# Part B

I have replicated the example in Lecture 12 using a Turtlebot robot, with fiducial markers for the beacons. You can see some of these markers placed around the Automation lab and the hallway, along the route we used for the lab. An image of one of these markers can be used to estimate the six-degree-of-freedom transformation between the robot and the marker, but for our purposes I’ve just kept and rotation. The variance of the estimate is approximately proportional to the square of the distance between the robot and the marker.

Your task is to implement a particle filter which uses the beacon observations and either odometry and/or commanded velocity/rotation rate to estimate the robot’s location. You are provided with one dataset as a CSV file, containing a “true” position from SLAM, estimated positions from odometry, the commanded forward speed and rotation rate, and the ID and position of any beacon observations. When two beacons are visible to the camera, there are two rows with the same time and position but different beacon observations. Additionally, you are provided with a CSV file containing the location of each beacon. Again, some Python example code is provided.

If you wish, you may use the true position to initialise your particles, but you must not use it after that. For extra credit, make your implementation robust to unknown starting positions.

In the video on Learn, I’m using the range-bearing sensor model from the lecture notes, and my motion model applies the difference between consecutive odometry positions to each particle with some process noise.

The SLAM algorithm used is RTABMAP, which is generally a bit better than gmapping but gets confused around the identical desks. In the example code I’ve just dropped the bits where it jumps around; it’s pretty good the rest of the time.

The SLAM algorithm estimates a 2-D pose for the Turtlebot (x, y; θ), where x and y are in metres and θ is in radians. The pose is of the Turtlebot’s base-frame with respect to the global map-frame. The heading angle, θ, is measured anti-clockwise from the x-axis of the map-frame.

The odometry is estimated from a wheel encoder for each wheel and a gyro using an EKF. Like the SLAM estimates, the odometry provides 2-D poses of the Turtlebot’s base-frame in the map-frame.

The poses of the beacons are estimated in the camera-frame of the Turtlebot. This cameraframe is offset from the base-frame of the Turtlebot. Full 6-D poses are measured but only (x, y; θ) are given. Again x and y are in metres and θ is in radians. The camera-frame x direction is in the Turtlebot’s forward direction of travel, the y direction is to its left, and θ is measured anti-clockwise from the x direction of the base-frame.

For your particle filter to work:

* You will need to understand the difference between the global (map) coordinates and the local (robot) coordinates and how to transform between the two. It is easy to get your coordinate transformations confused so I suggest plotting some data points on graph paper to check. 2
* An accurate motion model. I recommend testing this first without the sensor model. The particles should move in the correct direction and spread out with time.
* An accurate sensor model.

A pose in coordinate frame 1 can be converted to a pose in coordinate frame 2 using

where and are the translations of frame 1 with respect to frame 2 and is the anti-clockwise rotation of frame 1 with respect to frame 2.

The inverse transformation is

Note, when calculating differences between angles it is necessary to choose the smallest angle in the range .

The pose of the robot is its position and orientation. We can describe this with a vector

where , is its position and is its orientation (measured anticlockwise from the x-axis of the reference frame) at time-step . To avoid ambiguity, a pose is defined relative to a reference frame. For this problem we need to consider three reference frames:

* Map frame: This is used to reference the robot’s and beacon’s poses.
* Robot frame: This is used to reference the camera frame.
* Camera frame: This is used to reference the beacon measurements.

For each measurement, we need to update the weights of the particles based on measurements of the beacons. In this case, the measurements are the relative pose of the beacons with respect to the robot.

The joint likelihood function can be decoupled by converting the relative pose measurements into range and bearing. Dropping the timestep subscript, the measured range and bearing to a beacon are:

Note, both of these measurements have uncertainty and that the sensor model is non-linear. Each particle determines its range and bearing to the measured beacon. If the beacon is not identified, it needs to chose the closest beacon. Its weight depends on how close its range and bearing is to the measured values. The particles with the smallest discrepancies get higher weights. The range and bearing measured by the particle are:g

def motion\_model(particle\_poses, speed\_command, odom\_pose, odom\_pose\_prev, dt):

    """Apply motion model and return updated array of particle\_poses.

    Parameters

    ----------

    particle\_poses: an M x 3 array of particle\_poses where M is the

    number of particles.  Each pose is (x, y, theta) where x and y are

    in metres and theta is in radians.

    speed\_command: a two element array of the current commanded speed

    vector, (v, omega), where v is the forward speed in m/s and omega

    is the angular speed in rad/s.

    odom\_pose: the current local odometry pose (x, y, theta).

    odom\_pose\_prev: the previous local odometry pose (x, y, theta).

    dt is the time step (s).

    Returns

    -------

    An M x 3 array of updated particle\_poses.

    """

    M = particle\_poses.shape[0]

    if odom\_pose[0] != odom\_pose\_prev[0]:

        trajectory = np.arctan((odom\_pose[1] - odom\_pose\_prev[1]) / (odom\_pose[0] - odom\_pose\_prev[0]))

    else:

        trajectory = np.pi / 2

    d = np.sqrt(((odom\_pose[1] - odom\_pose\_prev[1]) \*\* 2) + ((odom\_pose[0] - odom\_pose\_prev[0]) \*\* 2))

    phi\_1\_local = np.radians(min((2\*np.pi - (odom\_pose\_prev[2] - trajectory)), (odom\_pose\_prev[2] - trajectory)))

    phi\_2\_local = np.radians(min((2\*np.pi - (odom\_pose[2] - trajectory)), (odom\_pose[2] - trajectory)))

    difference\_x = d \* np.cos(odom\_pose[2] + phi\_1\_local)#odom\_pose[0] -odom\_pose\_prev[0]# + d \* cos(odom\_pose\_prev[2] + phi\_1\_local)) # First column is x.

    difference\_y = d \* np.sin(odom\_pose[2] + phi\_1\_local)#odom\_pose[1] - odom\_pose\_prev[1]# + d \* sin(odom\_pose\_prev[2] + phi\_1\_local)) # Second column is y.

    difference\_theta = (((odom\_pose[2]  + phi\_1\_local + phi\_2\_local)) + np.pi) % (2 \* np.pi) - np.pi#odom\_pose[2] - odom\_pose\_prev[2]# + phi\_1\_local + phi\_2\_local) # Third colum is theta.

    #print(difference\_x, difference\_y, difference\_theta)

    #difference\_theta = min(2\*np.pi - (odom\_pose[2] - (odom\_pose\_prev[2] + phi\_1\_local + phi\_2\_local)),(odom\_pose[2] - (odom\_pose\_prev[2] + phi\_1\_local + phi\_2\_local))) # Third colum is theta.

    mu = 0

    sigma = 0.01

    for m in range(M):

        # Currently is outputting odometry. Need to convert to global?? Add in the noise?

        particle\_poses[m, 0] += difference\_x + mu + randn() \* sigma # First column is x.

        particle\_poses[m, 1] += difference\_y + mu + randn() \* sigma# Second column is y.

        particle\_poses[m, 2] = (particle\_poses[m, 2] + difference\_theta) % 2 \* np.pi + mu + randn() \* sigma # Third colum is theta.

    return particle\_poses

def sensor\_model(particle\_poses, beacon\_pose, beacon\_loc):

    """Apply sensor model and return particle weights. """

Parameters

    ----------

    particle\_poses: an M x 3 array of particle\_poses (in the map

    coordinate system) where M is the number of particles.  Each pose

    is (x, y, theta) where x and y are in metres and theta is in

    radians.

    beacon\_pose: the measured pose of the beacon (x, y, theta) in the

    robot's camera coordinate system.

    beacon\_loc: the pose of the currently visible beacon (x, y, theta)

    in the map coordinate system.

    M = particle\_poses.shape[0]

    particle\_weights = np.zeros(M)

    r = np.sqrt((beacon\_pose[0]) \*\* 2 + (beacon\_pose[1]) \*\* 2)

    phi = arctan2(beacon\_pose[1], beacon\_pose[0])

    for m in range(M):

        r\_m = np.sqrt((beacon\_loc[0] - particle\_poses[m][0]) \*\* 2 + (beacon\_loc[1] - particle\_poses[m][1]) \*\* 2)

        phi\_m = angle\_difference(arctan2(beacon\_loc[1] - particle\_poses[m][1], beacon\_loc[0] - particle\_poses[m][0]), particle\_poses[m][2])

        r\_val = r - r\_m

        phi\_val = angle\_difference(phi, phi\_m)

        particle\_weights[m] = gauss(r\_val) \* gauss(phi\_val)

    return particle\_weights