Week 2 Task

The goal is to implement an Extended Kalman Filter. You are estimating the state of a mobile robot; the state consists of the robot's 2D position and orientation (x,y,theta). The command to the robot consists of a forward translational velocity vel_trans and a rotational velocity vel_ang. The dynamic model of the robot thus is as follows:

```
x(k+1) = x(k) + t * vel_trans * cos(theta)
y(k+1) = y(k) + t * vel_trans * sin(theta)
theta(k+1) = theta(k) + t * vel_ang
```

where t is the length of the time step, which you can assume to be 0.01 seconds. Every time step, the robot will publish both vel_trans and vel_ang. However, keep in mind that the dynamic model is not perfectly accurate. Therefore, you should assume that the error between the model output and the real robot pose is zero mean white noise. The covariance of that noise is 0.0025 for translational components of the state and 0.005 for the rotational component.

In addition to publishing the commanded velocities, the robot can also localize itself with respect to a number of landmarks in the scene. Every time a landmark is within range of the robot's sensors, the robot will record its distance to the landmark, as well as the bearing of the landmark with respect to the robot's orientation. Assuming the robot is at coordinates (xr, yr, thetar) and the landmark is at coordinates (xl, yl), the range and bearing are computed as follows:

```
range = math.sqrt( (xr-xl)*(xr-xl) + (yr-yl)*(yr-yl) )
bearing = math.atan2(yl-yr, xl-xr) - thetar
```

The robot will publish a range and bearing for each of the landmarks that it can see at a given time along with its velocity commands. Due to sensor noise, these measurements will not be perfectly accurate. You should assume zero mean white noise with covariance 0.1 for the range and 0.05 for the bearing measurement.

The GUI:

- You can start the GUI using

```
rosrun state_estimator gui.py
```

- You will see two robots: The red one represents the real robot, the blue one the current pose estimation published by your code
- The real robot has a grey circle that represents its sensor range
- The landmarks are shown as blue stars. When a landmark is in range it turns red
- You should start the GUI first (before the estimator and the robot) and let it run - it will wait for information to be published. If the GUI is too small on your screen, feel free to edit state_estimator/scripts/gui.py to get it to look right for you.

The robot:

- You can start the robot using

```
rosrun  state_estimator robot.py
```

- The robot will start moving in random directions as soon as you start it. Note that you should start your estimator node **before** starting the robot, so that the two are synchronized: if you start the robot first, then the robot will move around by the time the estimator starts, so the start position of (0,0,0) assumed by your estimator will no longer be valid. You can reset the robot by stopping and re-starting it.

Your code:

You must write code for a node in "est.py" that implements an EKF and uses the information above to compute and publish an estimate of the robot's position. Topics are as follows:

- "/sensor_data": on this topic you will receive messages of the type "state_estimator/SensorData" which contain all the information published by the robot. This is all you need to estimate the robot pose. Every time you receive a new message on this topic, you must advance your estimation by one time step using the information received. You only need to advance by a time step when you receive new information on this topic. Furthermore, you can always assume that the length of a time step is 0.01, regardless of how long it actually takes between consecutive messages to arrive.
- "/robot_pose_estimate": on this topic, you must publish messages that contain the current estimate of the robot's pose. You must publish a message every time you advance your estimation by a time step (i.e. whenever receiving a sensor data message on the topic above).
- You can assume that the robot starts at coordinates (0,0,0).
- While the robot is also publishing it's real position at any given time (so it can be plotted by the GUI, described above), your estimator is obviously NOT allowed to subscribe to that topic (and we will change the name of that topic to something unknown when we do our final grading).
- We have given you some starter code - most of the above points have been taken care of for you. All you have to do is fill in the EKF. Check the comments for more information.
- You can start your code using:

```
rosrun state_estimator est.py
```

Expected behavior:

- When no landmark is within range (or you are not updating your model predictions with sensory information), the estimate and the robot can drift apart from each other. This can also happen when only a single landmark is in range (why?).
- However, when two landmarks are visible simultaneously, or when a new landmark is acquired, the estimate should quickly converge to the true robot location, to the point where only one of them is visible.

- Due to noise in both the model and the sensor data the robot and the prediction will rarely be exactly in the same place. However, they should be close at most times.

Tips and additional information

- Computation of innovation in EKF: the State Estimation handout mentions that the prediction and update steps are similar to the KF, after computing the F and H Jacobians. However, there is no need to linearize computation of the "expected" sensor data, since we do have an explicit sensor model in the h(x) function. Thus, the innovation can be computed as nu = y(k+1) - h (x^ (k+1)), where x^ stands for "x hat", or the predicted state.
- Watch out for discontinuities:
    - Make sure there are no singularities in your derivatives (divide by very small numbers etc.) You should only consider sensor data from landmarks if the distance between the landmark and your robot is greater than 0.1.
    - Careful with the computation of the innovation. In this case it represents the difference between two bearings, which means you need to handle wraparound cases (-270° is equivalent to 90°). The innovation should never exceed pi in magnitude.
    - If you do not properly handle these cases, they will cause your robot to jump unexpectedly.