Grid Localization Using Bayes Filter

Akshati Vaishnav

Grid Localization

The robot state is 3 dimensional and is given by (x, y, θ). The robot's world spans from $[-5.5, 6.5)$ ft in the x-direction, $[-4.5, 4.5)$ ft in the y-direction, and $[-180, 180)$ degrees in the θ-axis.
Since there are infinitely many locations at which the robot can be in this space, we decrease the number of states by discretizing the locations into a 3D grid space. The size of x and y grids is 0.3048m, and the size of the θ axis is 20 degrees. Therefore, the total number of cells are 12 in the x-axis, 8 in the y-axis, and 18 in the θ-axis.
Each grid cell stores the probability of the robot's presence at that cell.

Belief

The belief of the robot is represented by the set of probabilities of each grid cell. The sum of all of these probabilities is 1.
The grid cell with the highest probability after every Bayes filter iteration represents the most probable pose of the robot.

Sensor Model

The measurement noise can be represented using a Gaussian distribution.

Motion Model

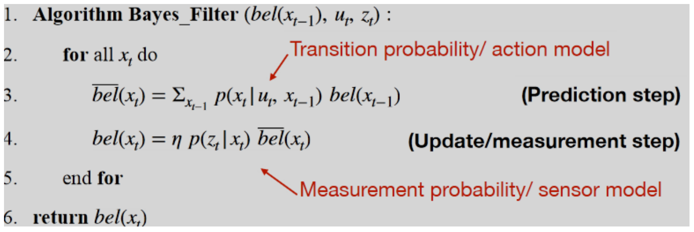
The odometry motion model is used in this lab, in which odometry data is recorded before and after the movement. Again, a Gaussian distribution is used to model noisy control data in the motion model.

Bayes Filter Algorithm

Notation

- u: control input that is fed to the robot and allows it to move from its previous position to its current position
- z: real-time sensor measurement
- x_t: iteration

Algorithm



```
1. Algorithm Bayes_Filter ($bel(x_{t-1}), u_t, z_t$) :
2.     for all $x_t$ do
3.         $\overline{bel}(x_t) = \Sigma_{x_{t-1}} p(x_t | u_t, x_{t-1})\ bel(x_{t-1})$        (Prediction step)
4.         $bel(x_t) = \eta\ p(z_t | x_t)\ \overline{bel}(x_t)$        (Update/measurement step)
5.     end for
6. return $bel(x_t)$
```

Transition probability/ action model
Measurement probability/ sensor model

Implementation

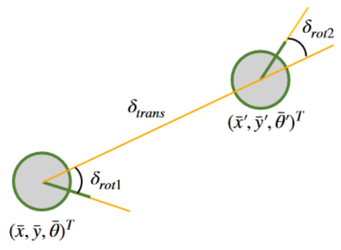
Compute Control

- Generates a control input given a current and previous state
- Returns three values: initial rotation (δ_{rot1}) , translation (δ_{trans}), final rotation (δ_{rot2})

$$\delta_{rot1} = \operatorname{atan2}(\bar{y}' - \bar{y}, \bar{x}' - \bar{x}) - \bar{\theta}$$

$$\delta_{trans} = \sqrt{(\bar{y}' - \bar{y})^2 + (\bar{x}' - \bar{x})^2}$$

$$\delta_{rot2} = \bar{\theta}' - \bar{\theta} - \delta_{rot1}$$


δ_{rot2}
δ_{trans}
$(\bar{x}', \bar{y}', \bar{\theta}')^T$
δ_{rot1}
$(\bar{x}, \bar{y}, \bar{\theta})^T$

```python
def compute_control(cur_pose, prev_pose):

    """ Given the current and previous odometry poses, this function extracts the control information based on the 
odometry motion model.
    Args:
        cur_pose  ([Pose]): Current Pose
        prev_pose ([Pose]): Previous Pose 
    Returns:
        [delta_rot_1]: Rotation 1  (degrees)
        [delta_trans]: Translation (meters)
        [delta_rot_2]: Rotation 2  (degrees)
    """

    del_theta = np.degrees(np.arctan2((cur_pose[1]-prev_pose[1]),(cur_pose[0]-prev_pose[0])))
    delta_rot_1 = mapper.normalize_angle(del_theta - prev_pose[2])
    delta_rot_2 = mapper.normalize_angle(cur_pose[2] - prev_pose[2] - delta_rot_1)
    delta_trans = np.sqrt((cur_pose[1]-prev_pose[1])**2+(cur_pose[0]-prev_pose[0])**2)

    return delta_rot_1, delta_trans, delta_rot_2
```

Odometer Motion Model

- Computes the probability that a robot ends up at a current state given a previous state and a control input u.
- We compute a temporary ctrl_u, which is the actual motion needed to get from prev_pose to cur_pose, while u is the control input provided.

```python
def odom_motion_model(cur_pose, prev_pose, u):

    """ Odometry Motion Model
    Args:
        cur_pose  ([Pose]): Current Pose
        prev_pose ([Pose]): Previous Pose
        (rot1, trans, rot2) (float, float, float): A tuple with control data in the format 
                                format (rot1, trans, rot2) with units (degrees, meters, degrees)
    Returns:
        prob [float]: Probability p(x'|x, u)
    """

    ctrl_u = compute_control(cur_pose, prev_pose)
    prob_rot_1 = loc.gaussian(ctrl_u[0], u[0], loc.odom_rot_sigma)
    prob_trans = loc.gaussian(ctrl_u[1], u[1], loc.odom_trans_sigma)
    prob_rot_2 = loc.gaussian(ctrl_u[2], u[2], loc.odom_rot_sigma)
    prob = prob_rot_1*prob_trans*prob_rot_2

    return prob
```

Bayes Prediction Step

- Incorporates the control input (movement) data
- For all grid cells, determine the belief and odometry model parameters to update $\overline{bel}$, which is our prediction
- Normalize the $\overline{bel}$ value at the end to ensure a total probability of 1
- To improve the speed of the prediction step, only probabilities greater than or equal to 0.0001 are contributed to the belief

```python
def prediction_step(cur_odom, prev_odom):

    """ Prediction step of the Bayes Filter.
    Update the probabilities in loc.bel_bar based on loc.bel from the previous time step and the odometry motion model.
    Args:
        cur_odom  ([Pose]): Current Pose
        prev_odom ([Pose]): Previous Pose
    """
    u = compute_control(cur_odom, prev_odom)
    loc.bel_bar = np.zeros((mapper.MAX_CELLS_X, mapper.MAX_CELLS_Y, mapper.MAX_CELLS_A))
    for x1 in range(mapper.MAX_CELLS_X):
        for y1 in range(mapper.MAX_CELLS_Y):
            for theta1 in range(mapper.MAX_CELLS_A):
                bel = loc.bel[x1,y1,theta1]
                #ignore impossible grids
                if(bel >= 0.0001):
                    for x2 in range(mapper.MAX_CELLS_X):
                        for y2 in range(mapper.MAX_CELLS_Y):
                            for theta2 in range(mapper.MAX_CELLS_A):
                                prob = odom_motion_model(mapper.from_map(x2, y2, theta2), mapper.from_map(x1, y1,
theta1), u)
                                loc.bel_bar[x2,y2,theta2] += (prob*bel)
    loc.bel_bar /= np.sum(loc.bel_bar)
```

Sensor Model

- Calculates the probability of every sensor observation based on a Gaussian distribution

```
def sensor_model(obs):

    """ This is the equivalent of p(z|x).
    Args:
        obs ([ndarray]): A 1D array consisting of the true observations for a specific robot pose in the map
    Returns:
        [ndarray]: Returns a 1D array of size 18 (=loc.OBS_PER_CELL) with the likelihoods of each individual sensor
measurement
    """

    sens_len = mapper.OBS_PER_CELL
    prob_array = [loc.gaussian(obs[i], loc.obs_range_data[i], loc.sensor_sigma) for i in range(sens_len)]

        return prob_array
```

Bayes Update Step

- For all grid cells, use the sensor model and the predicted belief to update the actual belief
- Normalize the beliefs to ensure that the total probability sums to 1.

```
def update_step():
    """ Update step of the Bayes Filter.
    Update the probabilities in loc.bel based on loc.bel_bar and the sensor model.
    """
    for x in range(mapper.MAX_CELLS_X):
        for y in range(mapper.MAX_CELLS_Y):
            for theta in range(mapper.MAX_CELLS_A):
                p = np.prod(sensor_model(mapper.get_views(x,y,theta)))
                loc.bel[x,y,theta] = (p*loc.bel_bar[x,y,theta])

    loc.bel /= np.sum(loc.bel)
```