1. Particle Filter Theory

In this section, we will briefly cover the mathematical backing behind the particle filter algorithm. For a more in depth treatment, we highly recommend you check out [1] or [2] below; we use similar notation and formulation.

The general idea of the particle filter is to infer unobservable state variables in a system by recursively conditioning belief on directly observable system variables. In our case, this means that we would like to infer the car's (unobservable) position and orientation in a map, based on available sensor and control data.

1-A Notation

- We refer to position, state, and pose interchangeably, in all cases referring to location (p_x, p_y) and orientation θ in a 2-D environment.
- State (position) at time t: $x_t = \{p_{x,t}, p_{y,t}, \theta_t\}$, X_t is the associated random variable
- Action at time t: $a_t = \{dp_{x,t}, dp_{y,t}, d\theta_t\}$ ← differences for each state component
- Observations at time t: $o_t = \{r_0, r_1, r_2, ..., r_N\}$ where r_i is the measured range in a particular direction - on our case an observation corresponds to a LiDAR reading
- Belief in the state at time t: $Bel(x_t)$ ← a probability distribution over possible states
    - $Bel(x_t = x)$ is the probability that the robot is at pose x at time t
- Time varies from 0 to T: $0 \leq t \leq T$ such that 0 is the start time, T is the most recent time
- *Position tracking:* the problem of tracking changes in system state over time, assuming the initial state is known
- *Global localization:* the harder problem of discovering system state when initial state is not known
- *Kidnapped robot problem:* when a robot in a well known state is teleported to a random different state without being told the new state, similar to global localization

While in some ways the state is inherently discrete (since our map is discrete), it is often useful to represent our state x_t as continuous (float) values. This will make it easier to manipulate the particle distribution since our actions and other values are often float variables as well.

1-B Mathematical Formulation

The MCL algorithm estimates the probability distribution over all possible states, conditioned on all available data:

$$Bel(x_t) = P(x_t \mid a_{0...T}, o_{0...T})$$

Key to the operation of the particle filter is the *Markov assumption*: if one knows system state at a time t future measurements are independent of past ones. This assumption basically states that it is sufficient to track system state, without maintaining an entire history of observations and actions. In addition to this assumption, MCL relies heavily on Bayes' rule, which is:

$$P(A \mid B) = \frac{P(B \mid A) * P(A)}{P(B)}$$

In a particle filter, we typically have two types of available data, sensor readings o_t and odometry/control data a_t. Each of these data will help update our belief distribution, though in slightly different ways. While it may not be strictly true, we safely assume that the data arrives in an alternating fashion.

Supposing the last data to arrive was sensor data, we are interested finding the belief of the current state x_t, given our previous observation and action history:

$$Bel(x_t) = P(x_t \mid o_t,\ a_{t-1}, o_{t-1}, ...,\ o_0)$$

Applying Bayes' rule, we arrive at the following:

$$Bel(x_t) = \frac{P(o_t \mid x_t, a_{t-1}, ..., o_0) * P(x_t \mid a_{t-1}, \dots, o_0)}{P(o_t \mid a_{T-1}, \dots, o_0)} = \eta * P(o_t \mid x_t,\ a_{t-1}, ...,\ o_0) * P(x_t \mid a_{t-1},\ \dots,\ o_0)$$

Since the denominator is a constant with respect to x_t this is usually denoted as simply a constant normalizing factor η which ensures the posterior distribution sums to 1. By applying our markov assumption, we can simplify the above:

$$P(o_t \mid x_t,\ a_{t-1}, ...,\ o_0) = P(o_t \mid x_t)$$

Update equation (1): $$Bel(x_t) = \eta * P(o_t \mid x_t) * P(x_t \mid a_{t-1},\ \dots, o_0)$$

In more plain english, this is the probability of recording a given measurement when at a given state, multiplied by the probability of being at that state given the history. The probability of recording a certain measurement at a given position $P(o_t \mid x_t)$ is determined according to the *sensor model* which you will implement in section 2.B.

Now, supposing the last recorded data is an odometry reading a_{t-1}, we are interested in:

$$Bel(x_t) = P(x_t \mid a_{t-1}, o_{t-1}, ...,\ o_0)$$

We can apply the Law of Total Probability to rewrite this as:

$$Bel(x_t) = \int P(x_t \mid x_{t-1},\ a_{t-1}, o_{t-1}, ...,\ o_0) * P(x_{t-1} \mid a_{t-1}, ..., o_0)\, dx_{t-1}$$

Another application of the markov assumption yields:

$$P(x_t \mid x_{t-1},\ a_{t-1}, o_{t-1}, ...,\ o_0) = P(x_t \mid x_{t-1},\ a_{t-1})$$

$$Bel(x_t) = \int P(x_t \mid x_{t-1},\ a_{t-1}) * P(x_{t-1} \mid a_{t-1}, ..., o_0)\, dx_{t-1}$$

We can substitute the definition of Bel into the above yields the following recursive equation:

Update equation (2): $$Bel(x_t) = \int P(x_t \mid x_{t-1},\ a_{t-1}) * Bel(x_{t-1})\, dx_{t-1}$$

The remaining conditional probability term in (2) is referred to as the *motion model,* which you will implement. It describes the probability of being at a given state, given the most recent odometry data and the previous probability distribution over the state space. Finally, plugging (2) back into update equation (1), we arrive at the most important recursive equation:

Update equation (3): $$Bel(x_t) = \eta * P(o_t \mid x_t) * \int P(x_t \mid x_{t-1},\ a_{t-1}) * Bel(x_{t-1})\, dx_{t-1}$$

Now that we have discussed the mathematical basis, some questions remain as to how we translate to an actual algorithm. As previously defined, our state space is defined as position and orientation in a map. Unfortunately, this contains a large (potentially infinite) number of states. It is immediately clear that attempting to represent $Bel(x_t)$ for every state is computationally problematic, even for a coarse discretization. Instead of representing belief over some state discretization, the particle filter algorithm approximates the belief distribution as a set of m weighted samples according to $Bel(x_t)$.

$$Bel(x_t) \approx \{ x_t^i, w_t^i \} \ \forall \ 0 \leq i < m$$

In this approximation, x^i is a hypothesis pose (often referred to as a particle), and w^i is an importance weight for that hypothesis. The initial set of particles is determined according any knowledge about the initial state of the system. In the *position tracking* case, which you should focus on, you might initialize the distribution by creating a mass of particles randomly distributed in the small region of the state around the known initial position. In *global localization* where nothing is known about initial state, the simplest thing to do is to distribute particles uniformly at random over the permissible regions of the state space with uniform weights $\frac{1}{m}$. Unfortunately, this method does not scale well with size of the state space. If no particles are distributed near the real state, it is likely the particle filter will fail to converge. Methods exist for dealing with these problems, such as Mixture MCL as discussed in [1], but it is not required for this lab.

1-C The Particle Filter Algorithm

```
# a single recursive update step of the MCL algorithm
def MCL(X_{t-1}, a_{t-1}, o):
    X_t = {}
    for i in range(m):
        # sample a pose from the old particles according to old weights. Samples
        # implicitly represent the prior prob. dist. Bel(x_{t-1}) in eqn. (3)
        x^i_{t-1} ~ X_{t-1}
        # update the sampled pose according to the motion model
        x^i_t ~ p(x_t | x_{t-1}, a_{t-1})
        # weight the updated pose according to the sensor model
        w^i_t = p(o | x^i_t)
        # add the new pose and weight to the new distribution
        X_t = X_t ∪ { (x^i_t, w^i_t) }
    # normalize weights, should sum to 1
    X_t = normalize(X_t)
    return X_t

# iterative application of the MCL algorithm
def particle_filter():
    X = Bel(x_o) ← initial particles
    while true:
        a = get_last_odometry()
        o = get_last_sensor_readings()
        X = MCL(X,a,o)
        # inferred pose ← expected value over particle distribution
        pose = Ex[X]
```

2. Implement your particle filter

2-A Motion Model

$$x^i_t \sim p(x_t \mid x_{t-1}, a_{t-1})$$

Your motion model should take as arguments the old particle pose, as well as the last odometry data, and return a new pose with the odometry "applied" to the old poses. For example, if you have an initial state (10,10,0) and record odometry data indicating a state change of (1,0,0) , your motion model might return (11,10,0). Of course, you do not have perfect odometry data - there is presumably some degree of noise in your measurements. To model this, you will also mix in some degree of randomness which represents your measurement uncertainty. In the aforementioned example, you might for example mix in a randomly chosen noise term of (0.1,-0.1,-0.05), and return (11.1,-0.1,-0.05).

One of the excellent things about particle filters is their flexible nature. Your car may or may not have a working IMU, but it certainly has rough odometry data based on the motor and steering commands sent to the car. You may assume that the motor commands relate to the movement of your car according to the geometric Ackermann model, with some noise due to tire slip and other imperfections. The `/vesc/odom` topic does this for you already on the car.

The precise definition of your motion model is up to you. You might empirically determine your noise coefficients based on what works, or could try to gather data from the car which allows you to directly determine your measurement uncertainty. You will also have to model how your available odometry data (i.e. steering commands speed and steering) correspond to changes in states (x,y, θ). If you have an IMU, you may be able to directly gather the delta values for each of the state coefficients, but you will still have to ensure that your unit and coordinate space conversions are correct.

2-B Sensor Model

$$w^i_t = p(o \mid x^i_t)$$

Once you have updated particle positions via your motion model, you use the sensor model to assign likelihood weights to each particle. This allows good hypotheses to have better likelihood of being sampled on the next iteration, and visa versa for bad hypotheses. In general, the sensor model should define how likely it is to record a given sensor reading from a hypothesis position in the map. The definition of this likelihood is strongly dependent on the type of sensor used - a laser scanner in your case.

Typically, we make the assumption that range measurements at different angles are independent of each other. Clearly, this is not strictly valid, but it simplifies things and is reasonable, especially if you downsample the number of scanner ranges considered. You should determine probability weights for individual range measurements, and then multiplicatively accumulate these weights to determine the total weight for a single particle.

$$p(o \mid x^i_t) = \prod_j p(o_j \mid x^i_t) \ \forall \ j \in \{subsampled\ lidar\ angles\}$$

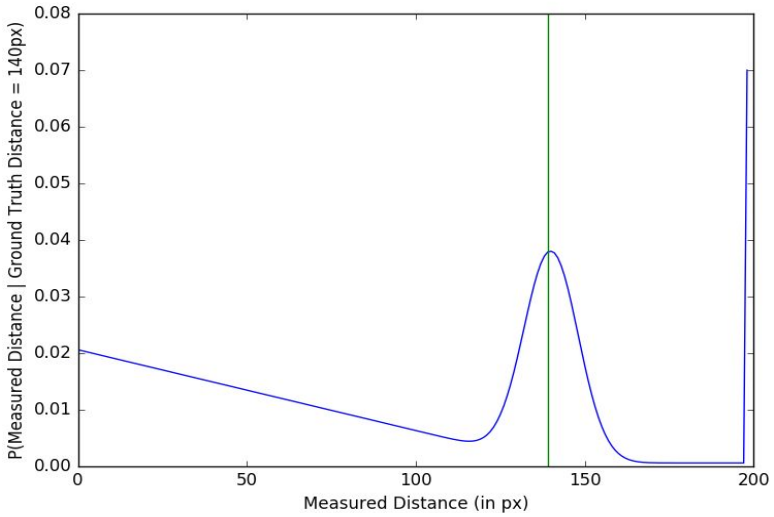
Typically, there are a few cases to be modeled:

a. Probability of detecting a known obstacle in the map
b. Probability of a short measurement, maybe due to unknown obstacles (cats, people, etc)
c. Probability of a missed measurement, usually due to a reflected LiDAR beam
d. Probability of a random measurement, maybe due to unexpected asteroid collisions

We typically represent (a) as a gaussian distribution around the ground truth distance between the hypothesis pose and the known map obstacle. Thus, if the measured range exactly matches the expected range, the probability is large. This is a gaussian to allow sensor readings which are close but not exactly the same as anticipated ranges to have high weights. In other words, if the measured range is r and the ground truth range is determined (via ray casting) to be d:

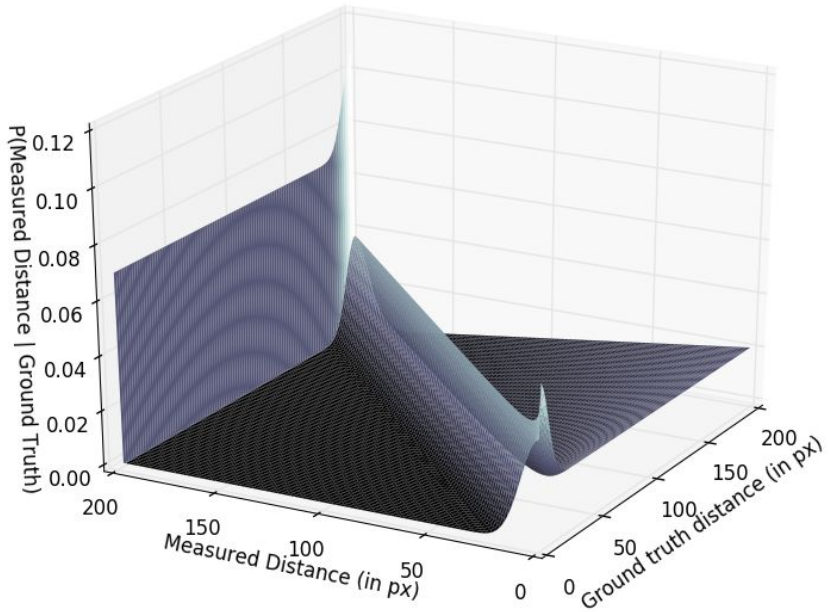
$$p(r|d) = \frac{1}{\sigma\sqrt{2\pi}} * e^{-\frac{(r-d)^2}{2\sigma^2}}$$

(b) is represented as a downward sloping line as the ray gets further from the robot. For a in depth analysis of the basis behind this, see [2]. (c) is represented by a large spike in probability at the maximal range value, so that reflected measurements do not significantly discount particle weights. Finally, (d) is represented by a small uniform value, to account for unforeseen influences. While it is not strictly necessary for your particle filter to function (since the normalization step reweights particles), ideally $p(r|d)$ integrates to 1, given that it is a probability. It can be tricky to define the sum of four terms such that it always adds to 1, so an easy way to handle this is to simply normalize the discretized model after precomputation (see below).



Prob. of measuring various distances given that computed distance is 140px in TA code. Left slope is prob. of short measurement. Peak around 140px is prob. of true measurement. Right peak is the prob. of erroneous max range measurement. Uniform value added (see ~190px). Empirical model parameters.

Precomputing the model



Full precomputed sensor model table in the TA implementation (above)

Once you have defined your sensor model $p(r|d)$, which provides the likelihood of recording a range measurement, given the ground truth range, you should precompute a two dimensional table of probability values for some discretization of r and d. The motivation is twofold: 1) discretization provides an easy numerical method of normalizing the probability distribution, 2) with only a small loss of accuracy it reduces the evaluation of your sensor model to a ray cast followed by a table lookup (as described in section 3.4.1 of [2]).

```
def p_r_given_d(r,d):
    # compute terms (a,b,c,d) and normalize weights
    assert(0<=r,d<=max_range)
    return lots_of_math() # exponentials, normalization, etc
```

Becomes…

```
p_r_given_d_table = precompute_sensor_model()
def p_r_given_d(r,d):
    assert(0<=r,d<=max_range)
    return p_r_given_d_table[discretize(r),discretize(d)]
```

And your sensor model is something like:

```
range_method = range_lib.[SomeRangeMethod](map_msg, max_range_in_pixels)

# this sensor model is defined for a single particle (px,py,theta)
# however, in practice, you will find it more efficient to define your
```

```
# sensor model for the entire array of particles instead
def sensor_model(px,py,theta,ranges):
    # queries is the set of (x,y,theta) poses to ray cast from
    queries = make_query_states(px,py,theta)
array
    ground_ranges = np.zeros(queries.shape[0])
    range_method.calc_range_many(queries,ground_ranges)
    # compute weight according to section 2B
    partial_weights = map(lambda r,d:p_r_given_d(r,d),ranges,ground_ranges)
    weight = product(partial_weights)
    # do other things to the weight...
    return weight
```

The above pseudocode for a sensor model is unnecessarily slow - think about how it could be improved. See below, and the documentation for RangeLibc for further discussion.

2-C Particle Resampling

$$x^{i}_{t-1} \sim X_{t-1}$$

In the resampling step of your particle filter, you should randomly draw from the set of particles according to the weights computed in the previous iteration. This set is called the proposal distribution. Notice that you will very likely (à la birthday paradox) draw the same particle more than once, but *this is ok* because your motion model incorporates randomness that will ensure we don't evaluate the observation model with several copies of the same particle.

You may find the `numpy.random.choice` function useful.

2-D Putting it all together

Once you have defined your motion and sensor models, you should implement the recursive algorithm provided above. If you are unsure how to proceed even after reading this guide, take a look at the references below. Many excellent resources on particle filters exist if you search for them, but just remember to **cite your sources.**

You should use the extensive visualization tools available to you in ROS and RViz to publish visualizable messages. This is required for the deliverable videos, but it will also be a huge aid in debugging, so we recommend you add visualization early and often.

2-E Hacks and other necessaries

If you implement the particle filter exactly as described above, you will likely discover that it has very poor performance. Largely, this poor performance is due to assumptions made during the derivation of the particle filter which are not strictly true.

For example, the particle distribution approximates the true distribution in the limit - with an infinite number of particles. Clearly this is problematic since we maintain at max a few thousand particles. You should take a look at [4,5] for some useful tips for improving real world problems.

You will almost certainly want to downsample the number of range measurements from the particle filter from 1080 to less than 100. This will make the probability distribution over your state space less “peaked” and increase the number of particles you can maintain in real time. Additionally, you will probably want to “squash” your importance weight by raising them to a power of less than one (⅓ for example) to make your distribution even less peaked. If your are confused by this paragraph, look at [4,5]. You may notice that the TA sensor model shown above is not as peaked as you might expect: this is largely to smooth the probability distribution over the particle state space - we also further “squash” our particle weights for the same reason.

3. Bonus: Map a New Environment

If you have extra bandwidth after (or while) completing the particle filter, you should attempt to use an existing mapping package like gmapping or hector_slam on the RACECAR to map out a new environment. This is not a requirement of the lab, but a definite bonus.

There are two general components to creating a map using one of these packages:

1. Collecting laser scanner data in a bag file
2. Running the slam program over the laser scan data to generate the map

What follows is a quick start guide to gmapping.

3-A Changing coordinate frames

Before we begin, we need to change the default parameters of a launch file. The frames chain in ROS is as follows.

`map` → `odom` → `base_link` → `laser`

The coordinate transformation between `odom` → `base_link` is given by `odometer` and published to topic `/tf`, while the transformation between `base_link` → `laser` is published by `static_transform_publisher` to the topic `/tf_static`. However, gmapping only subscribes to the topic `/tf` but not `/tf_static`. Therefore, as a temporary, hacky workaround to ensure that the data we collect is readable by the gmapping module, we discard the `laser` frame and “assume” that the two coordinate frames `base_link` and `laser` are exactly the same. This requires changing the `frame_id` of the `laser` scan data from `laser` to `base_link`.

Navigate to `racecar/racecar/launch/includes/common/sensors.launch.xml` In this file, there is a node called ”urg node” that launches the laser. Edit this node to specify another param.

```
<node pkg="urg_node" type="urg_node" name="laser_node" >
  <param name="frame_id" value="base_link" />
</node>
```

3-B Collecting rosbag data

Run teleop.launch on the racecar.

We want to collect data from the laser scan, the odometry, and the transforms between the different frames using rosbag. It is heavily recommended that you mount and store rosbag files on your ssd since the files tend to get large.

`rosbag record -O [ros_bag_file_name] /tf /tf_static /scan /vesc/odom.`
Use the joypad to drive at a **low speed** (to avoid tilting the car) around the desired area. Avoid placing moving objects in the laser scan field of view while bagging since these may confuse the SLAM algorithm.

3-C Mapping

Install the gmapping module onto your racecar.
`sudo apt-get install ros-kinetic-slam-gmapping`
Afterwards, start the gmapping module. The following launch file provides a couple of tuned parameters to get started with, but play around with the values and explore many more parameters available here.

```
<launch>
  <param name="use_sim_time" value="true"/>
    <node pkg="gmapping" type="slam_gmapping" name="slam_gmapping">
      # you may find other values for these parameters give better results
      <param name="linearUpdate" value="0.1"/>
      <param name="angularUpdate" value="0.1"/>
      <param name="particles" value="100"/>
      <param name="maxRange" value="9.5"/>
    </node>
</launch>
```

Since there's no real-time clock on the racecar, the timestamps of the bag file we collected might be in the year 1970 (depending on whether we boot up the racecar with internet connections). All messages in the bag file will be discarded if we don't set `use_sim_time` to true, as was done in the launch file, and play the rosbag below with the option --clock.

Additionally, gmapping will not easily be able the update the map with sharp detail over 100 particles and concurrently keep up with incoming laser scan message. Therefore, we will slow down the rate at which the rosbag is played to a tenth of it's current speed using -r.

```
rosbag play [ros_bag_file_name] --clock -r 0.1
```

3-D Viewing the Map

User rviz in your vm to visualize the map by adding the Map topic, and save the map on your racecar using.

```
rosrun map_server map_saver -f [output_file_name]
```

You can scp the map file to your laptop or VM for viewing. If you wish to use this map in the particle filter with RangeLibc, you will have to make sure the map metadata is accurate, especially with respect to coordinate space conversions (origin, resolution, etc). If all is well, you should be able to localize and see laser scans from the car align with walls in your map.

4. Code Review

An important aspect of software engineering is the process of reviewing your peers' code. No matter how proficient you are at designing and writing software, there are always tricks and better practices that you can pick up by a) reading other's code, and b) having others read and comment on your code. Efficiency is critical on this lab, so review should help you.

Stay tuned for more information on the logistics of the code review process.

5. Notes and Tips

5-A Writing efficient Python

The particle filter algorithm is relatively straightforward, but it can nevertheless be quite challenging to implement efficiently. Since the algorithm must run at >20Hz with a large number of particles, an efficient implementation is a requirement for success. There are a few tricks you can use, primarily:

- Use numpy arrays for absolutely everything - python lists → slow
- Use numpy functions on numpy arrays to do any computations
    - i.e. avoid Python for loops like the plague
        - If it helps, write your functions initially using for loops. Once it works, figure out how to use equivalent vectorized numpy calls without the loops
    - Slice indexing is your (best) friend.
- Use the smallest number of operations required to perform your arithmetic
    - Once again, it can help to do a logically simple but inefficient "first draft", followed by a better version which ditches unnecessary computation/memory reads
- Avoid unnecessary memory allocations
    - Cache and reuse important numpy arrays by setting them to the right size during initialization of your particle filter as "self" variables
        - Be careful! Avoid bugs due to stale state in reused variables!
- Identify your critical code paths, and keep them clean
    - Conversely, don't worry too much about code that is called infrequently
- Push code to Cython/C++ if necessary
    - You probably won't need to do this much since RangeLibc already does this
- Avoid excessive function calls - function call overhead in Python → slow
- **Use RangeLibc**
    - saveTrace with Bresenham's Line can be useful for debugging (see docs)
- Don't publish visualization messages unless someone is subscribed to those topics
    - `if self.[some_pub].get_num_connections() > 0 ...`

- **USE A PROFILER** to identify good candidates for optimization
    - http://projects.csail.mit.edu/pr2/wiki/index.php?title=Profiling_Code_in_ROS

5-B RangeLibc

Assuming a reasonably efficient architecture, you will quickly find that the bottleneck computation in running your particle filter is the determination of ground truth ranges between hypothesis poses (particles) and the nearest obstacles in the map. Rather than having you implement the "calc_range" function yourself in Python, we provide the RangeLibc library to enable real time particle filter operation. See this document for an algorithm overview and usage instructions. **We highly recommend you read through the RangeLibc documentation** since it includes tips to make your sensor model implementations significantly simpler and faster.
f

5-C RViz is your friend

- Visual inspection is the best debugging tool
    - Significantly better than looking at a stream of numbers in a terminal
    - Add visualization early and often
- Subscribe to the `/initialpose` or `/clicked_point` topics and use the associated tools in RViz ("2D Pose Estimate"/"Clicked Point") to initialize or reinitialize your particle filter (see video example of this)
- Matplotlib can be convenient for quick and dirty visualization of numpy related variables if you just need to look at something once

6. Download starter code

Skeleton code

```
cd ~/racecar-ws/src/
curl -L "https://docs.google.com/uc?id=0ByI4e_M8xttAbWpxaTNfVDMzblk&export=download" | tar xvz
cd ~/racecar-ws/
sudo apt-get update
# this is necessary to use the map_server
rosdep install -r --from-paths src --ignore-src --rosdistro kinetic -y
catkin_make
```

RangeLibc (see documentation for more info)

```
cd lab5/range_libc/pywrapper
sudo pip install cython
# on VM
sudo python setup.py install
# on car - compiles GPU ray casting methods
sudo WITH_CUDA=ON python setup.py install
```

If we ask you to update RangeLibc, simply cd into lab5/range_libc and do git pull
Latest version hosted here: https://github.com/kctess5/range_libc

7. References

1. S. Thrun, D. Fox, W. Burgard and F. Dellaert. “Robust Monte Carlo Localization for Mobile Robots.” *Artificial Intelligence Journal*. 2001.
    - http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.71.6016&rep=rep1&type=pdf
2. D. Fox, W. Burgard, and S. Thrun. “Markov localization for mobile robots in dynamic environments,” *Journal of Artificial Intelligence Research*, vol. 11, pp. 391427, 1999.
    - http://www.jair.org/media/616/live-616-1819-jair.pdf
3. D. Fox. “KLD-sampling: Adaptive particle filters,” *Advances in Neural Information Processing Systems* 14 (NIPS), Cambridge, MA, 2002. MIT Press.
    - https://papers.nips.cc/paper/1998-kld-sampling-adaptive-particle-filters.pdf
4. D. Bagnell “Particle Filters: The Good, The Bad, The Ugly”
    - http://www.cs.cmu.edu/~16831-f12/notes/F14/16831_lecture05_gseyfarth_zbatts.pdf
5. B. Boots “Importance Sampling, Particle Filters”
    - http://www.cc.gatech.edu/~bboots3/STR-Spring2017/Lectures/Lecture4/Lecture4_notes.pdf
    - Similar to [4] but slightly more in depth