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

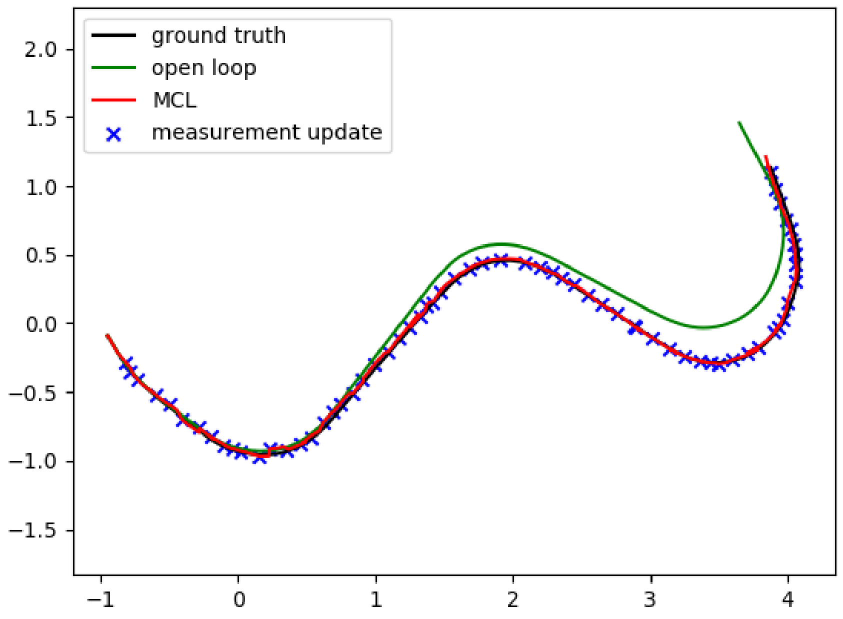
Name: Abhyudit Singh Manhas
SUID: 06645995

Extra Credit Problem: Monte Carlo Localization

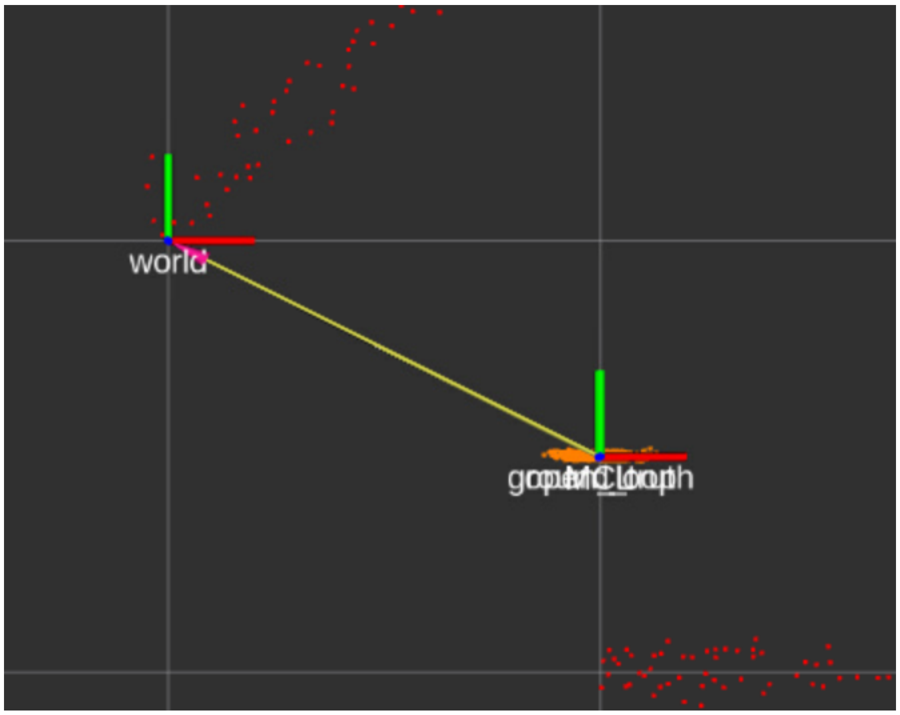
(i) Implemented the transition update step of MCL by completing the `transition_model()` method in the `MonteCarloLocalization` class and the `transition_update()` method in the `ParticleFilter` class in `particle_filter.py`.
Validated my work by running `validate_transition_model()` from `validate_particle_filter.py`.

(ii) Implemented the measurement update step by completing the `measurement_update()`, `measurement_model()`, `compute_innovations()`, and `compute_predicted_measurements()` methods of the `MonteCarloLocalization` class in `particle_filter.py`.
Validated my work by running `validate_predicted_measurements()` and `validate_compute_innovations()` from `validate_particle_filter.py`.

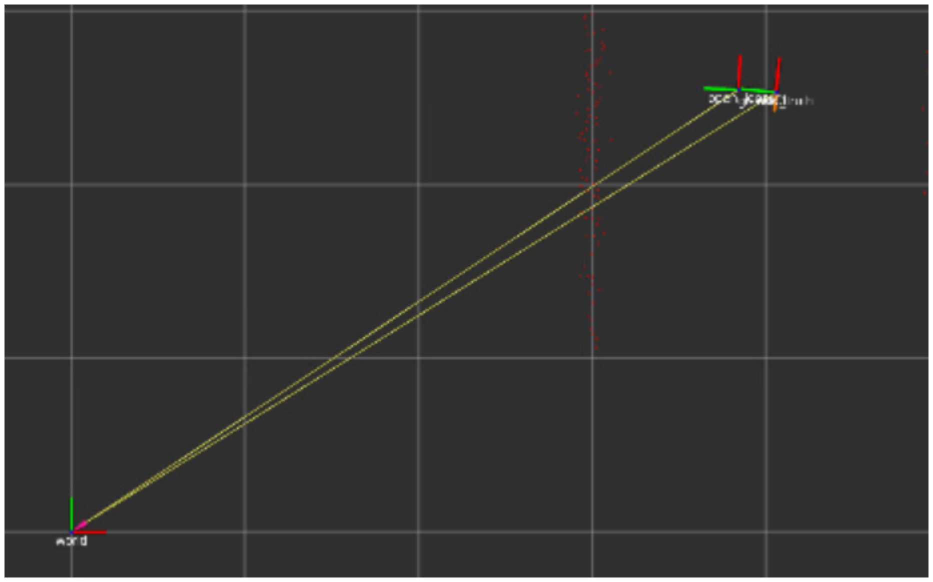
(iii) Implemented the low variance sampling algorithm in the `resample()` method of the `ParticleFilter` class in `particle_filter.py`.
Validated my work by running `validate_resample()` and `validate_mc_localization()` from `validate_particle_filter.py`.
The file `mc_localization.png` is shown below.

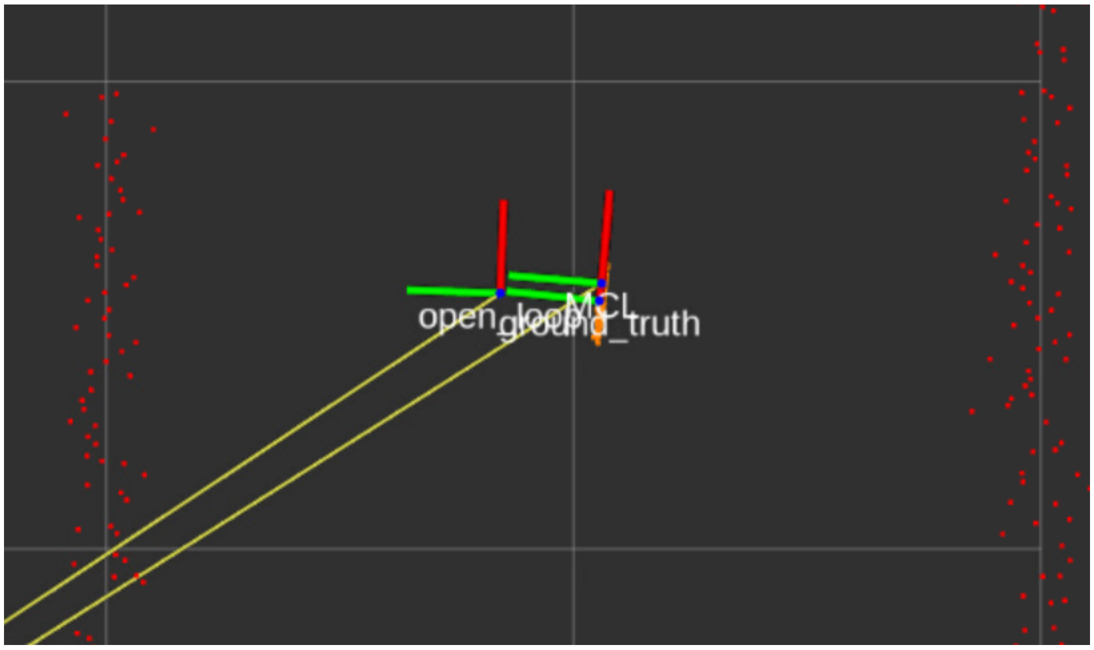


(iv) Just like in EKF, we observe that the MCL state estimates and ground truth diverge with sharp turns or collisions with the walls. But it eventually catches up by correcting for the error. With more particles, the MCL becomes more robust to sharp turns and wall collisions, i.e., the performance improves. The screenshots in RViz (for 1000 particles) are shown below:
(1) The initial state:

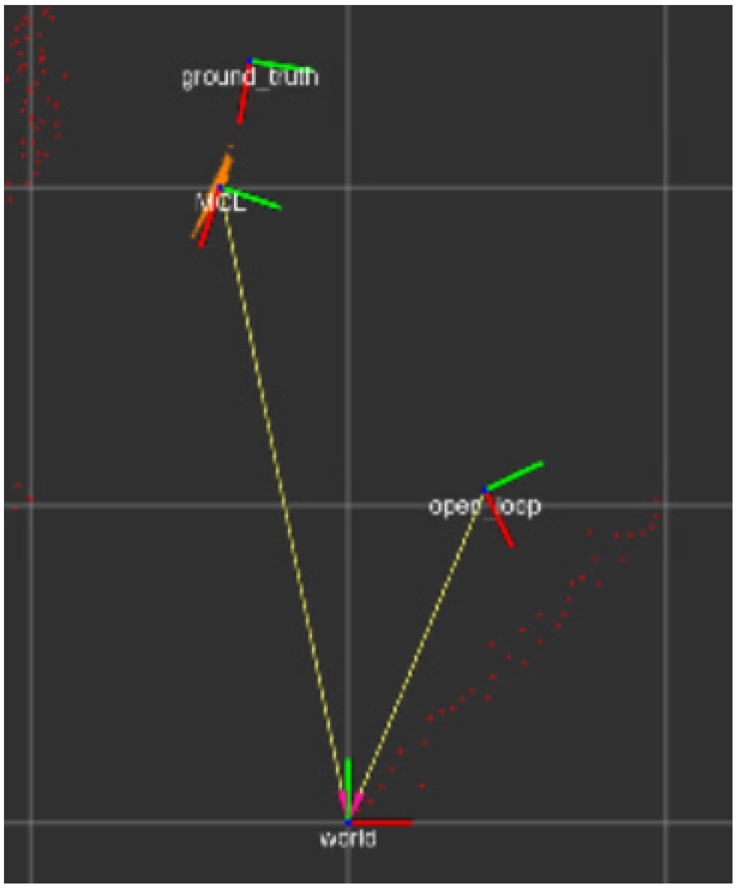


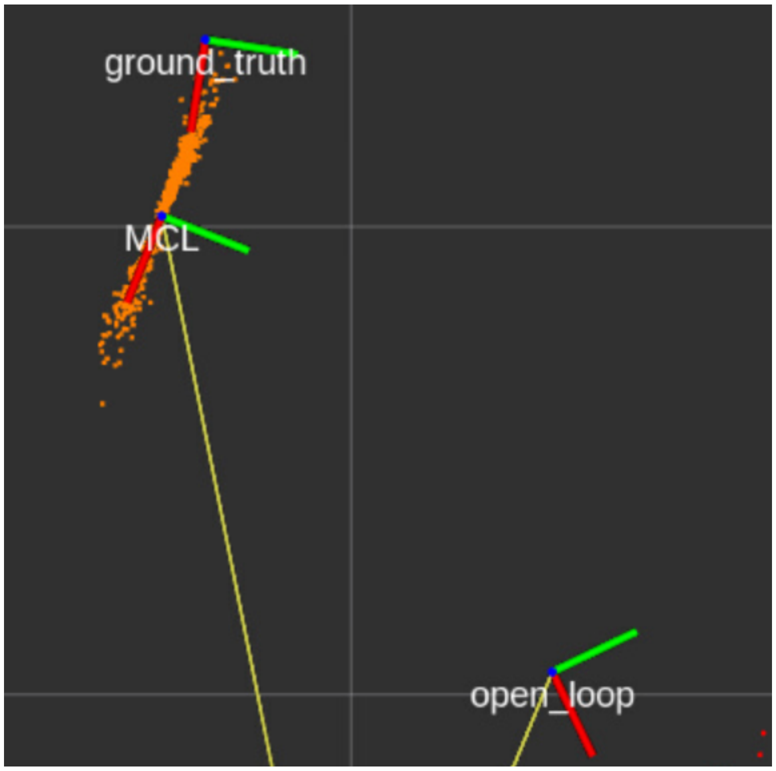
(2) The TurtleBot has moved far from the initial state:





(3) The state estimates have diverged:





(v) Vectorized all the functions and eliminated all loops. The entire program `particle_filter.py` is shown below:

```python
import numpy as np
import scipy.linalg  # You may find scipy.linalg.block_diag useful
import scipy.stats  # You may find scipy.stats.multivariate_normal.pdf useful
from . import turtlebot_model as tb

EPSILON_OMEGA = 1e-3

class ParticleFilter(object):
    """
    Base class for Monte Carlo localization and FastSLAM.

    Usage:
        pf = ParticleFilter(x0, R)
        while True:
            pf.transition_update(u, dt)
            pf.measurement_update(z, Q)
            localized_state = pf.x
    """

    def __init__(self, x0, R):
        """
        ParticleFilter constructor.

        Inputs:
            x0: np.array[M,3] - initial particle states.
             R: np.array[2,2] - control noise covariance (corresponding to dt = 1 second).
        """
        self.M = x0.shape[0]  # Number of particles
        self.xs = x0  # Particle set [M x 3]
        self.ws = np.repeat(1. / self.M, self.M)  # Particle weights (initialize to uniform) [M]
        self.R = R  # Control noise covariance (corresponding to dt = 1 second) [2 x 2]

    @property
    def x(self):
        """
        Returns the particle with the maximum weight for visualization.

        Output:
            x: np.array[3,] - particle with the maximum weight.
        """
        idx = self.ws == self.ws.max()
        x = np.zeros(self.xs.shape[1:])
        x[:2] = self.xs[idx,:2].mean(axis=0)
        th = self.xs[idx,2]
        x[2] = np.arctan2(np.sin(th).mean(), np.cos(th).mean())
        return x

    def transition_update(self, u, dt):
        """
        Performs the transition update step by updating self.xs.

        Inputs:
            u: np.array[2,] - zero-order hold control input.
            dt: float        - duration of discrete time step.
        Output:
            None - internal belief state (self.xs) should be updated.
        """
        ########## Code starts here ##########
        # TODO: Update self.xs.
        # Hint: Call self.transition_model().
        # Hint: You may find np.random.multivariate_normal useful.

        us = np.random.multivariate_normal(u, self.R, self.M)
        self.xs = self.transition_model(us, dt)

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

    def transition_model(self, us, dt):
        """
        Propagates exact (nonlinear) state 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.
        """
        raise NotImplementedError("transition_model must be overridden by a subclass of EKF")

    def measurement_update(self, z_raw, Q_raw):
        """
        Updates belief state according to the given measurement.

        Inputs:
            z_raw: np.array[I,2]   - matrix of I rows containing (alpha, r)
                                     for each line extracted from the scanner
                                     data in the scanner frame.
            Q_raw: [np.array[2,2]] - list of I covariance matrices corresponding
```

```python
                                    to each (alpha, r) row of z_raw.
        Output:
            None - internal belief state (self.x, self.ws) is updated in self.resample().
        """
        raise NotImplementedError("measurement_update must be overridden by a subclass of EKF")

    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.

        c = np.cumsum(ws)
        m = np.arange(0, self.M)
        u = c[-1] * (r + m / self.M)
        idx = np.searchsorted(c, u, side='left')  # indices corresponding to resampled states
        self.xs = xs[idx]
        self.ws = ws[idx]

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

    def measurement_model(self, z_raw, Q_raw):
        """
        Converts raw measurements into the relevant Gaussian form (e.g., a
        dimensionality reduction).

        Inputs:
            z_raw: np.array[I,2]   - I lines extracted from scanner data in
                                     rows representing (alpha, r) in the scanner frame.
            Q_raw: [np.array[2,2]] - list of I covariance matrices corresponding
                                     to each (alpha, r) row of z_raw.
        Outputs:
            z: np.array[2I,]   - joint measurement mean.
            Q: np.array[2I,2I] - joint measurement covariance.
        """
        raise NotImplementedError("measurement_model must be overridden by a subclass of EKF")


class MonteCarloLocalization(ParticleFilter):

    def __init__(self, x0, R, map_lines, tf_base_to_camera, g):
        """
        MonteCarloLocalization constructor.

        Inputs:
                       x0: np.array[M,3] - initial particle states.
                        R: np.array[2,2] - control noise covariance (corresponding to dt = 1 second).
                map_lines: np.array[J,2] - J map lines in rows representing (alpha, r).
        tf_base_to_camera: np.array[3,]  - (x, y, theta) transform from the
                                           robot base to camera frame.
                        g: float         - validation gate.
        """
        self.map_lines = map_lines  # Matrix of J map lines with (alpha, r) as rows
        self.tf_base_to_camera = tf_base_to_camera  # (x, y, theta) transform
        self.g = g  # Validation gate
        super(self.__class__, self).__init__(x0, R)

    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 partical
        #       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

        # indices corresponding to abs(om) <= EPSILON_OMEGA
        idx1 = np.where(np.abs(us[:, 1]) <= EPSILON_OMEGA)[0]
        x_new1 = self.xs[idx1, 0] + np.multiply(us[idx1, 0], np.cos(self.xs[idx1, 2])) * dt
        y_new1 = self.xs[idx1, 1] + np.multiply(us[idx1, 0], np.sin(self.xs[idx1, 2])) * dt
        theta_new1 = self.xs[idx1, 2] + us[idx1, 1] * dt
        g1 = np.vstack((x_new1, y_new1, theta_new1)).T

        # indices corresponding to abs(om) > EPSILON_OMEGA
        idx2 = np.where(np.abs(us[:, 1]) > EPSILON_OMEGA)[0]
        x_new2 = self.xs[idx2, 0] + np.multiply(np.divide(us[idx2, 0], us[idx2, 1]),
                                                np.sin(self.xs[idx2, 2] + us[idx2, 1] * dt) -
                                                np.sin(self.xs[idx2, 2]))
        y_new2 = self.xs[idx2, 1] - np.multiply(np.divide(us[idx2, 0], us[idx2, 1]),
                                                np.cos(self.xs[idx2, 2] + us[idx2, 1] * dt) -
                                                np.cos(self.xs[idx2, 2]))
        theta_new2 = self.xs[idx2, 2] + us[idx2, 1] * dt
        g2 = np.vstack((x_new2, y_new2, theta_new2)).T

        # final g matrix
        g = np.zeros((self.M, 3))
        g[idx1, :] = g1
        g[idx2, :] = g2

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

        return g

    def measurement_update(self, z_raw, Q_raw):
        """
        Updates belief state according to the given measurement.

        Inputs:
            z_raw: np.array[I,2]   - matrix of I rows containing (alpha, r)
                                     for each line extracted from the scanner
                                     data in the scanner frame.
            Q_raw: [np.array[2,2]] - list of I covariance matrices corresponding
                                     to each (alpha, r) row of z_raw.
        Output:
            None - internal belief state (self.x, self.ws) is updated in self.resample().
        """
        xs = np.copy(self.xs)
        ws = np.zeros_like(self.ws)

        ########## Code starts here ##########
        # TODO: Compute new particles (xs, ws) with updated measurement weights.
        # Hint: To maximize speed, implement this without looping over the
        #       particles. You may find scipy.stats.multivariate_normal.pdf()
        #       useful.
        # Hint: You'll need to call self.measurement_model()

        vs, Q = self.measurement_model(z_raw, Q_raw)
        ws = scipy.stats.multivariate_normal.pdf(vs, mean=None, cov=Q)

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

        self.resample(xs, ws)

    def measurement_model(self, z_raw, Q_raw):
        """
        Assemble one joint measurement and covariance from the individual values
        corresponding to each matched line feature for each particle.

        Inputs:
            z_raw: np.array[I,2]   - I lines extracted from scanner data in
                                     rows representing (alpha, r) in the scanner frame.
            Q_raw: [np.array[2,2]] - list of I covariance matrices corresponding
                                     to each (alpha, r) row of z_raw.
        Outputs:
            z: np.array[M,2I]  - joint measurement mean for M particles.
            Q: np.array[2I,2I] - joint measurement covariance.
        """
        vs = self.compute_innovations(z_raw, np.array(Q_raw))

        ########## Code starts here ##########
        # TODO: Compute Q.
        # Hint: You might find scipy.linalg.block_diag() useful

        Q = scipy.linalg.block_diag(*Q_raw)

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

        return vs, Q

    def compute_innovations(self, z_raw, Q_raw):
        """
        Given lines extracted from the scanner data, tries to associate each one
        to the closest map entry measured by Mahalanobis distance.

        Inputs:
            z_raw: np.array[I,2]   - I lines extracted from scanner data in
                                     rows representing (alpha, r) in the scanner frame.
            Q_raw: np.array[I,2,2] - I covariance matrices corresponding
                                     to each (alpha, r) row of z_raw.
        Outputs:
            vs: np.array[M,2I] - M innovation vectors of size 2I
                                 (predicted map measurement - scanner measurement).
        """
        def angle_diff(a, b):
            a = a % (2. * np.pi)
            b = b % (2. * np.pi)
            diff = a - b
            if np.size(diff) == 1:
                if np.abs(a - b) > np.pi:
                    sign = 2. * (diff < 0.) - 1.
                    diff += sign * 2. * np.pi
            else:
                idx = np.abs(diff) > np.pi
                sign = 2. * (diff[idx] < 0.) - 1.
                diff[idx] += sign * 2. * np.pi
            return diff

        ########## Code starts here ##########
        # TODO: Compute vs (with shape [M x I x 2]).
        # Hint: Simple solutions: Using for loop, for each particle, for each
        #       observed line, find the most likely map entry (the entry with
        #       least Mahalanobis distance).
        # Hint: To maximize speed, try to eliminate all for loops, or at least
        #       for loops over J. It is possible to solve multiple systems with
        #       np.linalg.solve() and swap arbitrary axes with np.transpose().
        #       Eliminating loops over J results in a ~10x speedup.
        #       Eliminating loops over I results in a ~2x speedup.
        #       Eliminating loops over M results in a ~5x speedup.
        #       Overall, that's 100x!
        # Hint: For the faster solution, you might find np.expand_dims(),
        #       np.linalg.solve(), np.meshgrid() useful.

        hs = self.compute_predicted_measurements()
        I = z_raw.shape[0]  # Number of observed lines
        J = hs.shape[1]  # Number of predicted lines

        mat = np.zeros((self.M, I, J, 2))  # stores vij for each observation and each sample

        # compute difference between observed and predicted values
        alpha_diff = angle_diff(np.expand_dims(z_raw[:, 0], axis=(1, 2)),
                                np.expand_dims(hs[:, :, 0], axis=0))
        r_diff = np.expand_dims(z_raw[:, 1], axis=(1, 2)) - np.expand_dims(hs[:, :, 1], axis=0)
        mat[:, :, :, 0] = alpha_diff.transpose(1, 0, 2)
        mat[:, :, :, 1] = r_diff.transpose(1, 0, 2)

        # changes dimensions of Q_raw from I x 2 x 2 to M x I x 2 x 2
        Q_big = np.repeat(Q_raw[np.newaxis, :, :, :], self.M, axis=0)

        # 'maha_dist' stores the Mahalanobis distance on its diagonals for each
        # observation and each sample. Its dimensions are M x I x J x J
        maha_dist = np.matmul(mat, np.matmul(np.linalg.inv(Q_big), mat.transpose(0, 1, 3, 2)))
        diag_elements = np.diagonal(maha_dist, axis1=2, axis2=3)
        idx = np.argmin(diag_elements, axis=2)  # indices corresponding to minimum distance

        # creating arrays to index from 'mat'
        i1 = np.repeat(np.array(range(self.M)), I)
        i2 = np.repeat(np.array([range(I)]), self.M, axis=0).flatten()

        # indexing from 'mat' to get 'vs'
        vs = mat[i1, i2, idx.flatten(), :].reshape((self.M, I, 2))

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

        # Reshape [M x I x 2] array to [M x 2I]
        return vs.reshape((self.M,-1))  # [M x 2I]

    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,J,2] - 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 frmae 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, J, 2))

        # pose of the camera in the world frame
        x_cam = self.xs[:, 0] + self.tf_base_to_camera[0] * np.cos(self.xs[:, 2]) - \
                self.tf_base_to_camera[1] * np.sin(self.xs[:, 2])
        y_cam = self.xs[:, 1] + self.tf_base_to_camera[0] * np.sin(self.xs[:, 2]) + \
                self.tf_base_to_camera[1] * np.cos(self.xs[:, 2])
        th_cam = self.xs[:, 2] + self.tf_base_to_camera[2]

        # line parameters in the world frame
        alpha = self.map_lines[0, :]
        r = self.map_lines[1, :]

        # broadcast 1D arrays to M x J matrices
        alpha_MJ = np.tile(alpha, (self.M, 1))
        r_MJ = np.tile(r, (self.M, 1))
        x_cam_MJ = np.tile(x_cam.reshape(self.M, 1), J)
        y_cam_MJ = np.tile(y_cam.reshape(self.M, 1), J)
        th_cam_MJ = np.tile(th_cam.reshape(self.M, 1), J)

        # line parameters in the camera frame
        alpha_in_cam = alpha_MJ - th_cam_MJ
        r_in_cam = r_MJ - np.multiply(x_cam_MJ, np.cos(alpha_MJ)) - \
                   np.multiply(y_cam_MJ, np.sin(alpha_MJ))

        # normalizing line parameters
        i1, i2 = np.where(r_in_cam < 0)
        r_in_cam[i1, i2] = -r_in_cam[i1, i2]
        alpha_in_cam[i1, i2] = np.pi + alpha_in_cam[i1, i2]
        alpha_in_cam = (alpha_in_cam + np.pi) % (2 * np.pi) - np.pi

        # final hs of dimensions M x J x 2
        hs[:, :, 0] = alpha_in_cam
        hs[:, :, 1] = r_in_cam

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

        return hs

```