Exercise 10 solutions

July 14, 2023

Exercise: FastSLAM Implementation

FastSLAM is a Rao-Blackwellized particle filter for simultaneous localization and mapping. The pose of the robot in the environment is represented by a particle filter. Furthermore, each particle carries a map of the environment, which it uses for localization. In the case of landmark-based FastSLAM, the map is represented by a Kalman Filter, estimating the mean position and covariance of landmarks.

Implement the landmark-based FastSLAM algorithm as presented in the lecture. Assume known feature correspondences.

To support this task, we provide a detailed listing of the algorithm as a PDF file and a small Python framework on the course website. The framework contains the following folders:

data *contains the world definition and sensor readings used by the filter.*

code *contains the FastSLAM framework with stubs for you to complete.*

You can run the fastSLAM framework in the terminal: `python fastslam.py`*. It will only work properly once you filled the blanks in the code.*

(a) *Complete the code blank in the* `sample_motion_model` *function by implementing the odometry motion model and sampling from it. The function updates the poses of the particles based on the old poses, the odometry measurements δ_{rot1}, δ_{trans} and δ_{rot2} and the motion noise. The motion noise parameters are:*

$$[\alpha_1, \alpha_2, \alpha_3, \alpha_4] = [0.1, 0.1, 0.05, 0.05] \tag{1}$$

How is sampling from the motion model different from the standard particle filter for localization (Exercise sheet 7)?

```
def sample_motion_model(odometry, particles):
    # Samples new particle positions, based on old positions, the odometry
    # measurements and the motion noise
```

```
    delta_rot1 = odometry['r1']
    delta_trans = odometry['t']
    delta_rot2 = odometry['r2']

    # the motion noise parameters: [alpha1, alpha2, alpha3, alpha4]
    noise = [0.1, 0.1, 0.05, 0.05]

    # standard deviations of motion noise
    sigma_delta_rot1 = noise[0] * abs(delta_rot1) + noise[1] * delta_trans

    sigma_delta_trans = noise[2] * delta_trans + \
                        noise[3] * (abs(delta_rot1) + abs(delta_rot2))

    sigma_delta_rot2 = noise[0] * abs(delta_rot2) + noise[1] * delta_trans

    # "move" each particle according to the odometry measurements plus sampled noise
    for particle in particles:

        #sample noisy motions
        noisy_delta_rot1 = delta_rot1 + np.random.normal(0, sigma_delta_rot1)
        noisy_delta_trans = delta_trans + np.random.normal(0, sigma_delta_trans)
        noisy_delta_rot2 = delta_rot2 + np.random.normal(0, sigma_delta_rot2)

        #remember last position to draw path of particle
        particle['history'].append([particle['x'], particle['y']])

        # calculate new particle pose
        particle['x'] = particle['x'] + \
                    noisy_delta_trans * np.cos(particle['theta'] + noisy_delta_rot1)

        particle['y'] = particle['y'] + \
                    noisy_delta_trans * np.sin(particle['theta'] + noisy_delta_rot1)

        particle['theta'] = particle['theta'] + \
                    noisy_delta_rot1 + noisy_delta_rot2

    return
```

Sampling from the motion model is done exactly like in the standard particle filter for localization. Our implementation differs slightly from the standard particle filter. Instead of generating a new particle set, we simply change the poses of each particle to apply the motion model. The other particle properties, like the observed landmarks and its weight, stay unchanged.

(b) *Complete the code blanks in the* `eval_sensor_model` *function. The function implements the measurement update of the Rao-Blackwellized particle filter, using range and bearing measurements. It takes the particles and landmark observations and*

updates the map of each particle and calculates its weight w. The noise of the sensor readings is given by a diagonal matrix

$$Q_t = \begin{bmatrix} 1.0 & 0 \\ 0 & 0.1 \end{bmatrix} \tag{2}$$

How is the evaluation of the sensor model different from the standard particle filter for localization (Exercise sheet 7)?

```
def eval_sensor_model(sensor_data, particles):
    #Correct landmark poses with a measurement and
    #calculate particle weight

    #sensor noise
    Q_t = np.array([[1.0, 0],
                    [0, 0.1]])

    #measured landmark ids and ranges
    ids = sensor_data['id']
    ranges = sensor_data['range']
    bearings = sensor_data['bearing']

    #update landmarks and calculate weight for each particle
    for particle in particles:

        landmarks = particle['landmarks']
        particle['weight'] = 1.0

        px = particle['x']
        py = particle['y']
        ptheta = particle['theta']

        #loop over observed landmarks
        for i in range(len(ids)):

            #current landmark
            lm_id = ids[i]
            landmark = landmarks[lm_id]

            #measured range and bearing to current landmark
            meas_range = ranges[i]
            meas_bearing = bearings[i]

            if not landmark['observed']:
                #landmark is observed for the first time

                #initialize landmark position based on the measurement and particle pose
```

```
                lx = px + meas_range * np.cos(ptheta + meas_bearing)
                ly = py + meas_range * np.sin(ptheta + meas_bearing)
                landmark['mu'] = [lx, ly]

                #get expected measurement and Jacobian wrt. landmark position
                h, H = measurement_model(particle, landmark)

                #initialize covariance for this landmark
                H_inv = np.linalg.inv(H)
                landmark['sigma'] = H_inv.dot(Q_t).dot(H_inv.T)

                landmark['observed'] = True

            else:
                #landmark was observed before

                #get expected measurement and Jacobian wrt. landmark position
                h, H = measurement_model(particle, landmark)

                #Calculate measurement covariance and Kalman gain
                S = landmark['sigma']
                Q = H.dot(S).dot(H.T) + Q_t
                K = S.dot(H.T).dot(np.linalg.inv(Q))

                #Compute the difference between the observed and the expected measurement
                delta = np.array([meas_range - h[0], angle_diff(meas_bearing,h[1])])

                #update estimated landmark position and covariance
                landmark['mu'] = landmark['mu'] + K.dot(delta)
                landmark['sigma'] = (np.identity(2) - K.dot(H)).dot(S)

                # compute the likelihood of this observation
                fact = 1 / np.sqrt(math.pow(2*math.pi,2) * np.linalg.det(Q))
                expo = -0.5 * np.dot(delta.T, np.linalg.inv(Q)).dot(delta)
                weight = fact * np.exp(expo)

                # alternatively, evaluate normal density with scipy:
                # weight = scipy.stats.multivariate_normal.pdf(delta, \
                #          mean=np.array([0,0]), cov=Q)

                # calculate overall weight, account for observing
                # multiple landmarks at one time step
                particle['weight'] = particle['weight'] * weight

    #normalize weights
    normalizer = sum([p['weight'] for p in particles])

    for particle in particles:
```

```
        particle['weight'] = particle['weight'] / normalizer

    return
```

The evaluation of the sensor model is quite different from the standard particle filter for localization. Like in the standard particle filter, we calculate the weight for each particle, according to the likelihood of the observation. However, in addition we need to update the Kalman Filter for the landmark observations.

(c) *Complete the function* `resample_particles` *by implementing stochastic universal sampling. The function takes as an input a set of particles which carry their weights, and returns a sampled set of particles.*

How does the resampling procedure differ from resampling in the standard particle filter for localization (Exercise sheet 7)?

```
def resample_particles(particles):
    # Returns a new set of particles obtained by performing
    # stochastic universal sampling, according to the particle
    # weights.

    # distance between pointers
    step = 1.0/len(particles)

    # random start of first pointer
    u = np.random.uniform(0,step)

    # where we are along the weights
    c = particles[0]['weight']

    # index of weight container and corresponding particle
    i = 0

    new_particles = []

    #loop over all particle weights
    for particle in particles:

        #go through the weights until you find the particle
        #to which the pointer points
        while u > c:

            i = i + 1
            c = c + particles[i]['weight']

        #add that particle
        new_particle = copy.deepcopy(particles[i])
        new_particle['weight'] = 1.0/len(particles)
        new_particles.append(new_particle)
```

```
        #increase the threshold
        u = u + step

    return new_particles
```

The resampling procedure is again exactly like in the standard particle filter.