AA 274: Principles of Robot Autonomy
Problem Set 4: Filtering, Localization, and Mapping

Problem 1: EKF Localization

viii)
By driving around the TurtleBot, the open loop and EKF state estimates are converged at first. But it starts to diverge when the robot hits the wall. The collision between the robot and the wall seems to cause the state estimates to diverge from each other. The three screenshots of RViz are shown below.

Figure 1 The initial state

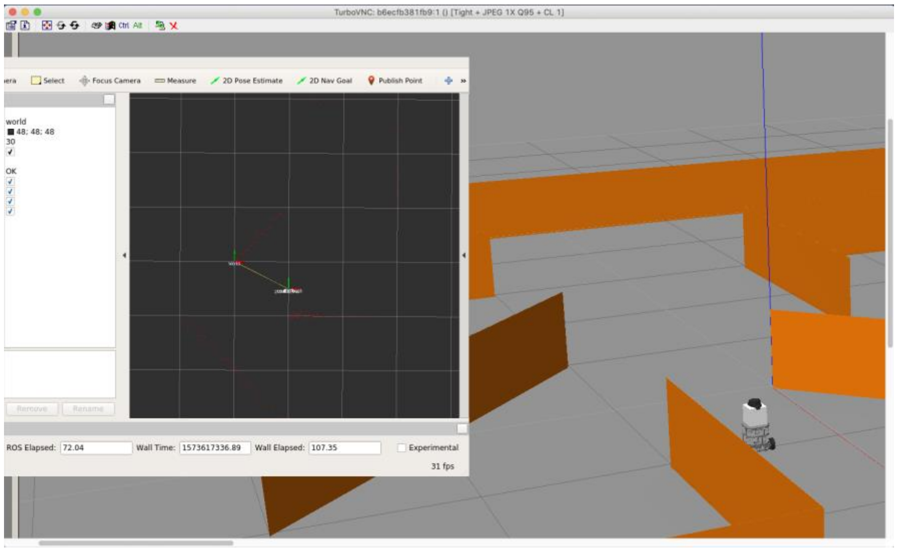


Figure 2 Far from the initial state

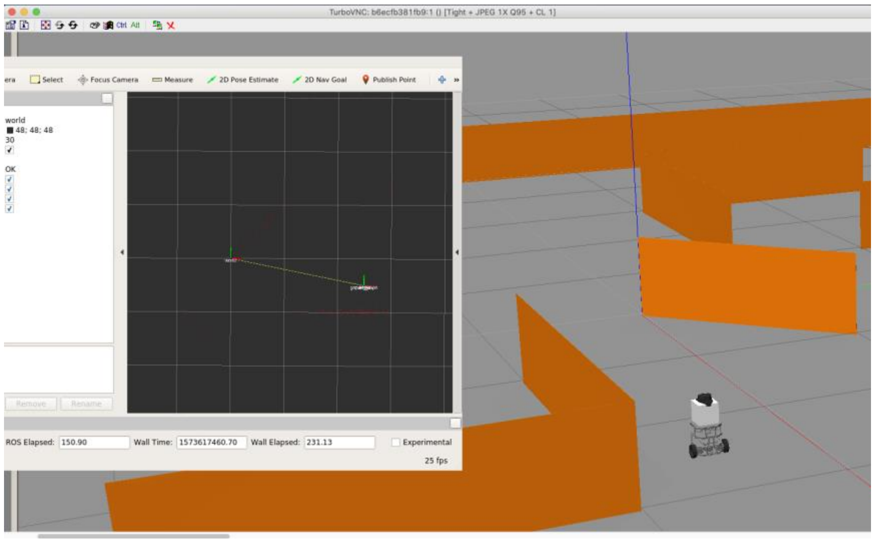
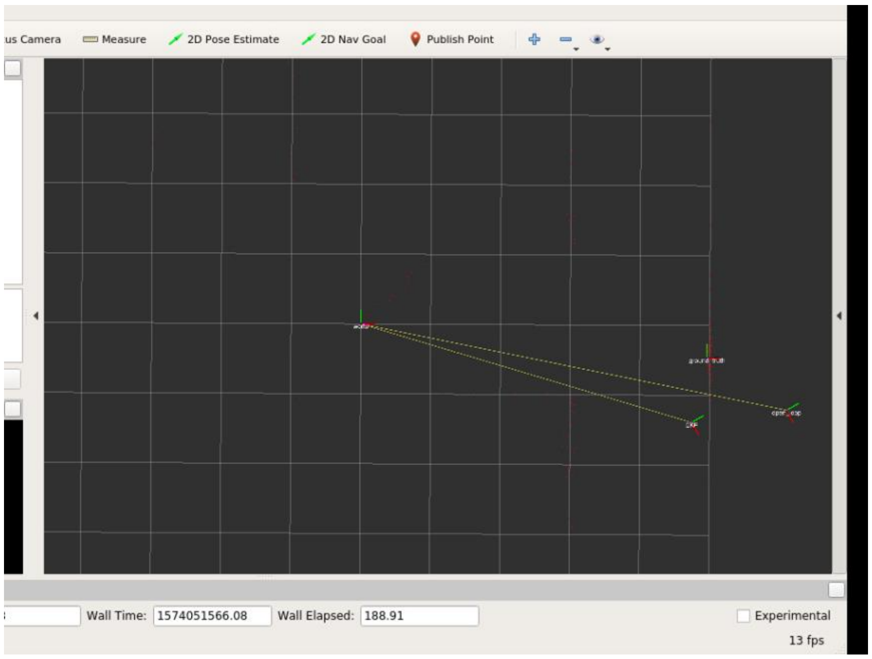


Figure 3 The state estimates start to diverge



Wall Time: 1574051566.08 Wall Elapsed: 188.91

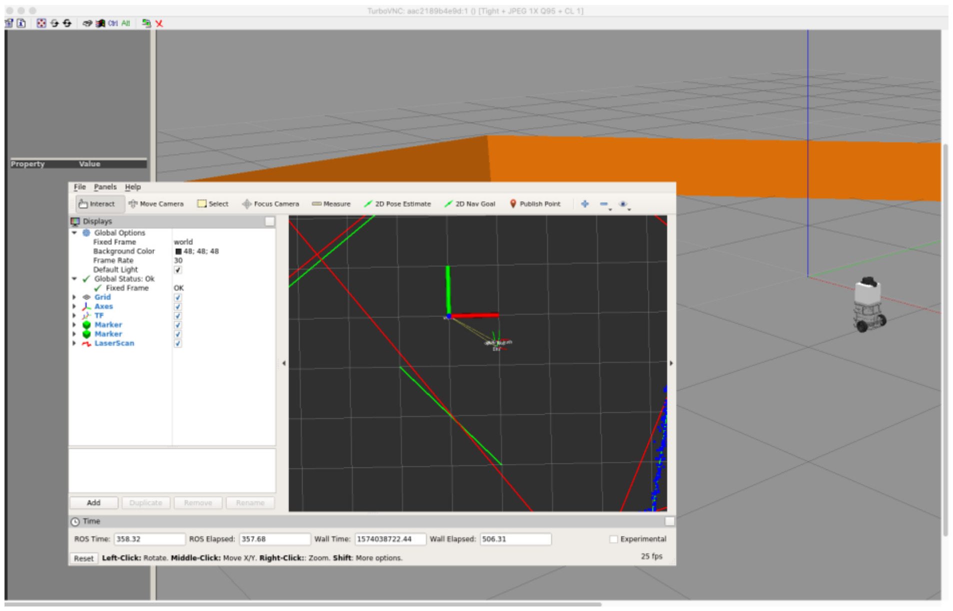
13 fps

Problem 2: EKF Localization

iii)

By driving the TurtleBot around, the map estimates changes at first. The EKF estimate will converge to the true estimate over time but the open loop estimate will not. It seems that when robot hits the wall, the EKF and ground truth estimates starts to diverge. And when the robot runs normally around the arena, the EKF estimate will gradually converge to the true estimate.

Figure 4 Initial state

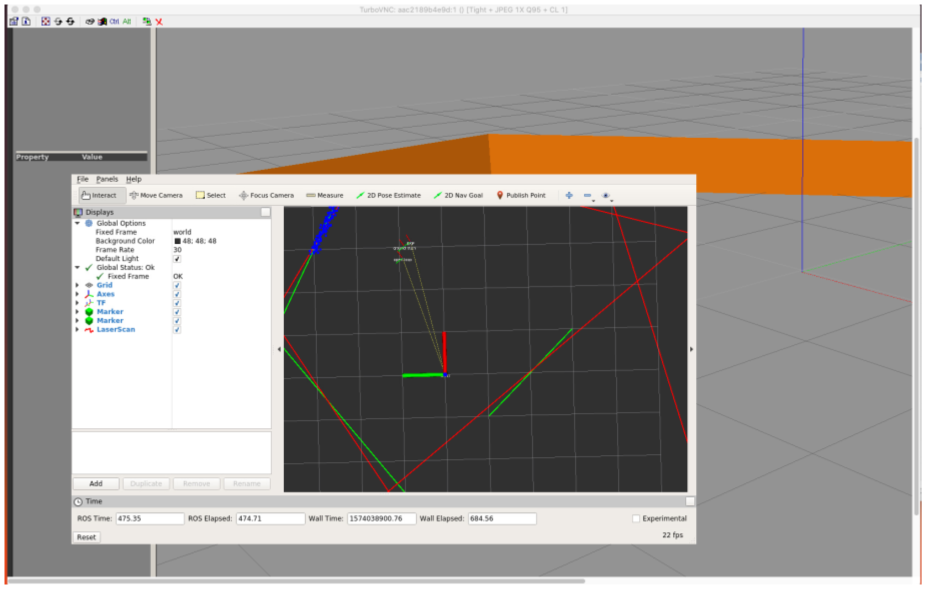


Global Options	
Fixed Frame	world
Background Color	■ 48; 48; 48
Frame Rate	30
Default Light	✓
Global Status: Ok	
Fixed Frame	OK
Grid	✓
Axes	✓
TF	✓
Marker	✓
Marker	✓
LaserScan	✓

ROS Time: 358.32 ROS Elapsed: 357.68 Wall Time: 1574038722.44 Wall Elapsed: 506.31 Experimental

Left-Click: Rotate. **Middle-Click:** Move X/Y. **Right-Click:**: Zoom. **Shift**: More options. 25 fps

Figure 5 Away from the initial state

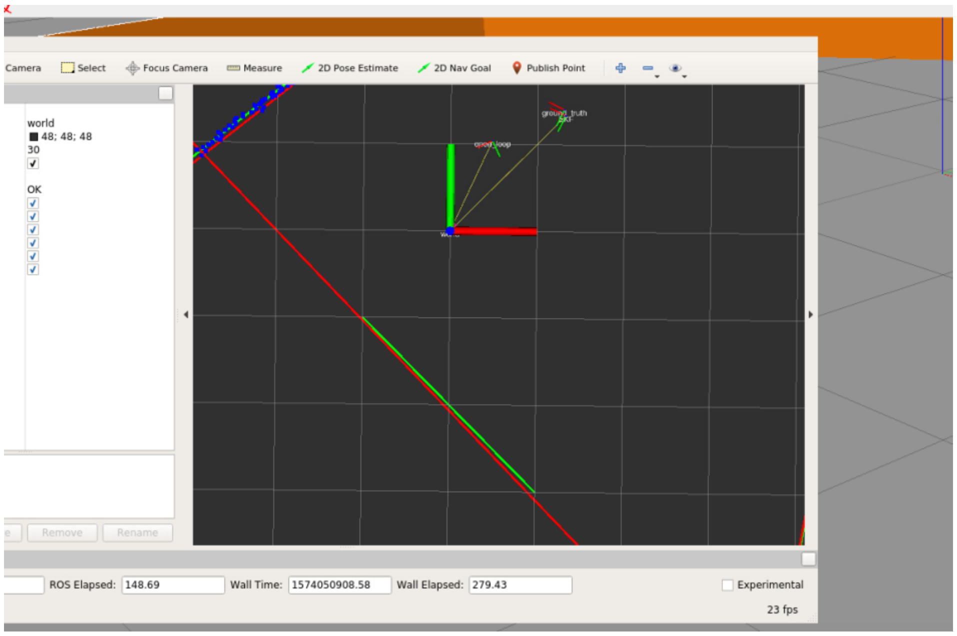


Global Options	
Fixed Frame	world
Background Color	■ 48; 48; 48
Frame Rate	30
Default Light	✓
Global Status: Ok	
Fixed Frame	OK
Grid	✓
Axes	✓
TF	✓
Marker	✓
Marker	✓
LaserScan	✓

ROS Time: 475.35 ROS Elapsed: 474.71 Wall Time: 1574038900.76 Wall Elapsed: 684.56 Experimental

22 fps

Figure 6 The map estimates have converged



ROS Elapsed: 148.69 Wall Time: 1574050908.58 Wall Elapsed: 279.43

Extra Problem: Monte Carlo Localization

iv)

By driving around the TurtleBot, the open loop and MCL state estimates converge to the true estimates at first. When the robot hits the wall, both the MCL and open loop state estimates will diverge from the true estimate. But the MCL estimate will self-correct the error after a certain time interval and converge to true estimate.

When the number of particles increase, the time it takes to for MCL estimate to self-correct will decrease. More number of particles will improve the performance of the self-correctness of MCL estimate.

Figure 7 Initial state:

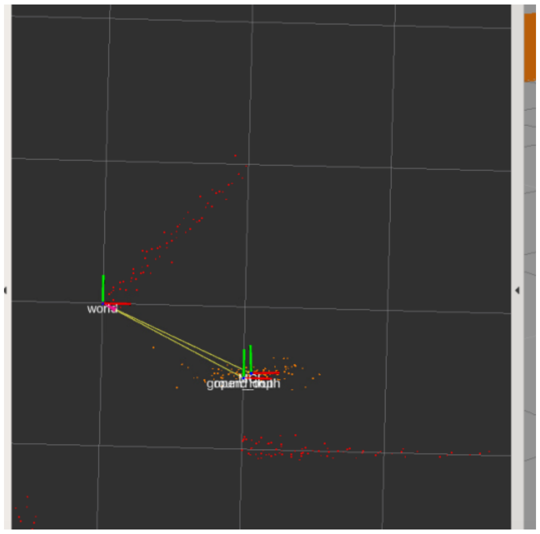


Figure 8 Away from initial state.

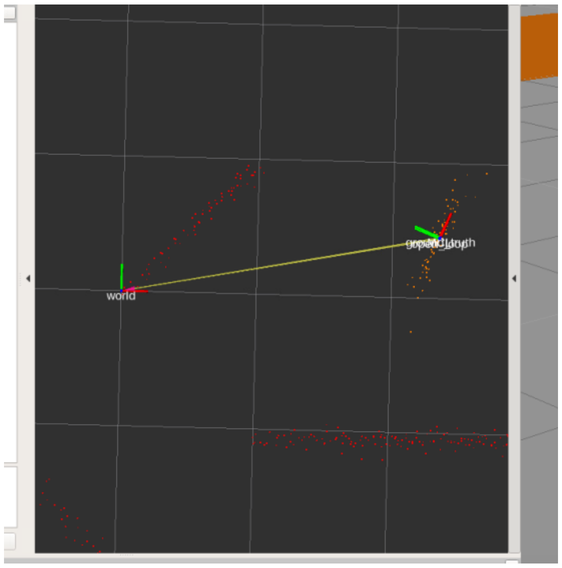
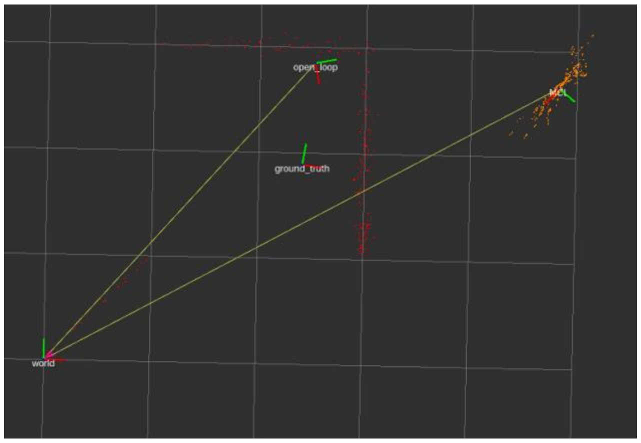


Figure 9 the MCL and ground truth estimates diverge



v)

- resample()

```
m = np.linspace(0, self.M - 1, num=self.M)
u = np.sum(ws) * (r + m / self.M)

ws_sum = np.cumsum(ws)
idx = np.searchsorted(ws_sum, u)
self.xs = xs[idx]
self.ws = ws[idx]
```

- transition_model()

```
x = self.xs[:, 0]  # [M, 1]
y = self.xs[:, 1]  # [M, 1]
th = self.xs[:, 2]  # [M, 1]
V = us[:, 0]  # [M, 1]
om = us[:, 1]  # [M, 1]
th_t = th + om * dt
# for om < Epsilon
x_t1 = x + V * dt * np.cos(th_t)
y_t1 = y + V * dt * np.sin(th_t)

x_t1 = np.where(abs(om) >= EPSILON_OMEGA, 0, x_t1)
y_t1 = np.where(abs(om) >= EPSILON_OMEGA, 0, y_t1)
```

```
# for om > Epsilon
x_t2 = (V / om) * (np.sin(th_t) - np.sin(th)) + x
y_t2 = (-V / om) * (np.cos(th_t) - np.cos(th)) + y

x_t2 = np.where(abs(om) < EPSILON_OMEGA, 0, x_t2)
y_t2 = np.where(abs(om) < EPSILON_OMEGA, 0, y_t2)

x_t = x_t1 + x_t2
y_t = y_t1 + y_t2

g = np.c_[x_t, y_t, th_t]
```

- compute_innovations()

```
alpha = self.map_lines[0, :]  # [1, J]
r = self.map_lines[1, :]  # [1, J]

x_cam, y_cam, th_cam = self.tf_base_to_camera  # float
# compute pose of camera in world frame

hs = np.zeros((self.M, 2, self.map_lines.shape[1]))
for i in range(self.M):
    x = self.xs[i, 0]  # [M, 1]
    y = self.xs[i, 1]  # [M, 1]
    th = self.xs[i, 2]  # [M, 1]
    x_cam_w = x_cam * np.cos(th) - y_cam * np.sin(th) + x  # [1, 1]
    y_cam_w = x_cam * np.sin(th) + y_cam * np.cos(th) + y

    alpha_pre = alpha - th - th_cam
    r_pre = r - x_cam_w * np.cos(alpha) - y_cam_w * np.sin(alpha)

    # normalize
    idx = r_pre < 0
    r_pre[idx] *= -1
    alpha_pre[idx] += np.pi
```

```
alpha_pre = (alpha_pre + np.pi) % (2 * np.pi) - np.pi

hs[i] = np.array([alpha_pre, r_pre])
```