AA 274A: Principles of Robot Autonomy I
Problem Set 4

Problem 1

(viii)

The EKF and the open loop estimates are driven apart by driving the robot especially close to the walls. By driving the robot into a wall without sending the stop command, the estimates grew further and further apart. Additionally, the difference in open loop and EKF estimates grew as the robot moved further from its initial position. We perturbed the robot's initial state estimate with a small angle offset and position offset. While the open loop and EFK estimates diverged more as the robot got further from the initial point, they got closer as we made the robot return.

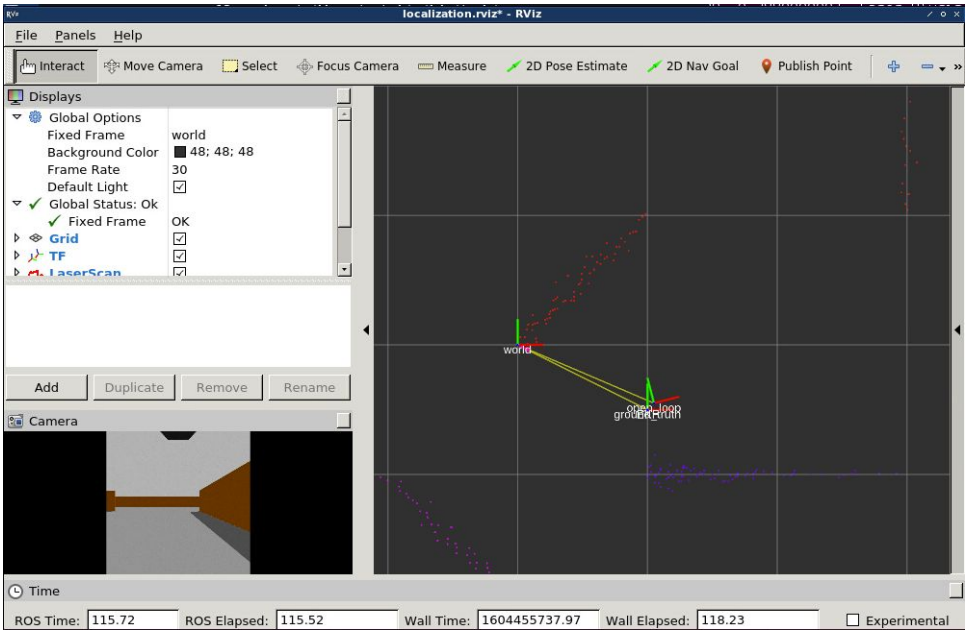

Figure 1: Initial Robot State and Estimated State

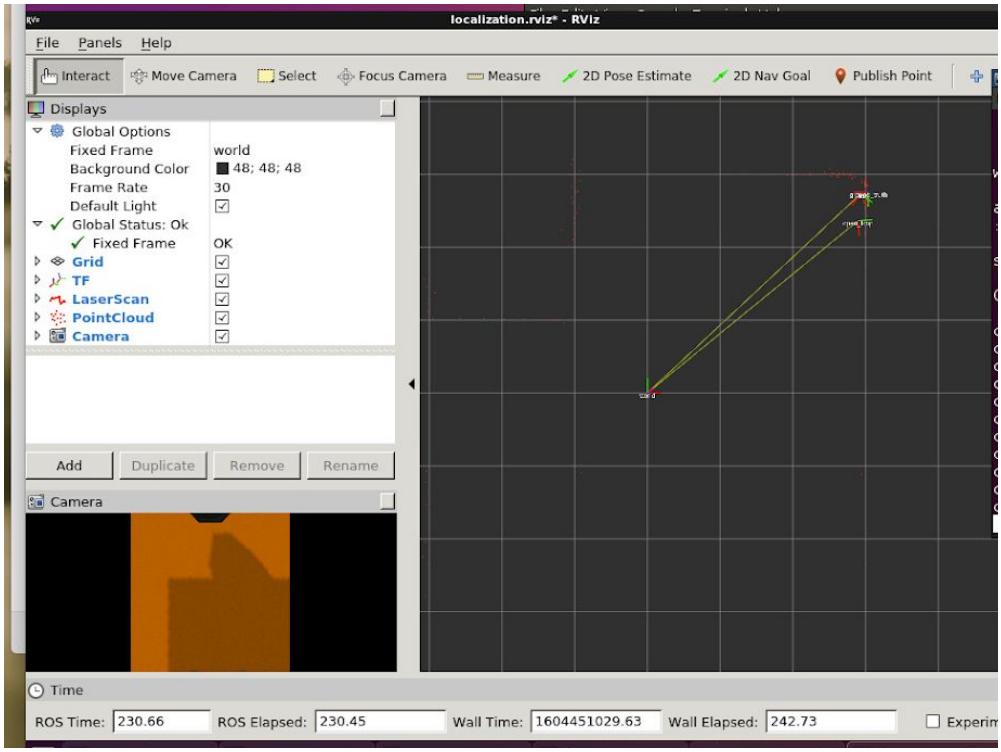

Figure 2: Far Away from Initial State

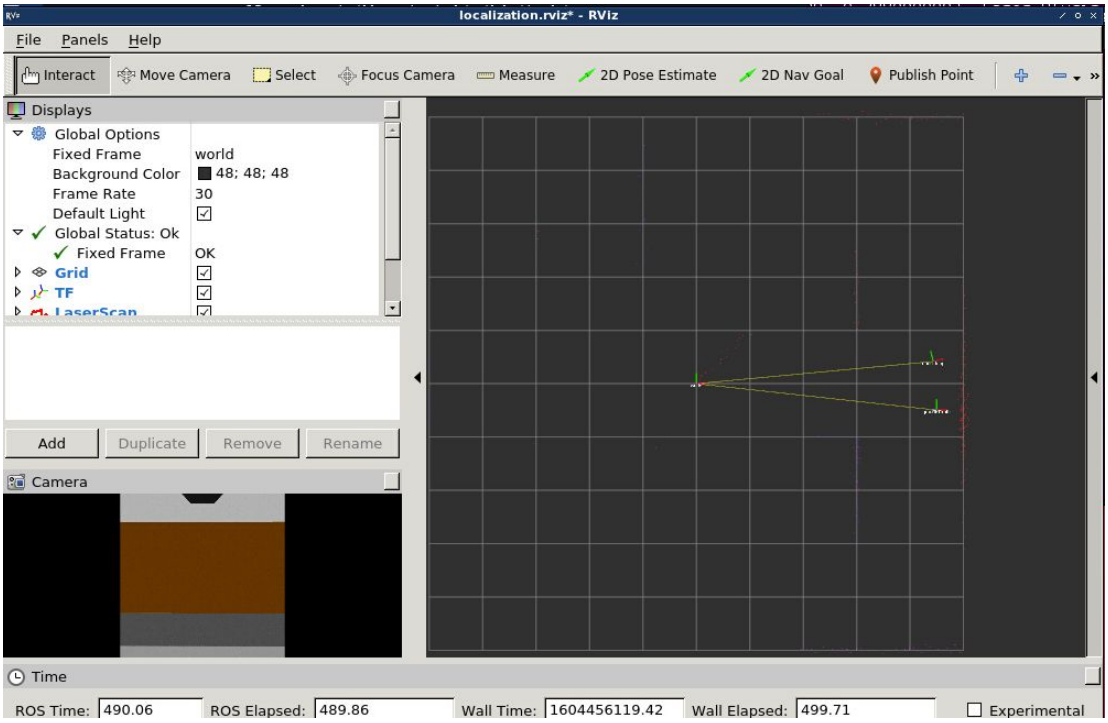

Figure 3: Drifting of the Open Loop from the EKF Estimate

Problem 2

(iii)

To make the map estimate converge to the right one, the robot must move around and explore the space. Figure 4 shows the initial state and how the estimate (red lines) differ from the actual (green lines). As the robot moves, more information is collected and the map becomes more accurate. Figures 5 shows the result after the robot can see another wall. At this point, the estimate is very accurate for two walls. Figure 6 shows the result after the robot has moved around the entire space. The map estimate almost perfectly matches the actual map.

The EKF and ground truth estimates diverge when there is a lot of motion without new information of the map collected. Figure 7 shows a large divergence when the robot is initially driven straight into the first wall.

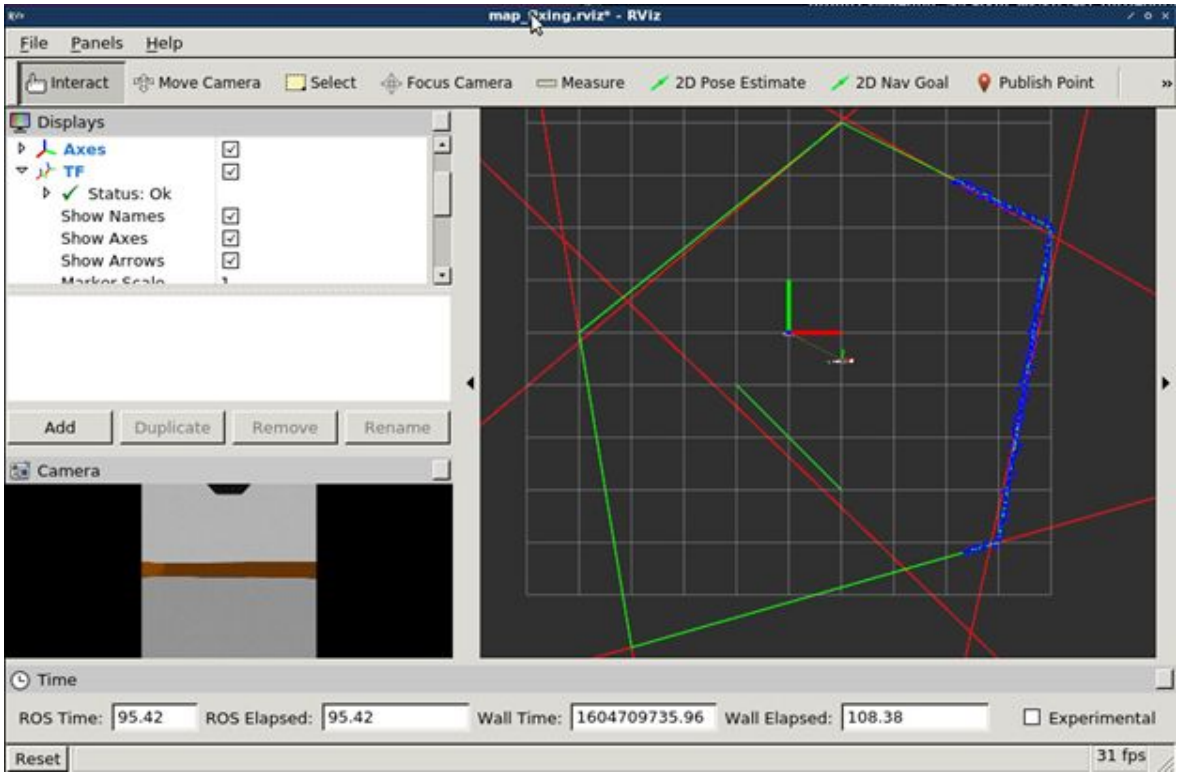

Figure 4: Initial Robot State

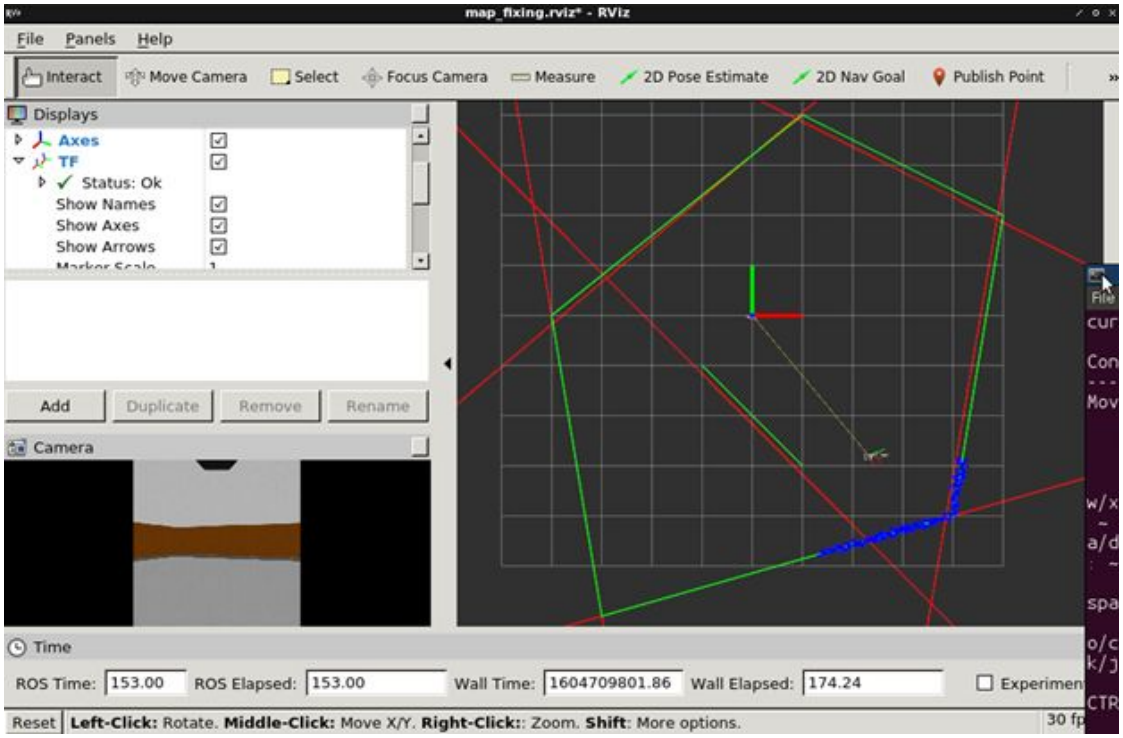

Figure 5: Line Estimates Change as the Robot moves far from Initial State

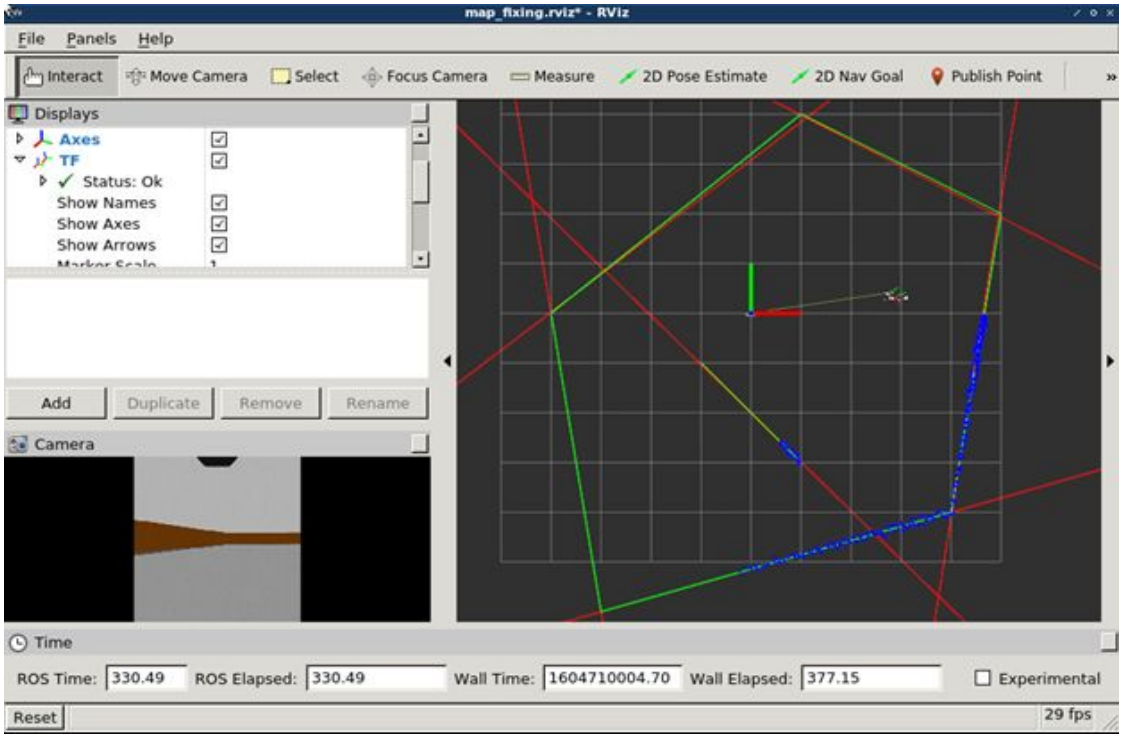

Figure 6: Map Estimates have Converged

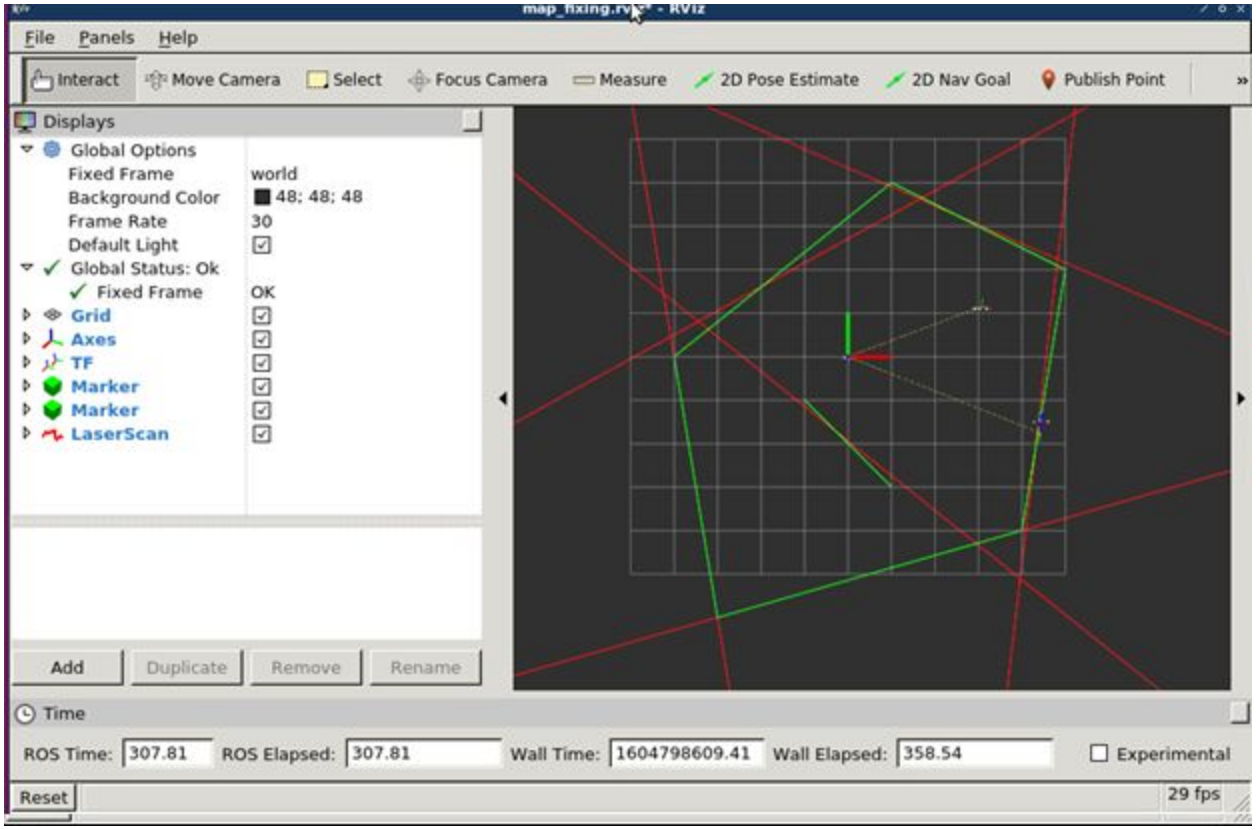

Figure 7: Robot State Estimate is Diverged from Truth

Extra Credit: Monte Carlo Localization

(iv) The MCL estimate diverges from the true robot state particularly when the robot drives straight for a long time without being close to a wall. This caused the estimate to move wildly off the truth value, as seen in Figure 10, which was taken when using 100 points for the MCL. Using 1000 points was prohibitively slow. 500 also proved to be impossible to run as the estimates were generated very infrequently. Running the estimation with only 50 points did not change the behaviour; the estimate rapidly diverged when driving straight and far from a wall. When the robot parked close to a wall, the MCL estimate drew back closer to the truth value.

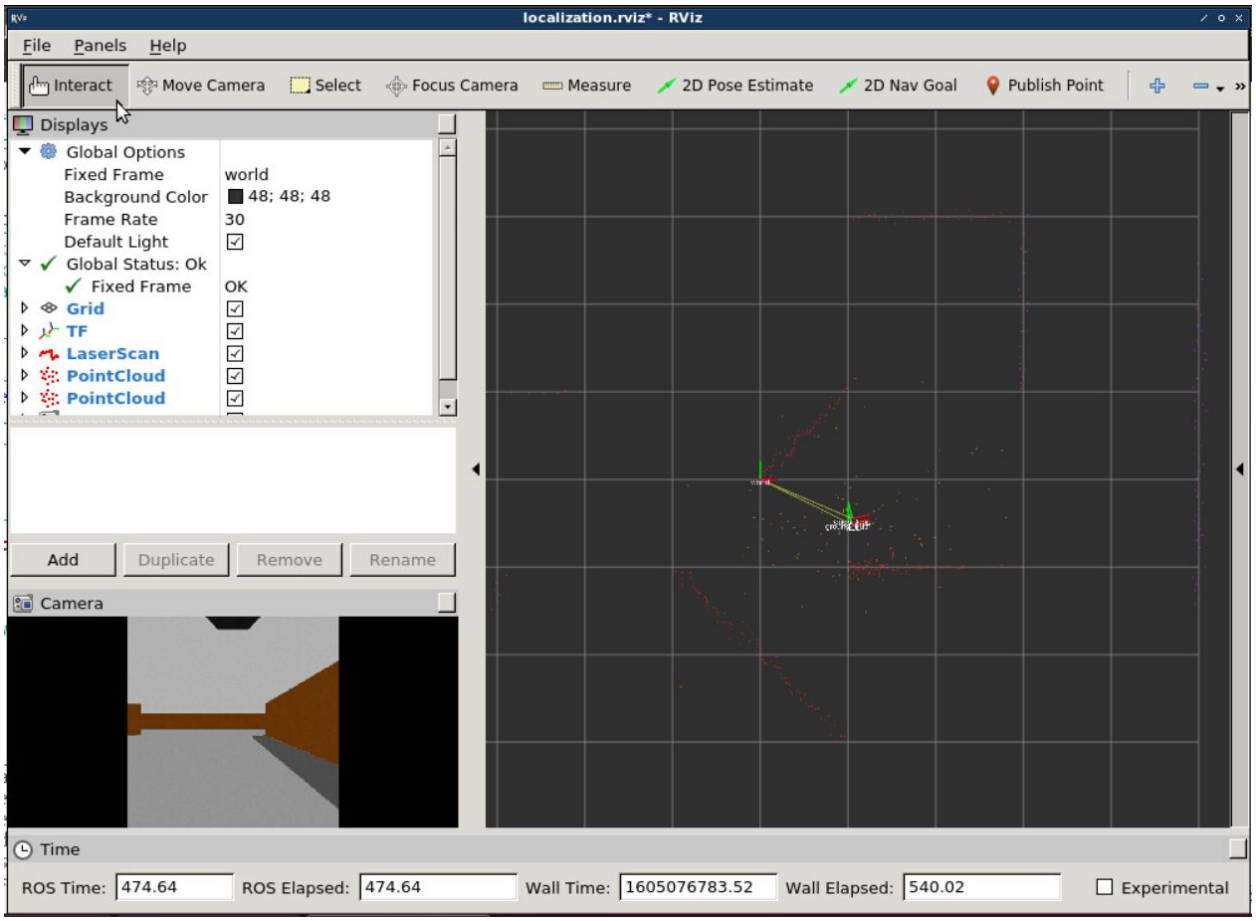

Figure 8: Initial Robot State

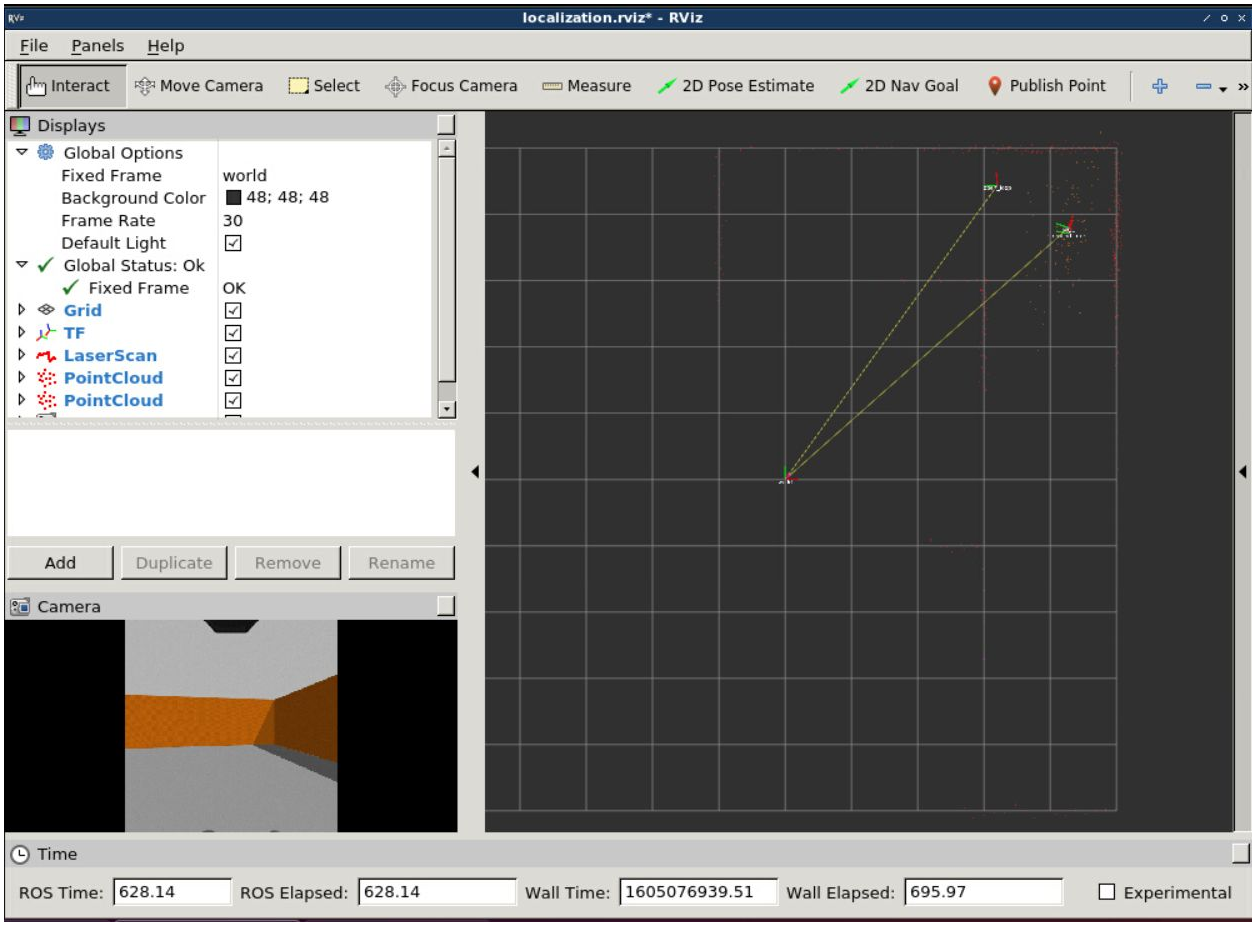

Figure 9: Far Away from Initial

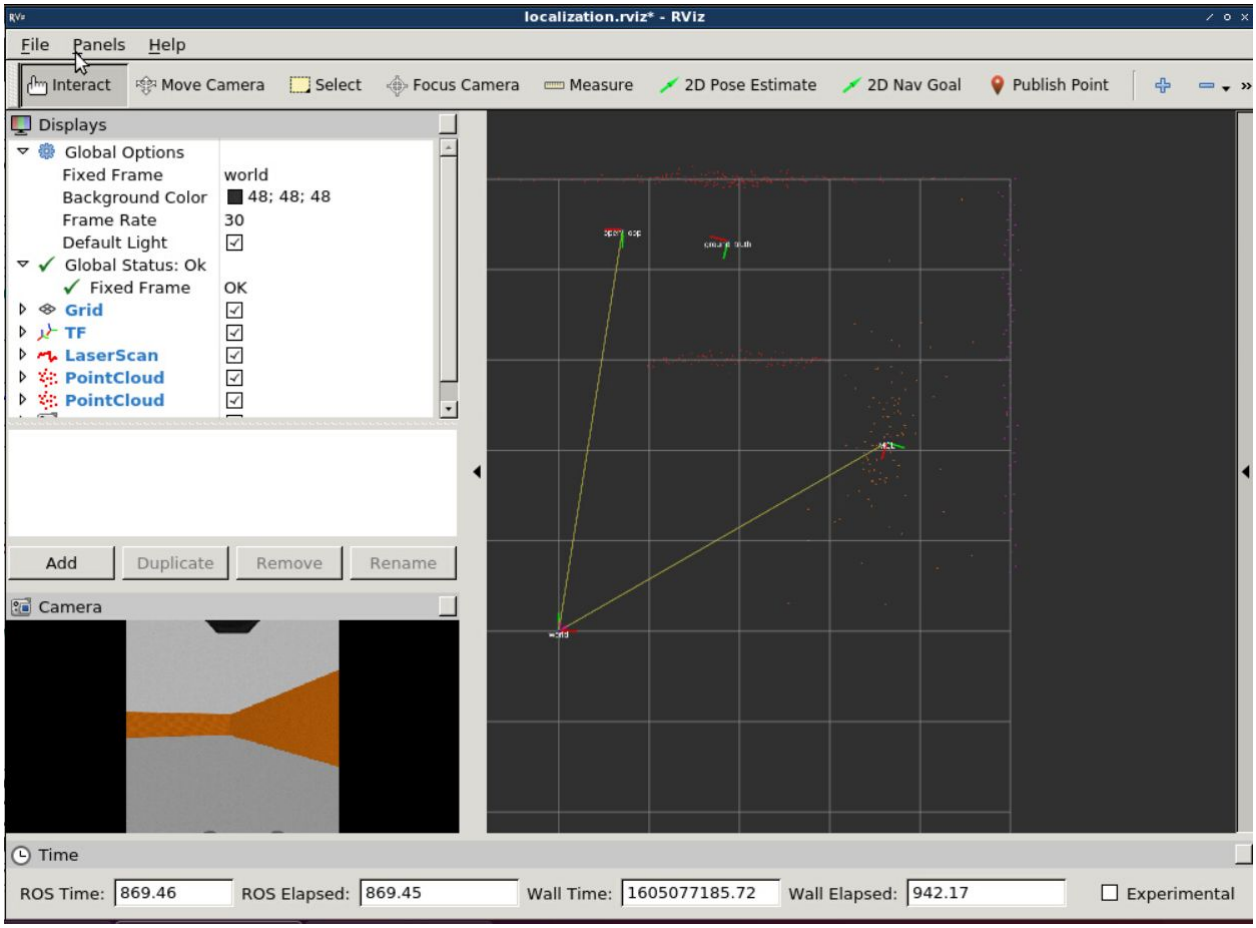

Figure 10: MCL Estimate Diverged from Truth

(v)

compute_predicted_measurements():

```
def compute_predicted_measurements(self):

    """
    Given a single map line in the world frame, outputs the line parameters
    in the scanner frame so it can be associated with the lines extracted
    from the scanner measurements.

    Input:
      None
    Output:
      hs: np.array[M,2,J] - J line parameters in the scanner (camera) frame for M particles.
    """
    ########## Code starts here ##########
    # TODO: Compute hs.
    # Hint: We don't need Jacobians for particle filtering.
    # Hint: Simple solutions: Using for loop, for each particle, for each
    #       map line, transform to scanner frame using tb.transform_line_to_scanner_frame()
    #       and tb.normalize_line_parameters()
    # Hint: To maximize speed, try to compute the predicted measurements
    #       without looping over the map lines. You can implement vectorized
    #       versions of turtlebot_model functions directly here. This
    #       results in a ~10x speedup.
    # Hint: For the faster solution, it does not call tb.transform_line_to_scanner_frame()
    #       or tb.normalize_line_parameters(), but reimplement these steps vectorized.

    J = self.map_lines.shape[1]
    hs = np.zeros((self.M,2,J))

    def vectorized_transform_line_to_scanner_frame():

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

      for ii, x in enumerate(self.xs):

        rot_b_to_w = np.array([[np.cos(x[2]), -np.sin(x[2]), x[0]],
                               [np.sin(x[2]),  np.cos(x[2]), x[1]],
                               [0, 0, 1]])

        x_cam_b = self.tf_base_to_camera[0]
        y_cam_b = self.tf_base_to_camera[1]
        th_cam_b = self.tf_base_to_camera[2]

        cam_b = np.array([x_cam_b, y_cam_b, 1])
        cam_w = np.matmul(rot_b_to_w, cam_b)
        cam_w[2] = th_cam_b + x[2]

        alpha_cam = alpha - cam_w[2]

        alpha_l = alpha - np.arctan2(cam_w[1], cam_w[0])
        r_cam = r - np.sqrt(cam_w[0]**2 + cam_w[1]**2) * np.cos(alpha_l)

        h = vectorized_normalize_line_parameters(np.array([alpha_cam, r_cam]))

        hs[ii, :,0:J] = h
```

```
    return hs

def vectorized_normalize_line_parameters(h):
    idx = np.argwhere(h[1,:] < 0)[:,0]
    if len(idx) > 0:
        h[0,idx] += np.pi
        h[1,idx] *= -1
    h[0,:] = (h[0,:] + np.pi) % (2*np.pi) - np.pi

    return h

hs = vectorized_transform_line_to_scanner_frame()


########## Code ends here ##########

return hs
```

transition_model():

```
def transition_model(self, us, dt):
    """
    Unicycle model dynamics.

    Inputs:
      us: np.array[M,2] - zero-order hold control input for each particle.
      dt: float         - duration of discrete time step.
    Output:
      g: np.array[M,3] - result of belief mean for each particle
                         propagated according to the system dynamics with
                         control u for dt seconds.
    """

    ########## Code starts here ##########
    # TODO: Compute g.
    # Hint: We don't need Jacobians for particle filtering.
    # Hint: A simple solution can be using a for loop for each particle
    #       and a call to tb.compute_dynamics
    # Hint: To maximize speed, try to compute the dynamics without looping
    #       over the particles. If you do this, you should implement
    #       vectorized versions of the dynamics computations directly here
    #       (instead of modifying turtlebot_model). This results in a
    #       ~10x speedup.
    # Hint: This faster/better solution does not use loop and does
    #       not call tb.compute_dynamics. You need to compute the idxs
    #       where abs(om) > EPSILON_OMEGA and the other idxs, then do separate
    #       updates for them

#       g = np.zeros((us.shape[0], 3))
    g = np.zeros((self.M, 3))

    idx_small_omg = np.argwhere(us[:,1] < EPSILON_OMEGA)[:,0]
    idx_omg = np.argwhere(us[:,1] >= EPSILON_OMEGA)[:,0]

    if len(idx_small_omg) > 0:
      theta_t = self.xs[idx_small_omg, 2] + us[idx_small_omg, 1] * dt

                        g[idx_small_omg, :] = (np.array([self.xs[idx_small_omg, 0] + us[idx_small_omg, 0] * (np.cos(theta_t) +
np.cos(self.xs[idx_small_omg, 2]))/2. * dt,
                                self.xs[idx_small_omg, 1] + us[idx_small_omg, 0] * (np.sin(theta_t) + np.sin(self.xs[idx_small_omg, 2]))/2. *
dt,
                               theta_t])).T

    if len(idx_omg) > 0:
      theta_t = self.xs[idx_omg, 2] + us[idx_omg, 1] * dt

      g[idx_omg, :] = (np.array([self.xs[idx_omg, 0] + (us[idx_omg, 0]/us[idx_omg, 1]) * (np.sin(theta_t) - np.sin(self.xs[idx_omg, 2])),
                        self.xs[idx_omg, 1] - (us[idx_omg, 0]/us[idx_omg, 1]) * (np.cos(theta_t) - np.cos(self.xs[idx_omg, 2])),
                        theta_t])).T


    ########## Code ends here ##########

    return g
```

resample():

```
def resample(self, xs, ws):
        """
        Resamples the particles according to the updated particle weights.

        Inputs:
            xs: np.array[M,3] - matrix of particle states.
            ws: np.array[M,]  - particle weights.

        Output:
            None - internal belief state (self.xs, self.ws) should be updated.
        """
        r = np.random.rand() / self.M

        ########## Code starts here ##########
        # TODO: Update self.xs, self.ws.
        # Note: Assign the weights in self.ws to the corresponding weights in ws
        #       when resampling xs instead of resetting them to a uniform
        #       distribution. This allows us to keep track of the most likely
        #       particle and use it to visualize the robot's pose with self.x.
        # Hint: To maximize speed, try to implement the resampling algorithm
        #       without for loops. You may find np.linspace(), np.cumsum(), and
        #       np.searchsorted() useful. This results in a ~10x speedup.

        s = np.sum(ws)
        m = np.arange(self.M)
        u = s * (r + (m/float(self.M)))

        c = np.cumsum(ws)

        inds = np.searchsorted(c,u)

        self.xs = xs[inds]
        self.ws = ws[inds]

        ########## Code ends here ##########
```