < old_p:  # if new distance is shorter
than before, update it
                        idx = nodes.index((old_p[0], (n_x, n_y)))
                        nodes[idx] = (new_p, (n_x, n_y))
                        heapify(nodes)  # rebalance the heap
                else:
                    self.cost_map[n_x][n_y] = new_p
                    heapq.heappush(nodes, (new_p, (n_x, n_y)))

            # gray_max = np.max(self.cost_map)
```

```
            # plt.imsave("flood.png", self.cost_map,
            #            cmap='gray', vmin=0, vmax=300)
            # plt.show()
        print("could not find goal")
        return False

    def is_wall(self, image, row, column):
        if image[row][column] == 100:
            return True
        else:
            return False

    def get_neighbors(self, image, row, column):
        ret_pixels = []
        #get all immediate neighbors not near a wall (include
diagnols)

        if not self.near_wall(image, row - 1, column):
            ret_pixels.append((row - 1, column))

        if not self.near_wall(image, row + 1, column):
            ret_pixels.append((row + 1, column))

        if not self.near_wall(image, row, column - 1):
            ret_pixels.append((row, column - 1))

        if not self.near_wall(image, row, column + 1):
            ret_pixels.append((row, column + 1))

        if not self.near_wall(image, row - 1, column - 1):
            ret_pixels.append((row - 1, column - 1))

        if not self.near_wall(image, row - 1, column + 1):
            ret_pixels.append((row - 1, column + 1))

        if not self.near_wall(image, row + 1, column - 1):
            ret_pixels.append((row + 1, column - 1))

        if not self.near_wall(image, row + 1, column + 1):
            ret_pixels.append((row + 1, column + 1))
```

```
    return ret_pixels
```



Part 2: Moving along the path

For this part, just load the map you already made. Hard-code a point on the map to plan a path to. Draw that path in RViz. Move the robot along that path.

Step 1: Add your go_around package to slam_mapping so you can use that code.

Step 2: Decide how far along the A* path you want to travel each time (find the waypoints along the path). Use those to generate target goals for your go_around code.

Step 3: Write a bit of code to handle if you *don't* get where you wanted to go. When do you want to give up and re-plan?

This requires some message/topics setup to handle when you're sufficiently close to one waypoint to start going to the next.

There is no "right" answer to decide how to chop up your path into waypoints. Longer paths are better because they result in smoother robot motion, but the robot might have trouble with obstacles and turning corners/doors.

Algorithm for generating waypoints/when to replan

```
% Summarize your algorithm choices here (in English)
Generating waypoints (subgoals):
      for each point in path
           if the point 2 points back is diagonal to the current point,
      mark it as a corner and save it as a waypoint

when to replan:
      during each map/odom update
      if we are at subgoal, set next goal/subgoal and reset timer
      if we are not at subgoal, and timer is greater than 60 seconds,
      replan and reset timer
      If there are no subgoals, and we are not at goal, check timer
      if timer is greater than 60 seconds, replan and reset timer
```

Code

```
% Copy and paste the relevant code here
REPLAN:
if self.sub_goal != None:
            # check to see if we are at subgoal
            if self.is_at_location(self.sub_goal[0],
self.sub_goal[1]):
                print("at sub_goal")
```

```
                # set sub goal to next subgoal
                if len(self.sub_goals) != 0:
                    self.sub_goal = self.sub_goals.pop()
                    self.sub_goal_timer = rospy.Time.now()
                    print(self.sub_goal)
                else:
                    self.sub_goal = None
                    self.sub_goal_timer = rospy.Time.now()
            else:
                # if we aren't at sub goal, and it has been more
than 30 seconds
                current_time = rospy.Time.now()
                if current_time.secs - self.sub_goal_timer.secs >
60:
                    self.move_block = True
                    self.mark_path_to_goal()
                    if len(self.sub_goals) > 0:
                        self.sub_goal = self.sub_goals.pop()
                        self.sub_goal_timer = rospy.Time.now()
        else:
            # check to see if we are at goal
            if self.is_at_location(self.goal_pos[0],
self.goal_pos[1]):
                self.move_block = True # stop moving indefinitely
                print("at goal")
            else:
                current_time = rospy.Time.now()
                if current_time.secs - self.sub_goal_timer.secs >
60:
                    self.move_block = True
                    self.mark_path_to_goal()
                    if len(self.sub_goals) > 0:
                        self.sub_goal = self.sub_goals.pop()
                        self.sub_goal_timer = rospy.Time.now()
```

GENERATING WAYPOINTS

```
def find_path(self, x_0, y_0, x_1, y_1):
        print("In find_path")
        cost_map = self.cost_map
        img = np.flipud(self.map_array.copy())
```

```
        # work backwards towards goal
        node = cost_map[x_1][y_1] #node = distance
        c_x = x_1
        c_y = y_1

        path = [(c_x, c_y)]
        corners = []
        while node:
            # print("{}, {}".format(c_x, c_y), node)
            # get neighbors
            _open, walls = self.get_neighbors(img, c_x, c_y, 1)
            neighbors = _open + walls
            # print(neighbors)

            # find the neighbor with the lowest distance value
            # shortest_distance = np.inf
            for neighbor in neighbors:
                n_x = neighbor[0]
                n_y = neighbor[1]

                if cost_map[n_x][n_y] <= node and not (n_x, n_y) in
path:
                    c_x = n_x
                    c_y = n_y
                    node = cost_map[n_x][n_y]

            if len(path) > 3:
                prev_point = path[-2]
                if prev_point[0] != c_x and prev_point[1] != c_y:
                    corners.append((c_x, c_y))

            path.append((c_x, c_y))

            if node == 0:
                print("we found the path")

                # corners.reverse()
                self.sub_goals = [(self.x_array_to_rviz(
```

```
                    b), self.y_array_to_rviz(a)) for a, b in
corners]

                path.append((x_0, y_0))
                return path

        print("no path")
        return []
```

Part 3: Global planning

For this assignment, you're going to start from scratch with the robot in an unknown map and "explore" until the robot has filled out the entire map. Again, no "right" answer – there are several ways to pick the next point to path plan to.

To guarantee that you visit all possible places, you DO need to keep going until you have no "new" places to go to. New places are those pixels that you've seen that are adjacent to pixels that are "unknown" (not walls, not seen pixels).

To implement this you'll again need some message passing code, specifically, you'll need to re-generate the slam map as you wander around AND you need an outer loop that – when you've finished the last A* path – generates a new one or quits and says you've visited everything.

How you're picking the next place to visit (in English)

If we have no goal, flood fill until we find nearest unmapped point
navigate to subgoals/goal until we are at the point, and the point is mapped
repeat

Relevant code

```
% Copy and paste your code here
def flood_fill(self, x, y):
        print("In Flood Fill")
        map_img = np.flipud(self.map_array.copy())

        nodes = [(0, (x, y))]
        heapify(nodes)

        while nodes:
            # pop the node
            node = heapq.heappop(nodes)
            # print(node)
            # get its values
            cur_p = node[0]
            cur_x, cur_y = node[1]

            # update the map object
            self.cost_map[cur_x][cur_y] = cur_p
```

```
            # if node is unknown
            if map_img[cur_x][cur_y] == -1:
                self.goal_pos = (self.x_array_to_rviz(cur_y),
self.y_array_to_rviz(cur_x))
                print("found nearest unknown space")
                return True

            # get neighbors
            open_pixels, near_walls = self.get_neighbors(map_img,
cur_x, cur_y, 6)
            neighbors = open_pixels + near_walls

            # already skipping occupied spaces
            for neighbor in neighbors:
                n_x = neighbor[0]
                n_y = neighbor[1]
                n = np.array([n_x, n_y])

                # make the path distance the priority
                new_p = (cur_p + np.linalg.norm(np.array([cur_x,
cur_y]) - n))

                # print(idx)
                idx = [(a, b) for a, b in near_walls if a == n_x
and b == n_y]
                if len(idx) > 0:
                    new_p *= 100

                # if the block is already seen, possibly update
priority
                if self.cost_map[n_x][n_y] != np.inf:
                    # get the p value that matches the node in the
heap
                    old_p = [a for a, b in nodes if (n_x, n_y) ==
b]
                    if len(old_p) > 1:
                        print("Error - quiting")
                        quit()
```

```
                    if new_p < old_p:  # if new distance is shorter
than before, update it
                        idx = nodes.index((old_p[0], (n_x, n_y)))
                        nodes[idx] = (new_p, (n_x, n_y))
                        heapify(nodes)  # rebalance the heap
                else:
                    self.cost_map[n_x][n_y] = new_p
                    heapq.heappush(nodes, (new_p, (n_x, n_y)))

            # plt.imsave("flood.png", self.cost_map, cmap='gray',
vmin=0, vmax=300)
            # plt.show()
        print("could not find goal")
        return False
```

Resulting map

Copy your final slam map here - maze_closed world (as seen in our final report)