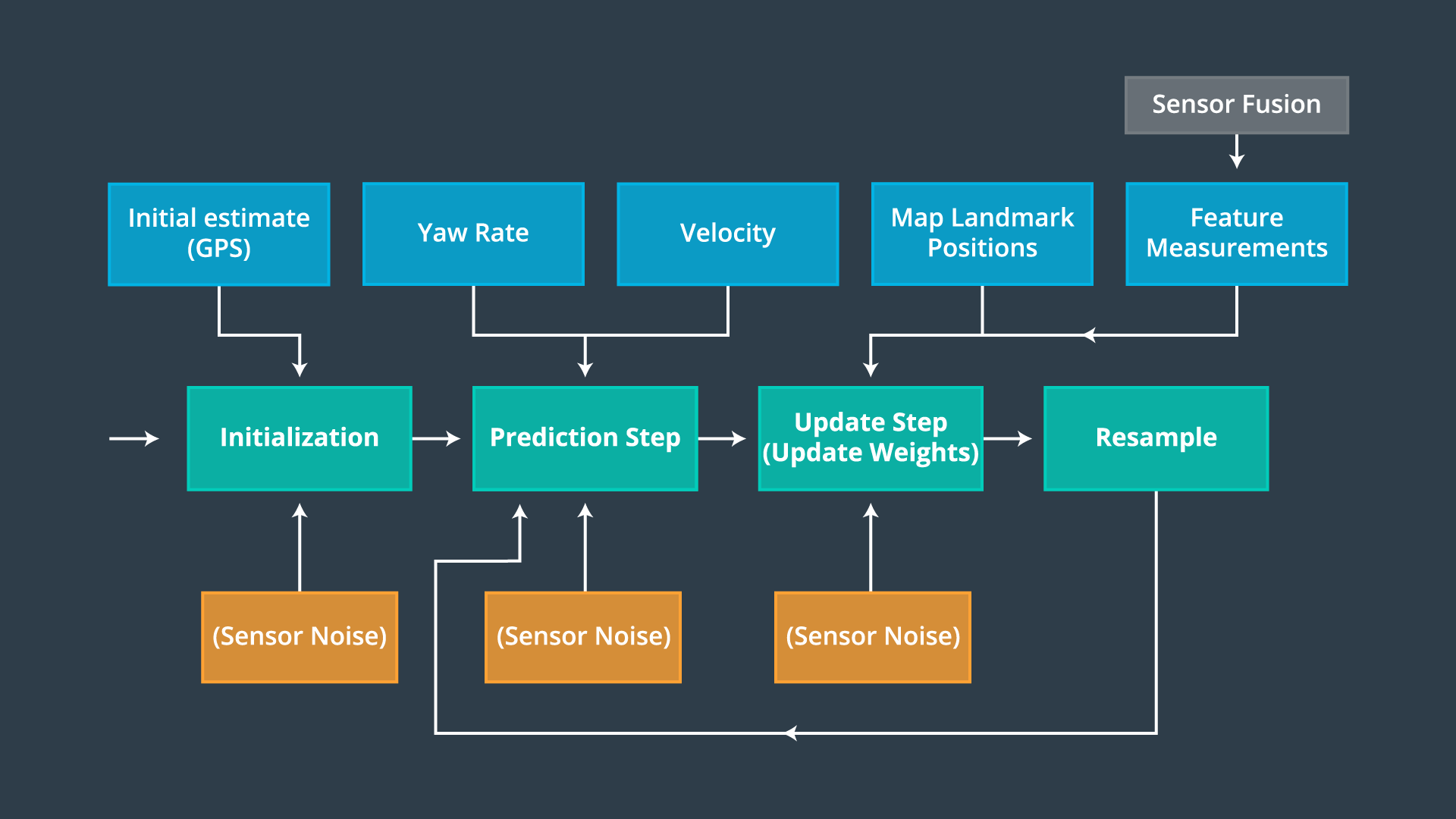
**Process and Implementation**

As an accompaniment to the videos we will follow the particle filter algorithm process and implementation details.

**Particle Filter Algorithm Steps and Inputs**

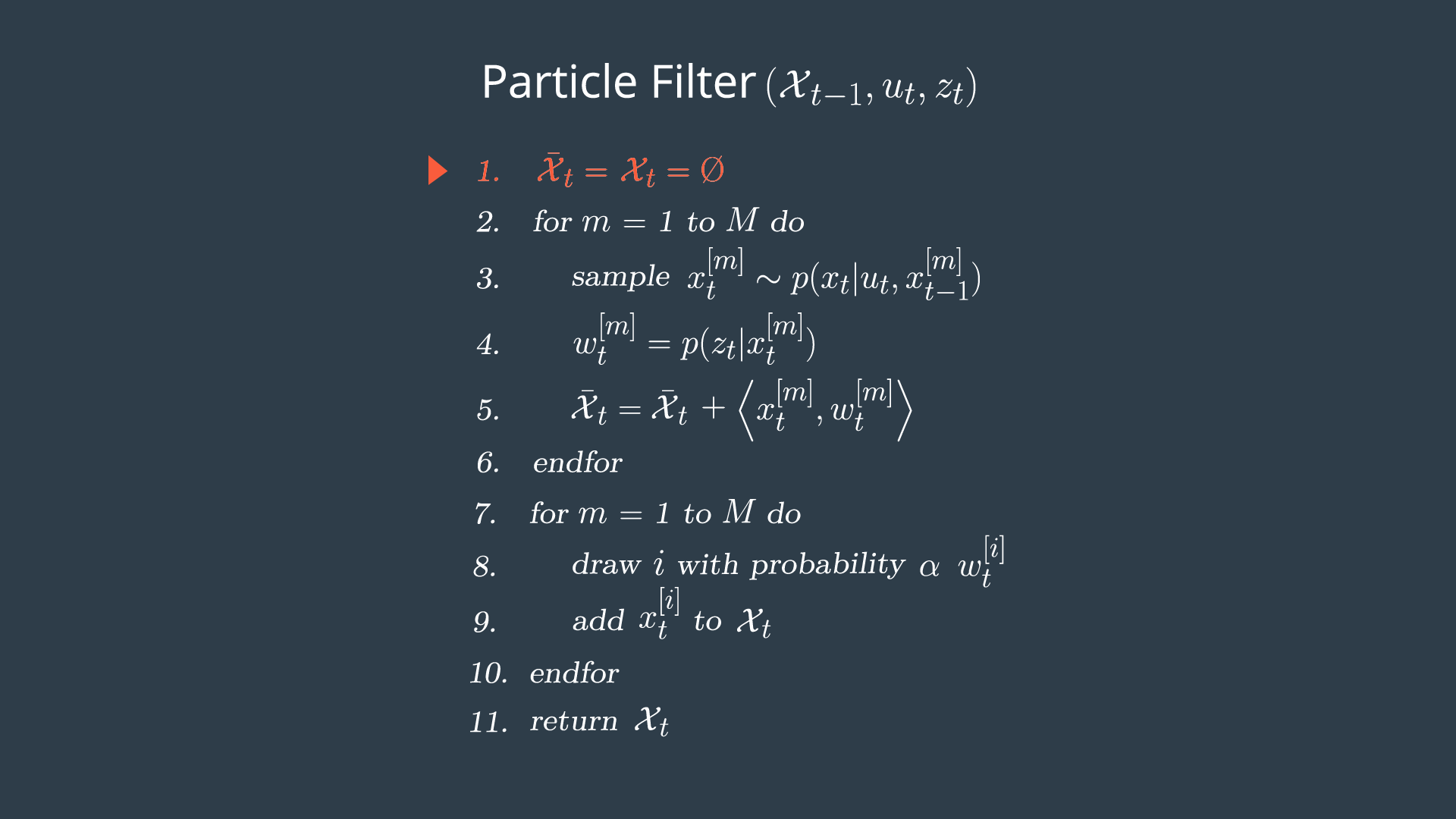
The flowchart below represents the steps of the particle filter algorithm as well as its inputs.



Particle Filter Algorithm Flowchart

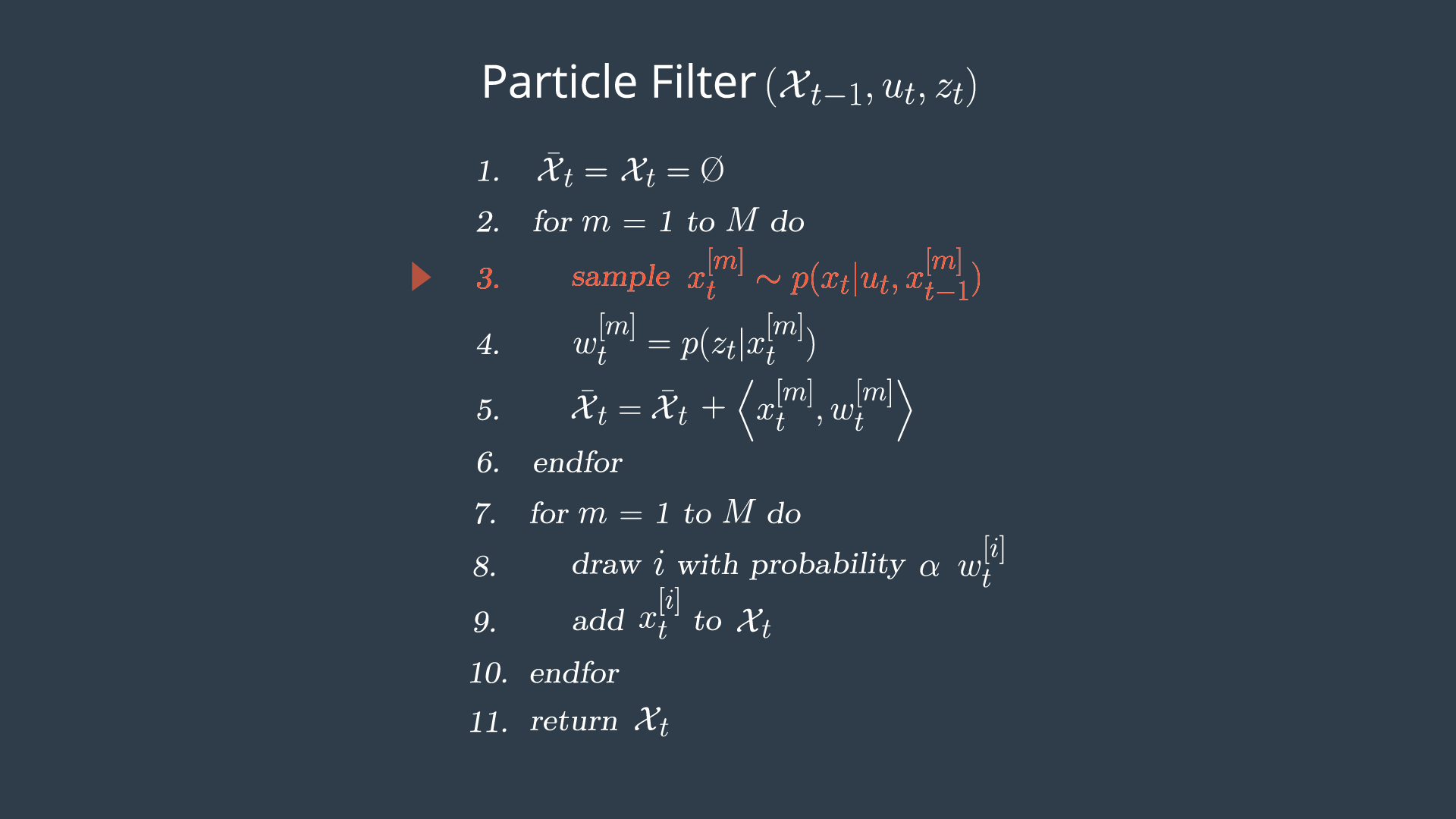
**Psuedo Code**

This is an outline of steps you will need to take with your code in order to implement a particle filter for localizing an autonomous vehicle. The pseudo code steps correspond to the steps in the algorithm flow chart, initialization, prediction, particle weight updates, and resampling. Python implementation of these steps was covered in the previous lesson.



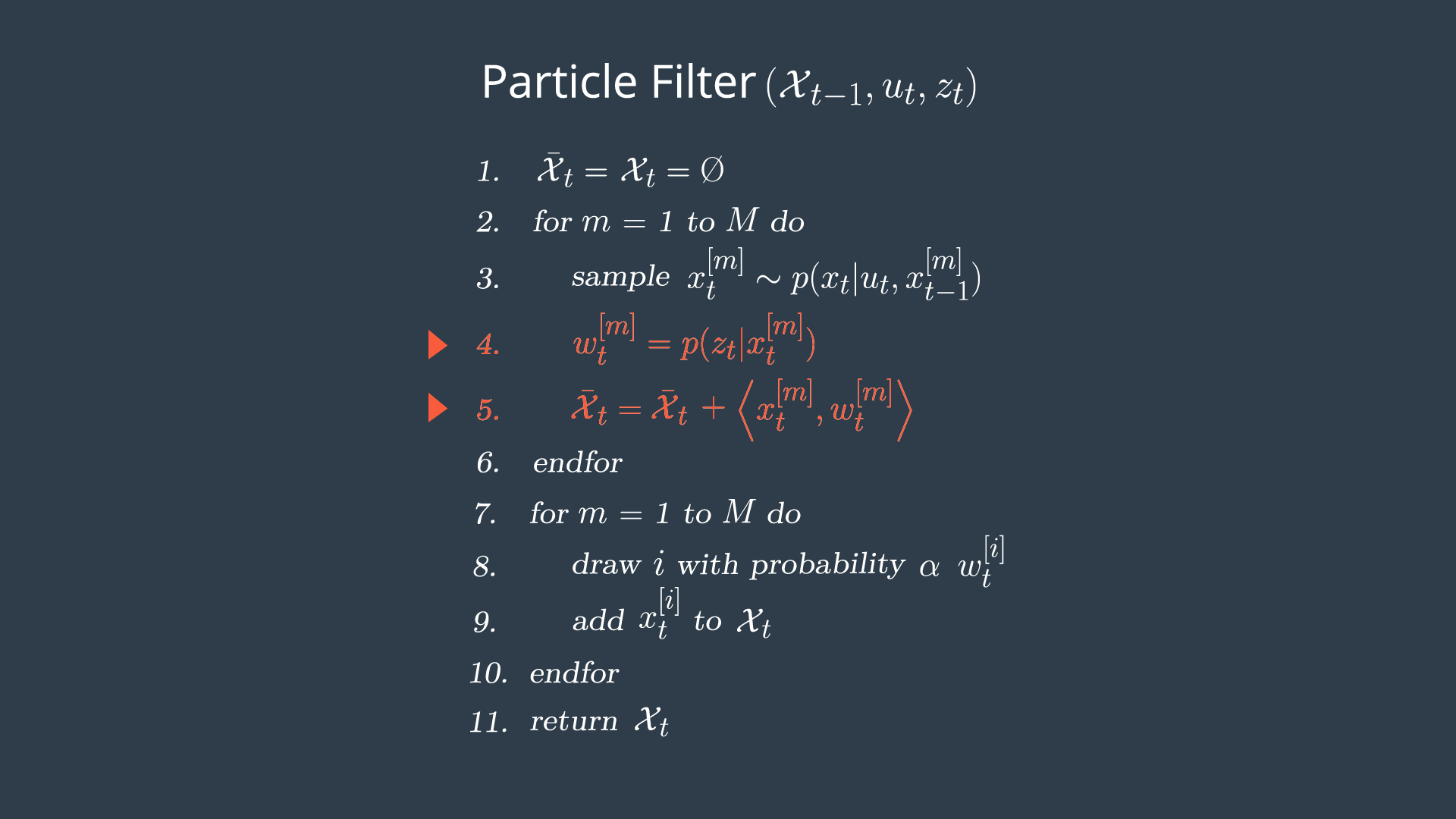
Initialization

At the initialization step we estimate our position from GPS input. The subsequent steps in the process will refine this estimate to localize our vehicle.



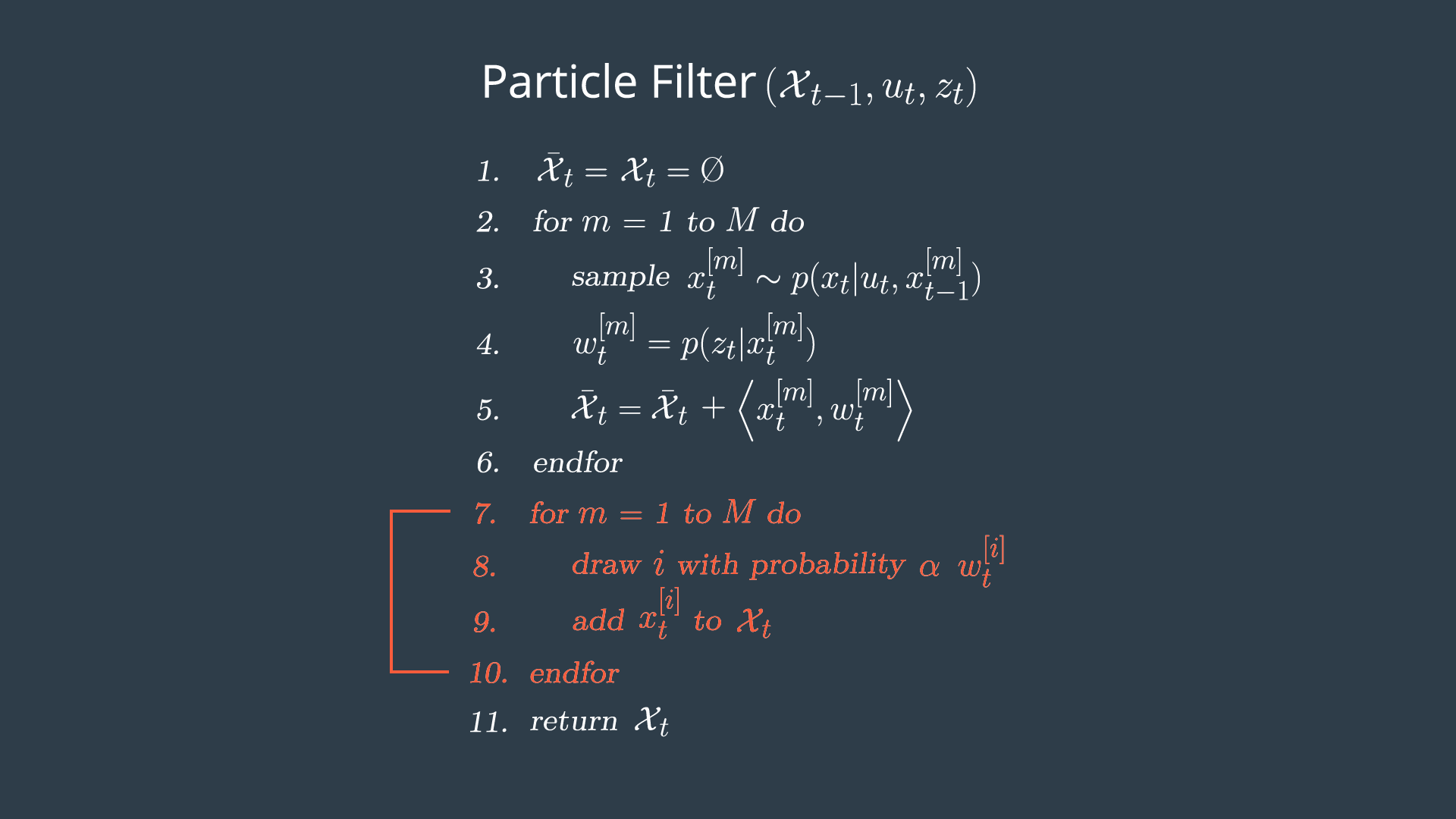
Prediction

During the prediction step we add the control input (yaw rate & velocity) for all particles



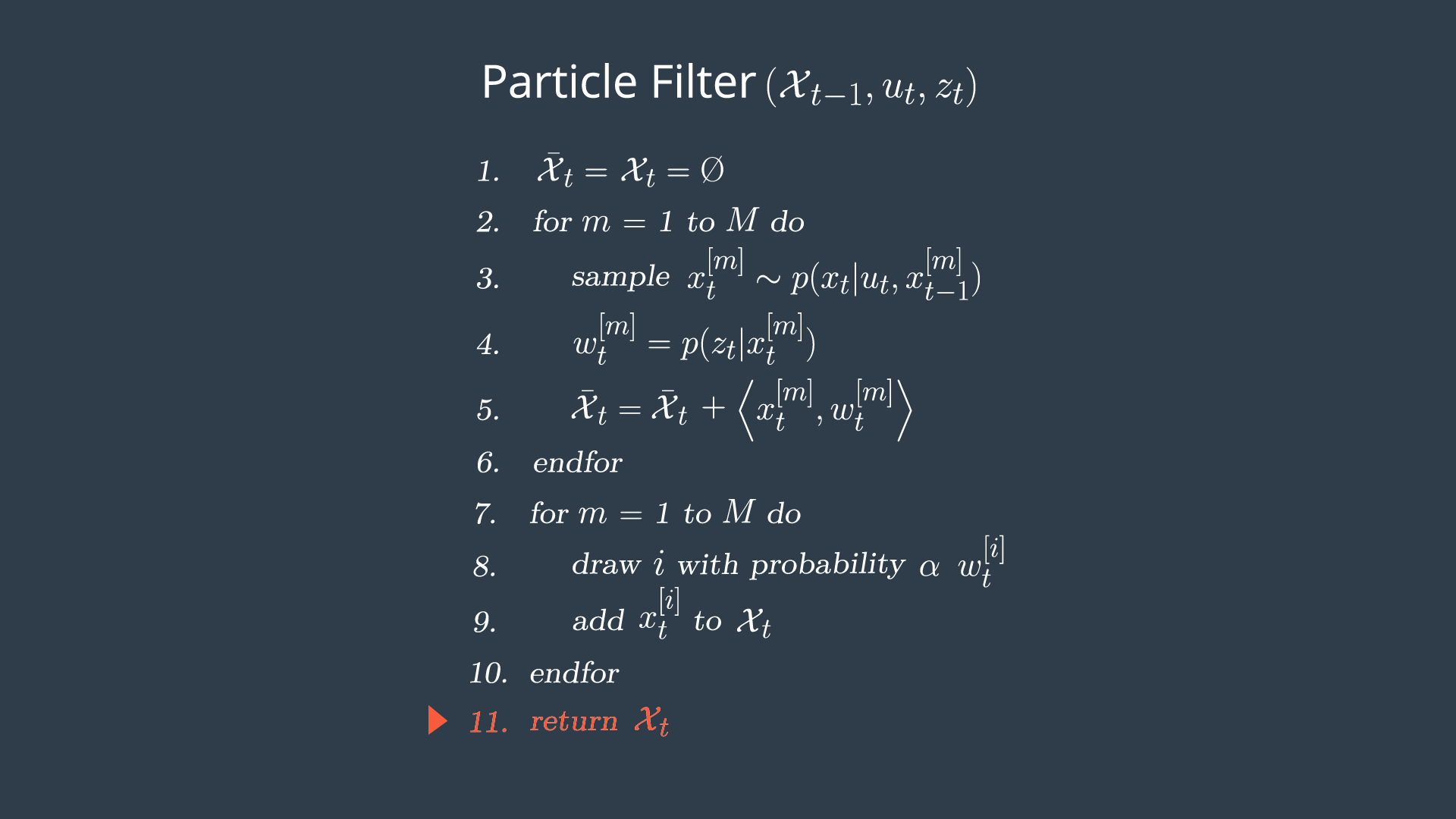
Update

During the update step, we update our particle weights using map landmark positions and feature measurements.



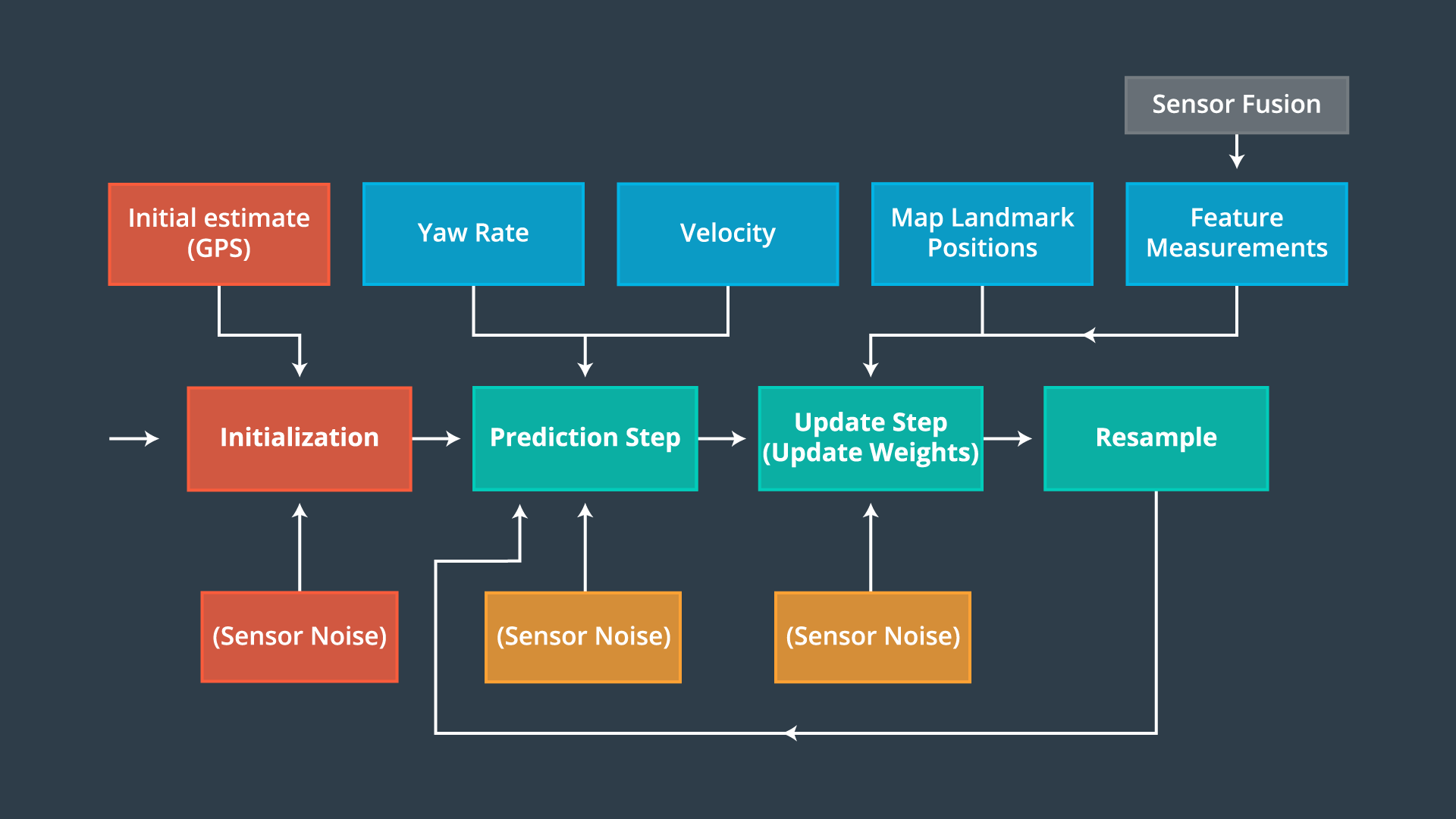
Resampling

During resampling we will resample M times (M is range of 0 to length\_of\_particleArray) drawing a particle i (i is the particle index) proportional to its weight . Sebastian covered one implementation of this in his [discussion and implementation of a resampling wheel](https://classroom.udacity.com/nanodegrees/nd013/parts/40f38239-66b6-46ec-ae68-03afd8a601c8/modules/2c318113-724b-4f9f-860c-cb334e6e4ad7/lessons/4d7950f7-f519-4dc6-9142-052a1261f5bf/concepts/487480820923" \t "_blank).



Return New Particle Set

The new set of particles represents the Bayes filter posterior probability. We now have a refined estimate of the vehicles position based on input evidence.



The most practical way to initialize our particles and generate real time output, is to make an initial estimate using GPS input. As with all sensor based operations, this step is impacted by noise.

## Project Implementation

* Particles shall be implemented by sampling a Gaussian distribution, taking into account Gaussian sensor noise around the initial GPS position and heading estimates.
* Use the [C++ standard library normal distribution](http://en.cppreference.com/w/cpp/numeric/random/normal_distribution" \t "_blank) and [C++ standard library random engine](http://www.cplusplus.com/reference/random/default_random_engine" \t "_blank) functions to sample positions around GPS measurements.

### Coding Instructions

I have provided you with a function that takes a GPS position and initial heading as input. I want you to print out to the terminal 3 samples from a normal distribution with mean equal to the GPS position and initial heading measurements and standard deviation of 2 m for the x and y position and 0.05 radians for the heading of the car.

Fill out the "TODO" sections in the code.

/\*\*

\* print\_samples.cpp

\*

\* Print out to the terminal 3 samples from a normal distribution with

\* mean equal to the GPS position and IMU heading measurements and

\* standard deviation of 2 m for the x and y position and 0.05 radians

\* for the heading of the car.

\*

\* Author: Tiffany Huang

\*/

#include <iostream>

#include <random> // Need this for sampling from distributions

using std::normal\_distribution;

/\*\*

\* Prints samples of x, y and theta from a normal distribution

\* @param gps\_x GPS provided x position

\* @param gps\_y GPS provided y position

\* @param theta GPS provided yaw

\*/

void printSamples(double gps\_x, double gps\_y, double theta);

int main() {

// Set GPS provided state of the car.

double gps\_x = 4983;

double gps\_y = 5029;

double theta = 1.201;

// Sample from the GPS provided position.

printSamples(gps\_x, gps\_y, theta);

return 0;

}

void printSamples(double gps\_x, double gps\_y, double theta) {

std::default\_random\_engine gen;

double std\_x, std\_y, std\_theta; // Standard deviations for x, y, and theta

// TODO: Set standard deviations for x, y, and theta

std\_x = 2;

std\_y = 2;

std\_theta = 0.05;

// This line creates a normal (Gaussian) distribution for x

normal\_distribution<double> dist\_x(gps\_x, std\_x);

// TODO: Create normal distributions for y and theta

normal\_distribution<double> dist\_y(gps\_y, std\_y);

normal\_distribution<double> dist\_theta(theta, std\_theta);

for (int i = 0; i < 3; ++i) {

double sample\_x, sample\_y, sample\_theta;

// TODO: Sample from these normal distributions like this:

// sample\_x = dist\_x(gen);

// where "gen" is the random engine initialized earlier.

sample\_x = dist\_x(gen);

sample\_y = dist\_y(gen);

sample\_theta = dist\_theta(gen);

// Print your samples to the terminal.

std::cout << "Sample " << i + 1 << " " << sample\_x << " " << sample\_y << " "

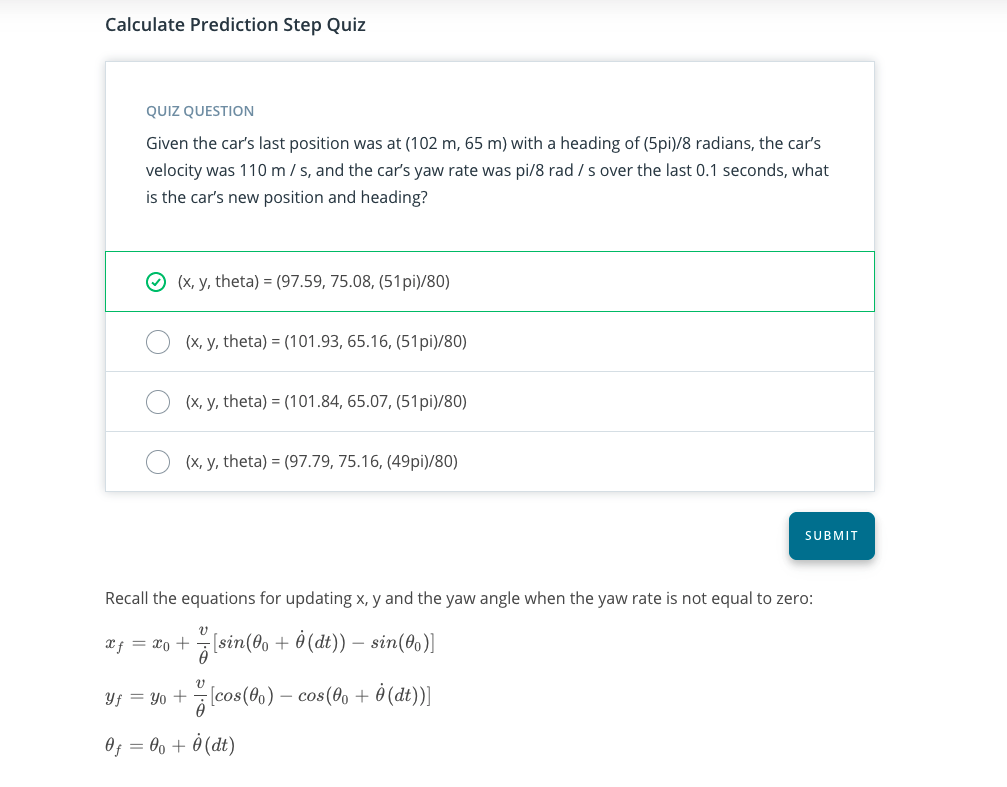
<< sample\_theta << std::endl;

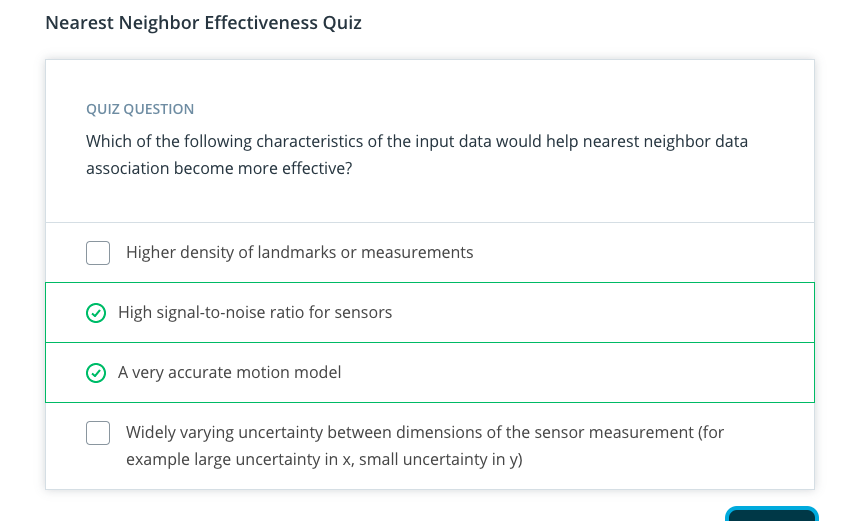
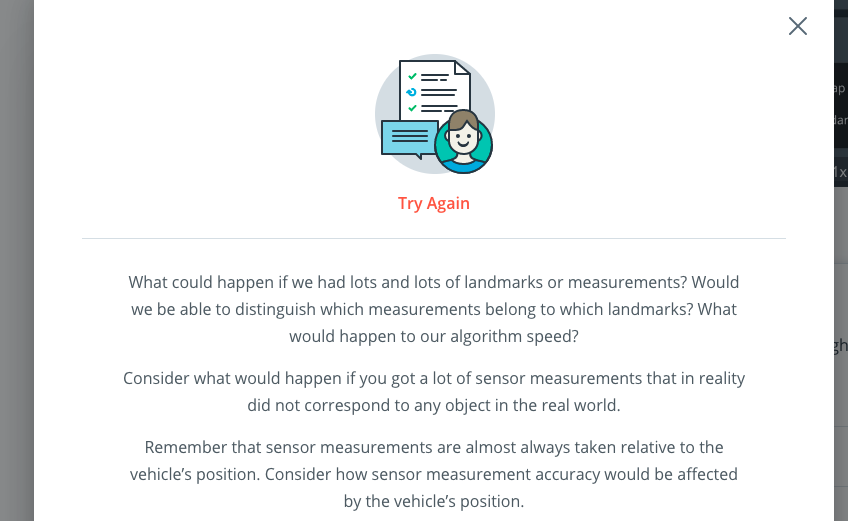
}

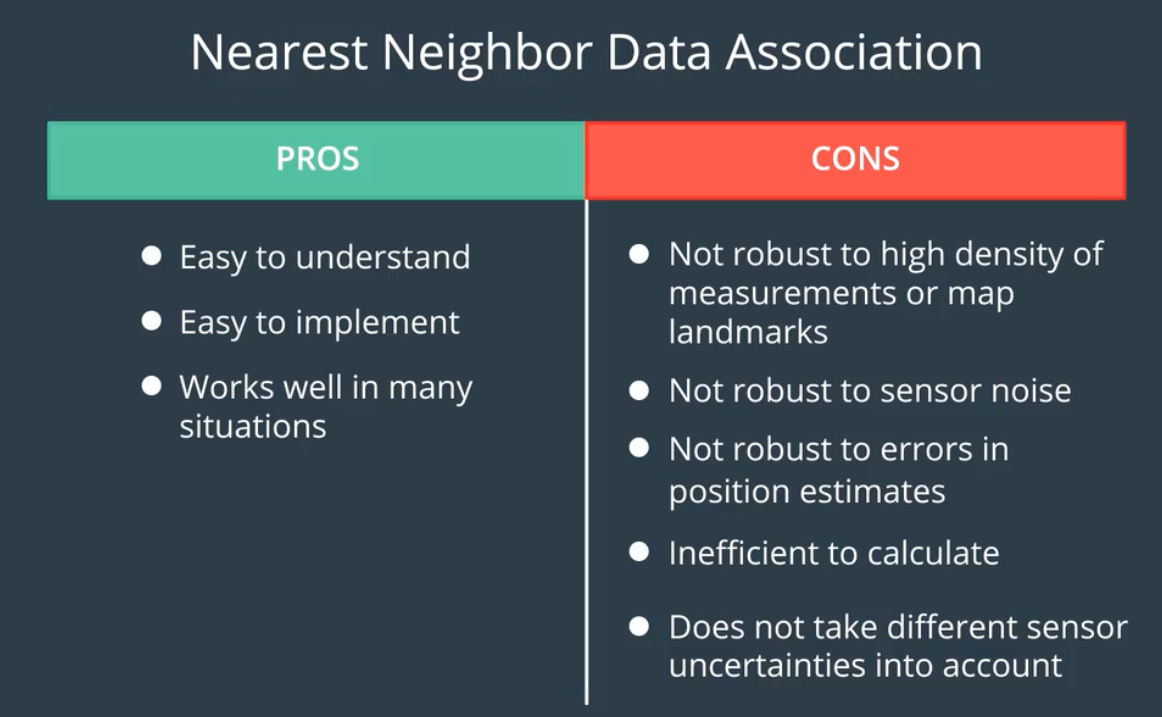
return;

}

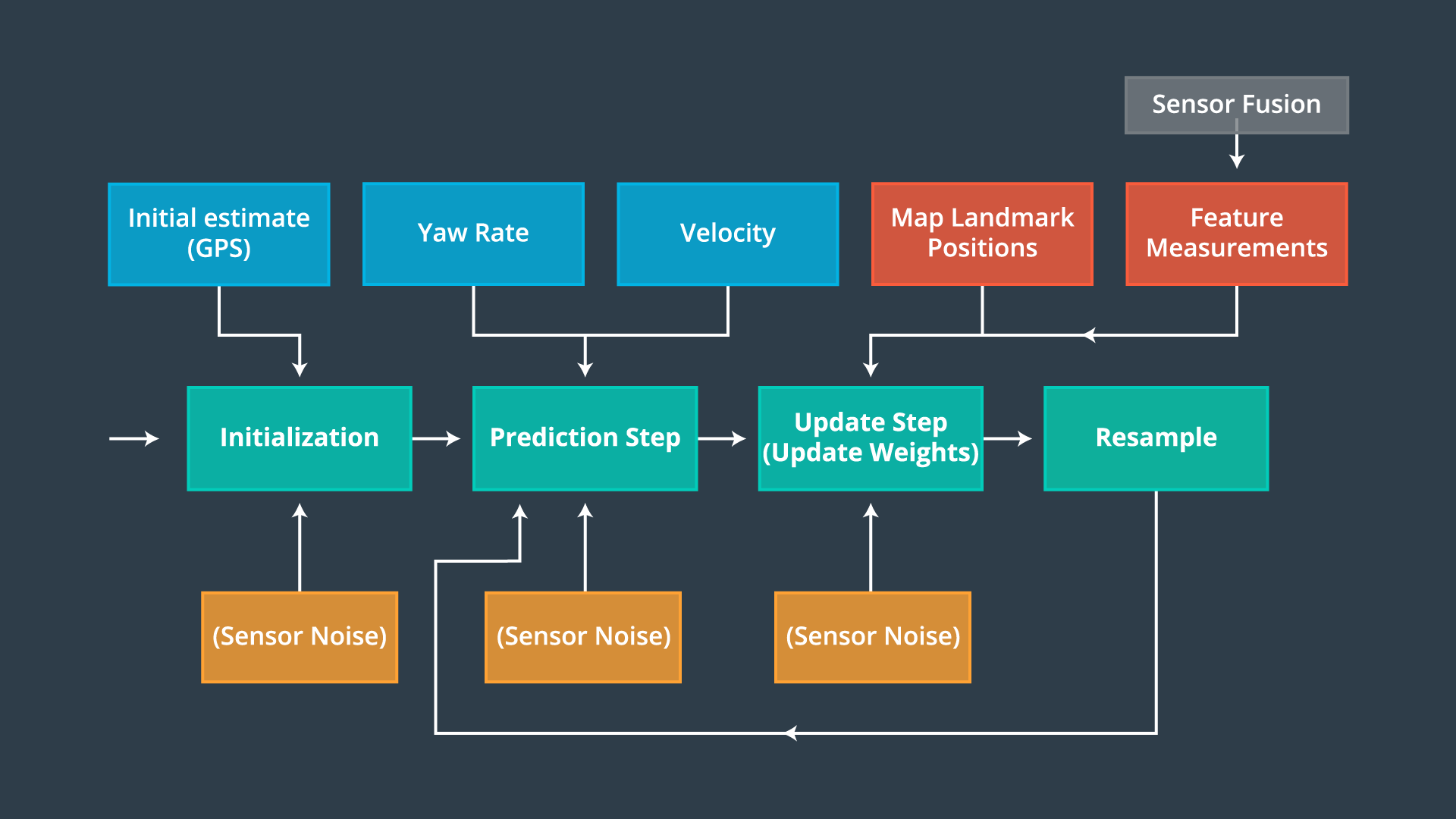
Now that we have initialized our particles it's time to predict the vehicle's position. Here we will use what we learned in the motion models lesson to predict where the vehicle will be at the next time step, by updating based on yaw rate and velocity, while accounting for Gaussian sensor noise.

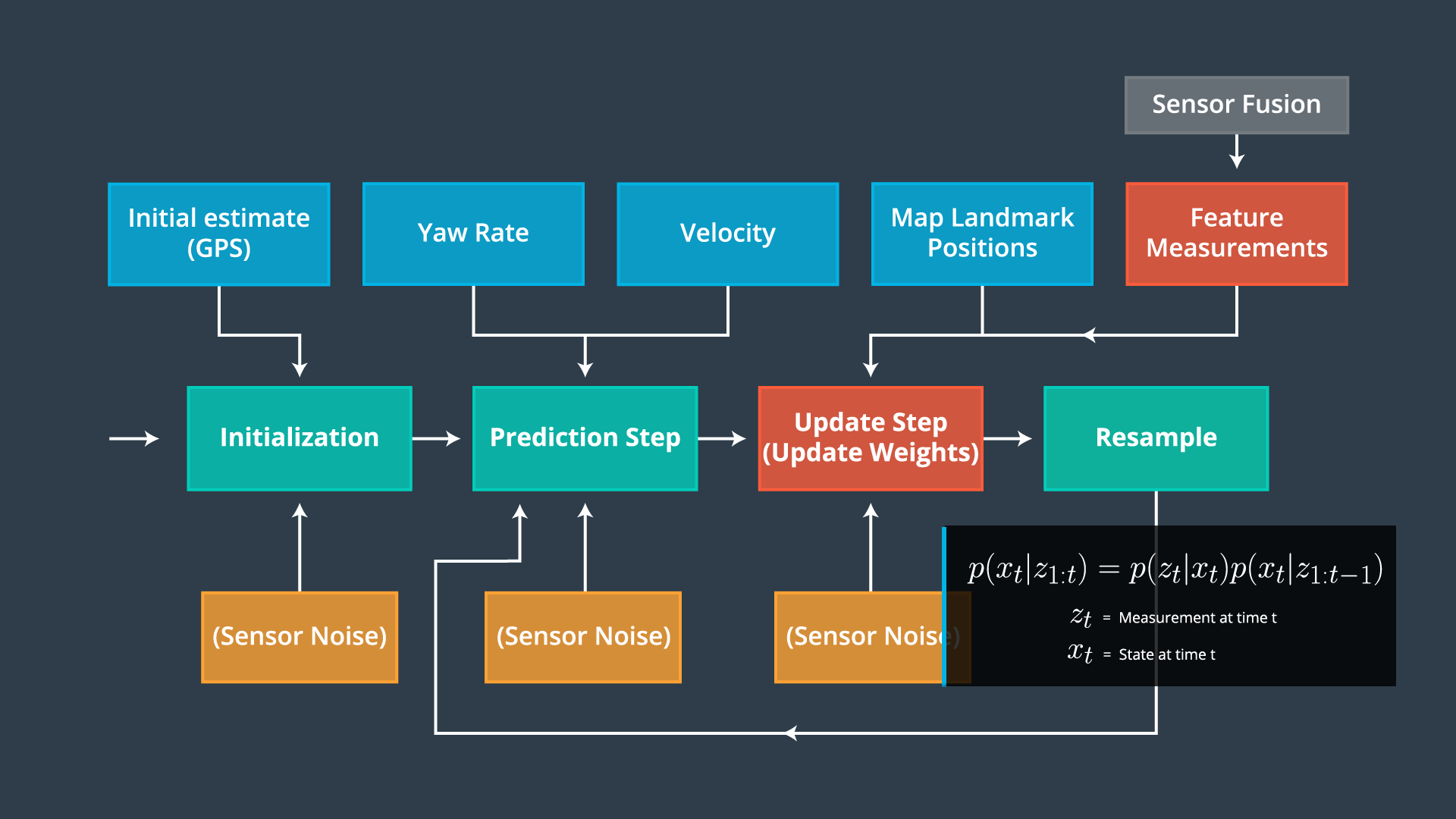






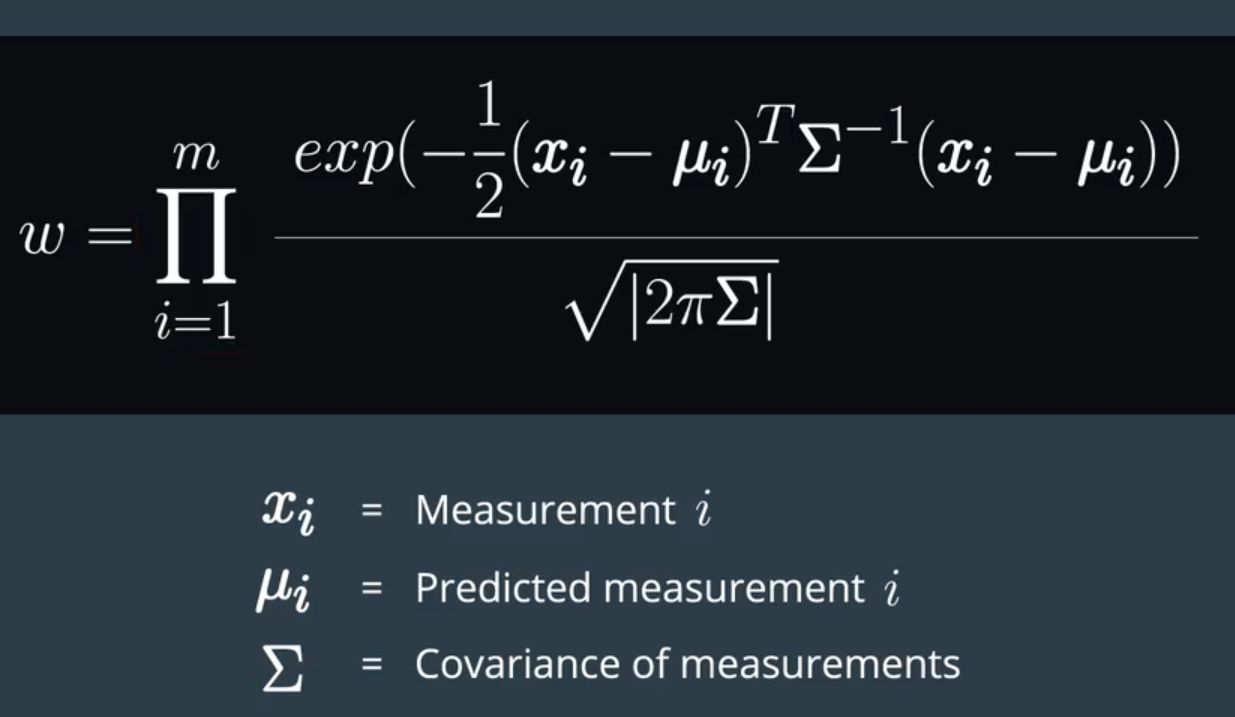
Note that the x and y errors are depicted from the point of view of the map (x is horizontal, y is vertical) rather than the point of view of the car where x is in the direction of the car’s heading,( i.e. It points to where the car is facing), and y is orthogonal (90 degrees) to the left of the x-axis (pointing out of the left side of the car).

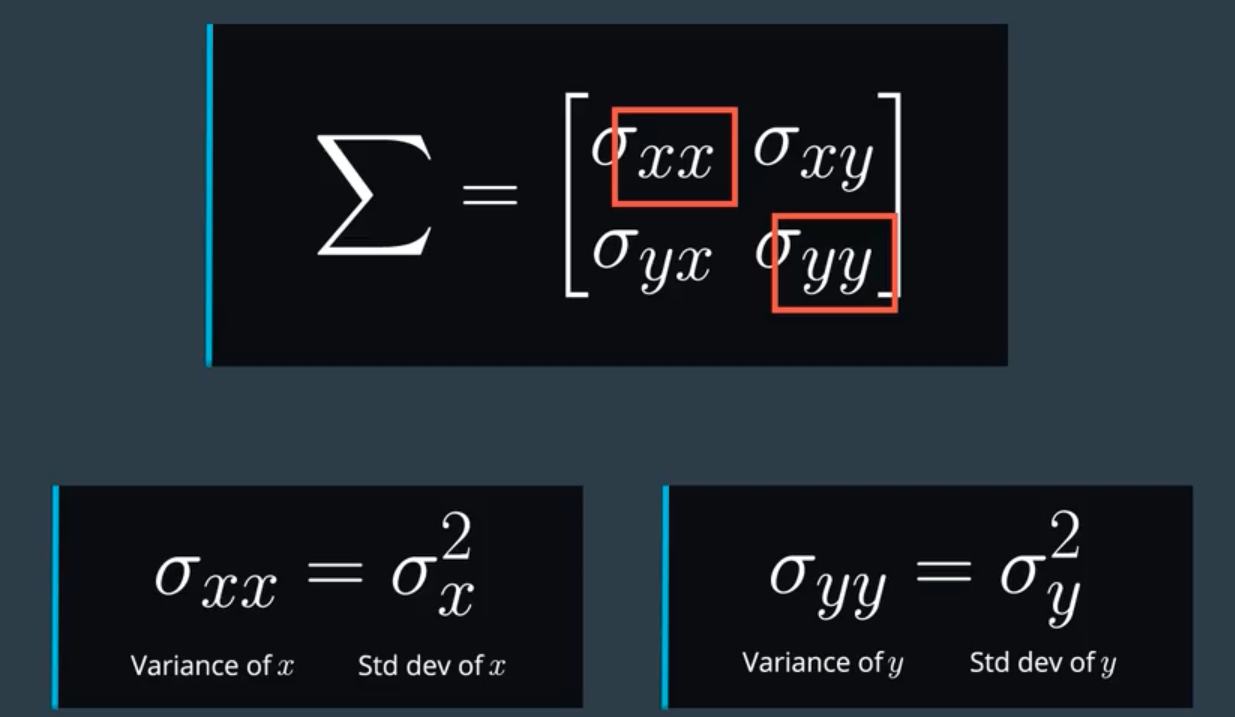


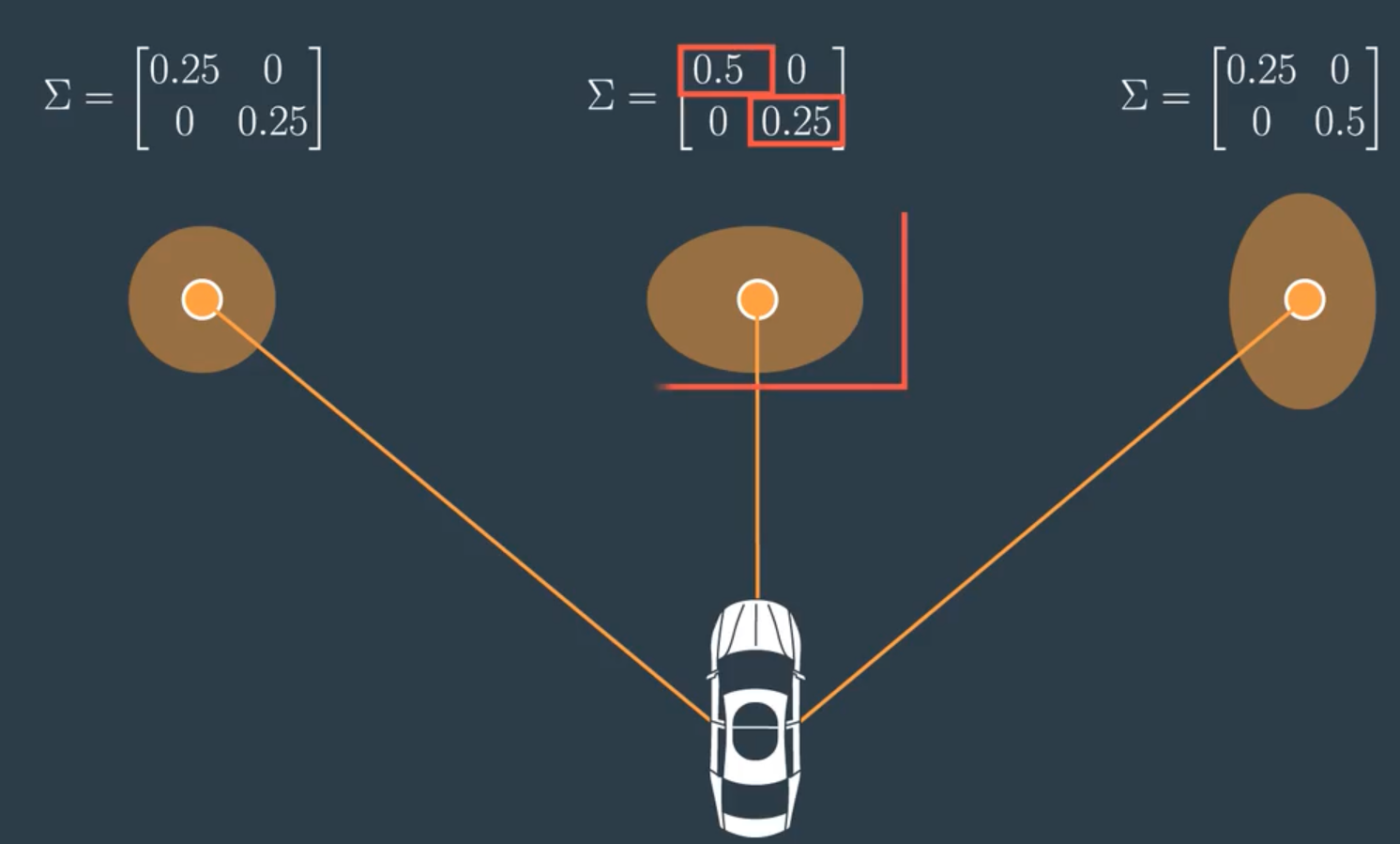


Now that we have incorporated velocity and yaw rate measurement inputs into our filter, we must update particle weights based on LIDAR and RADAR readings of landmarks. We will practice calculating particle weights, later in this lesson, with the Particle Weights Quiz.

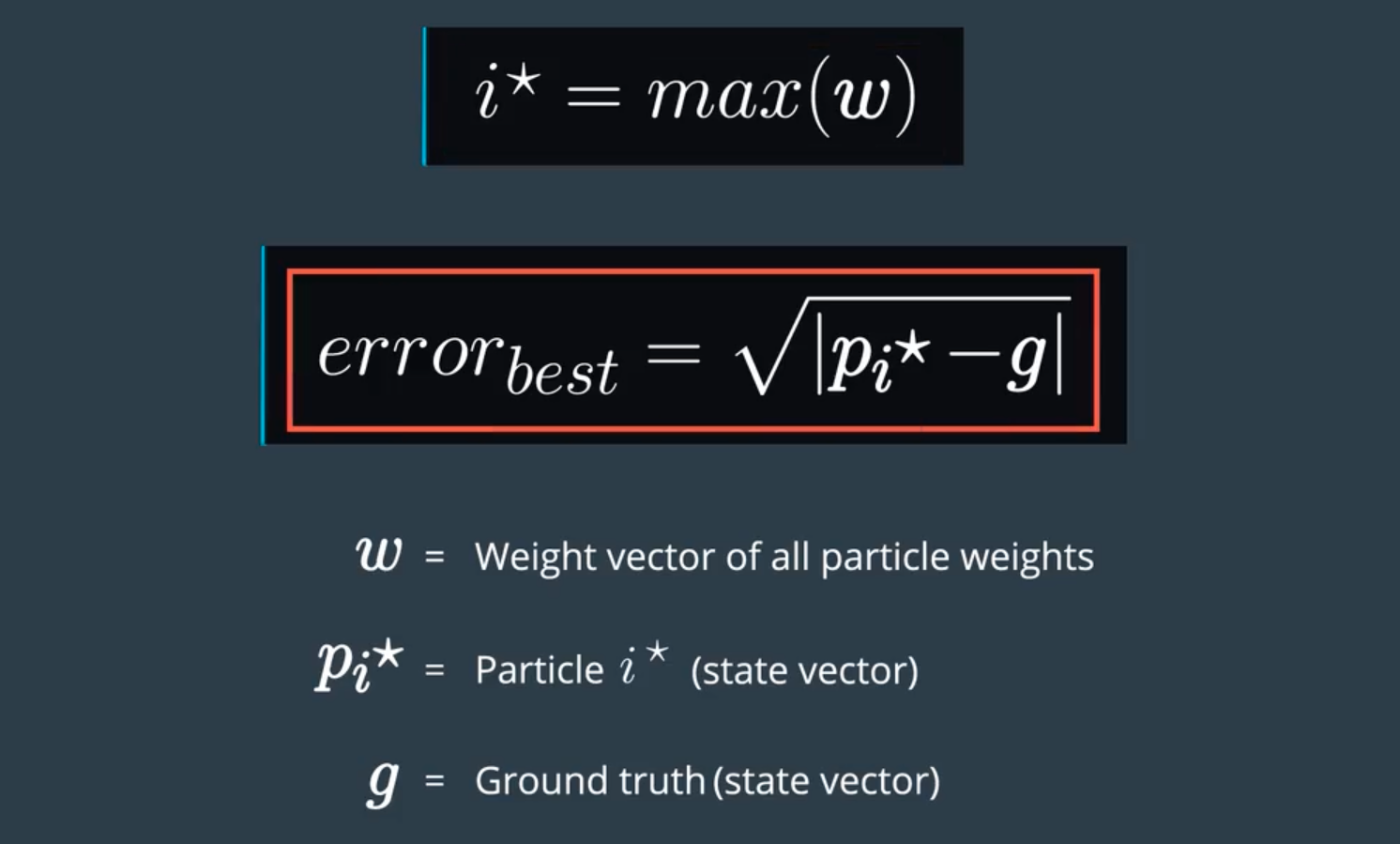
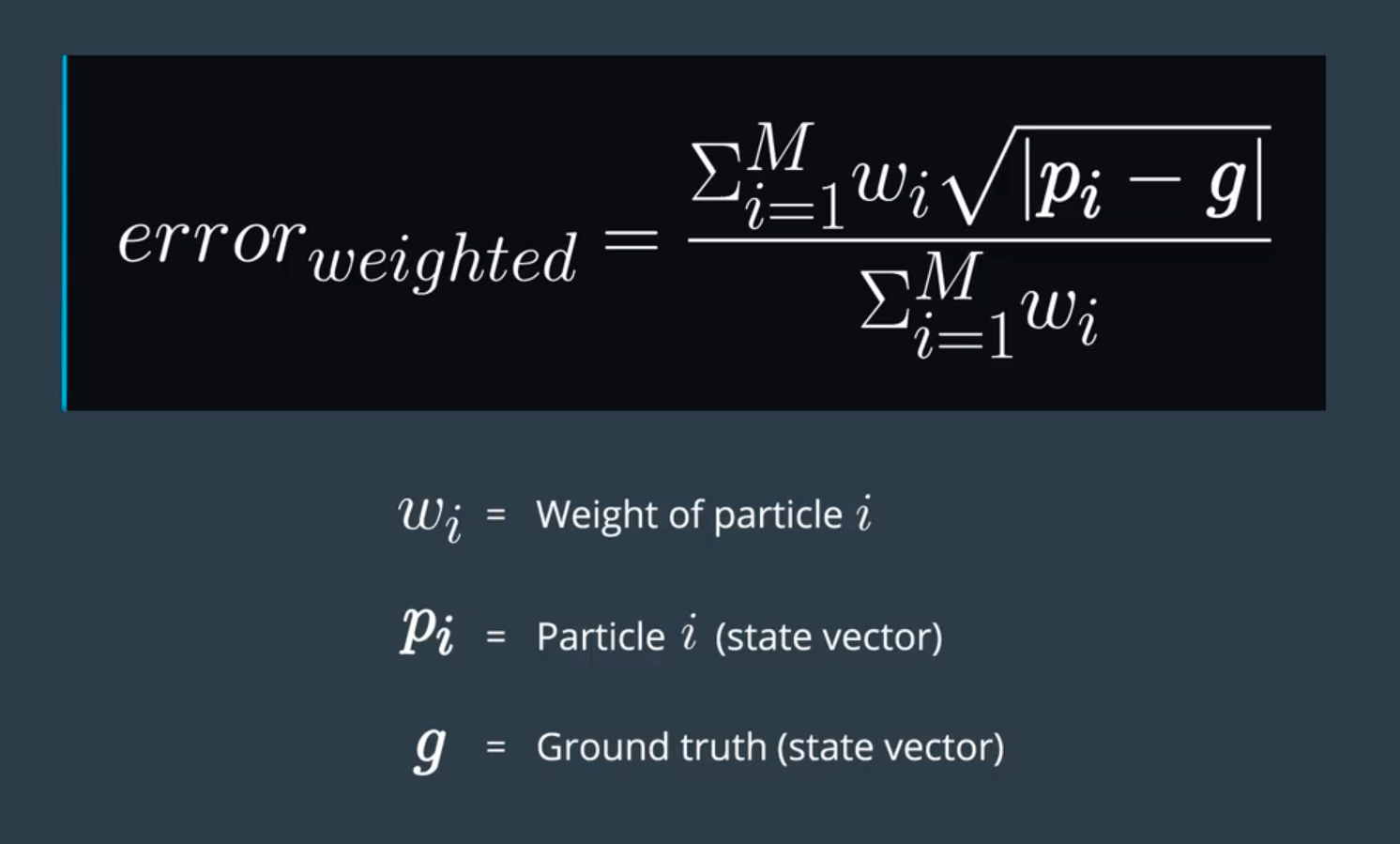
NEXT

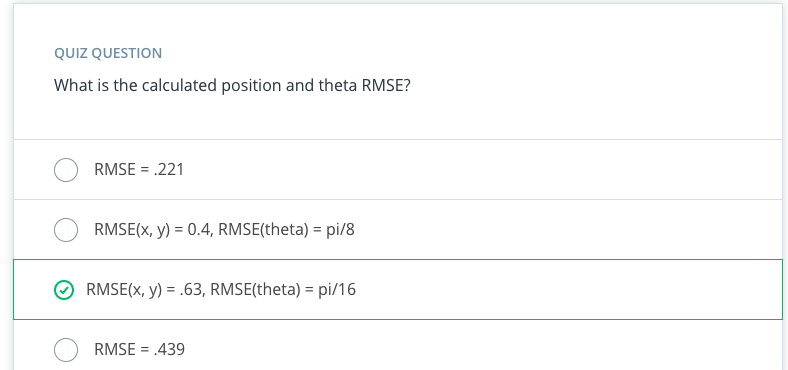
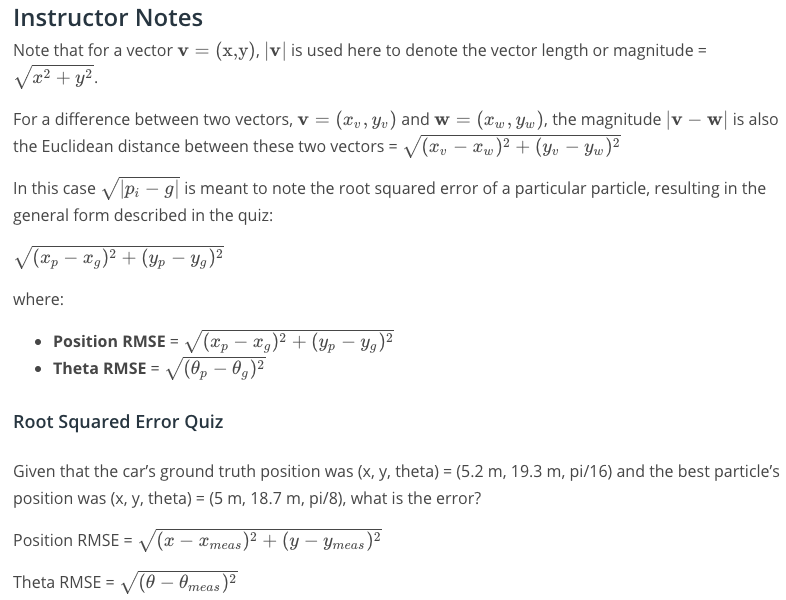






ERROR ESTIMATION

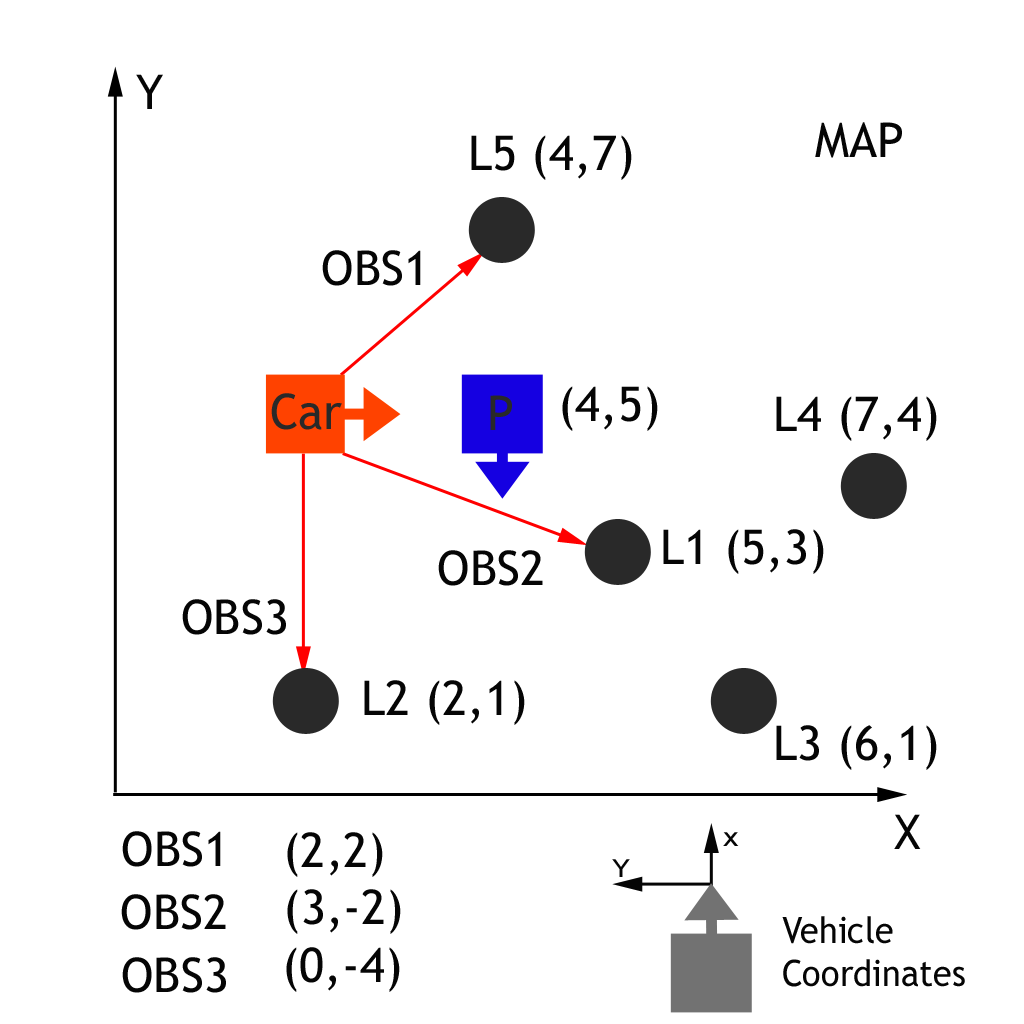




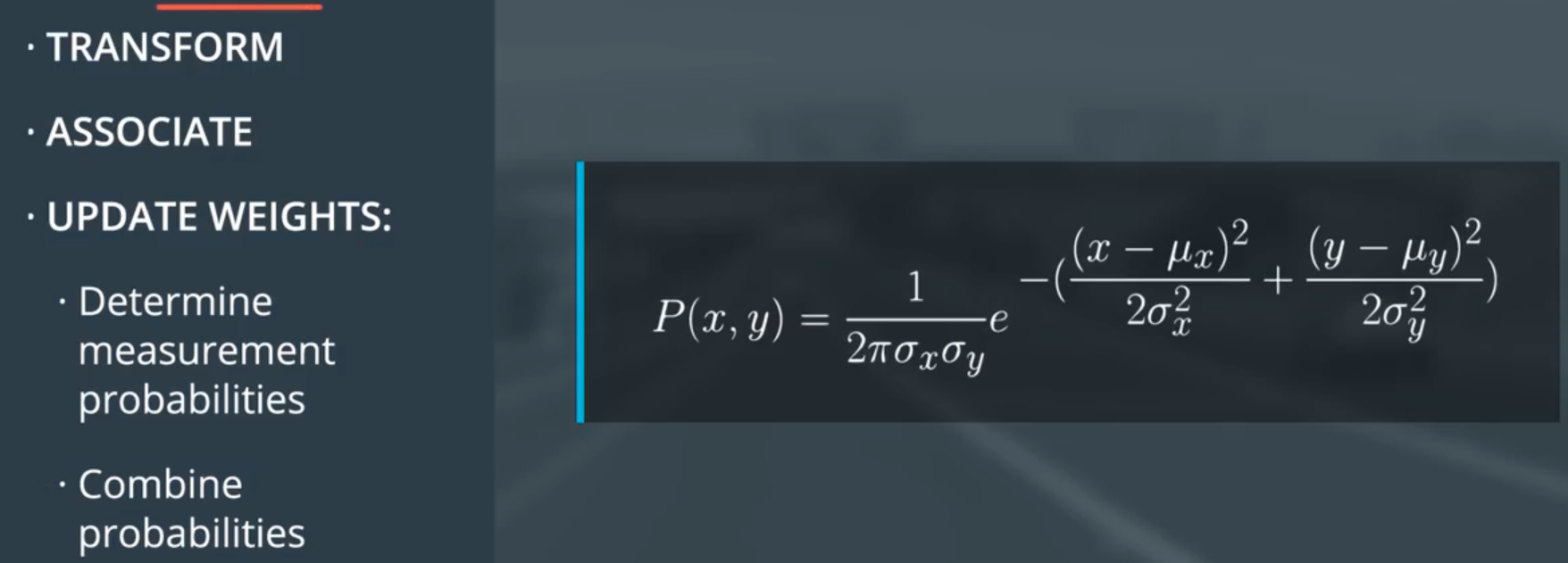
**Transformations and Associations**

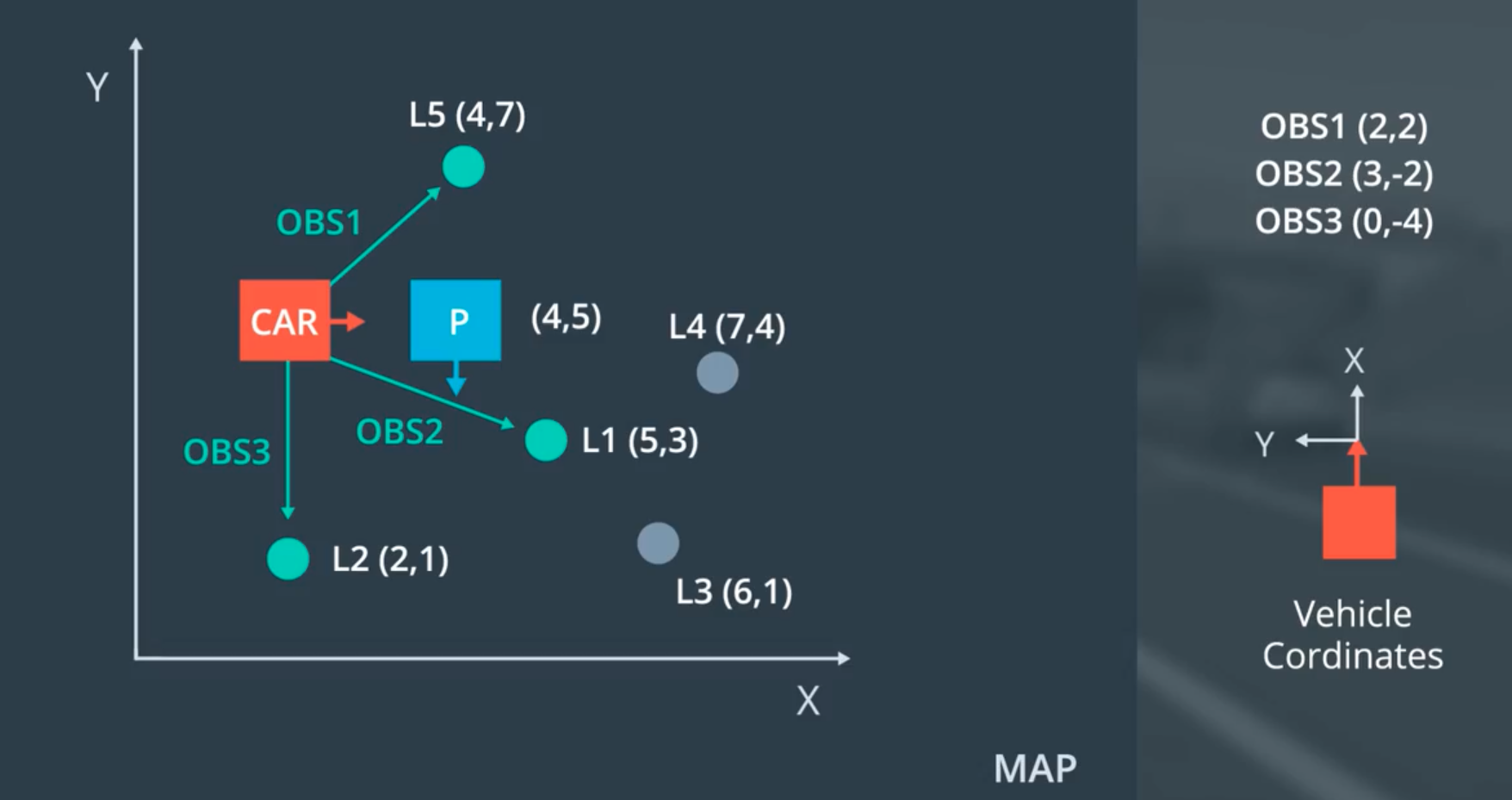
In the project you will need to correctly perform observation measurement transformations, along with identifying measurement landmark associations in order to correctly calculate each particle's weight. Remember, our ultimate goal is to find a weight parameter for each particle that represents how well that particle fits to being in the same location as the actual car.

In the quizzes that follow we will be given a single particle with its position and heading along with the car's observation measurements. We will first need to transform the car's measurements from its local car coordinate system to the map's coordinate system. Next, each measurement will need to be associated with a landmark identifier, for this part we will take the closest landmark to each transformed observation (refer to the **Data Association: Nearest Neighbor** page). Finally, we will use this information to calculate the weight value of the particle.

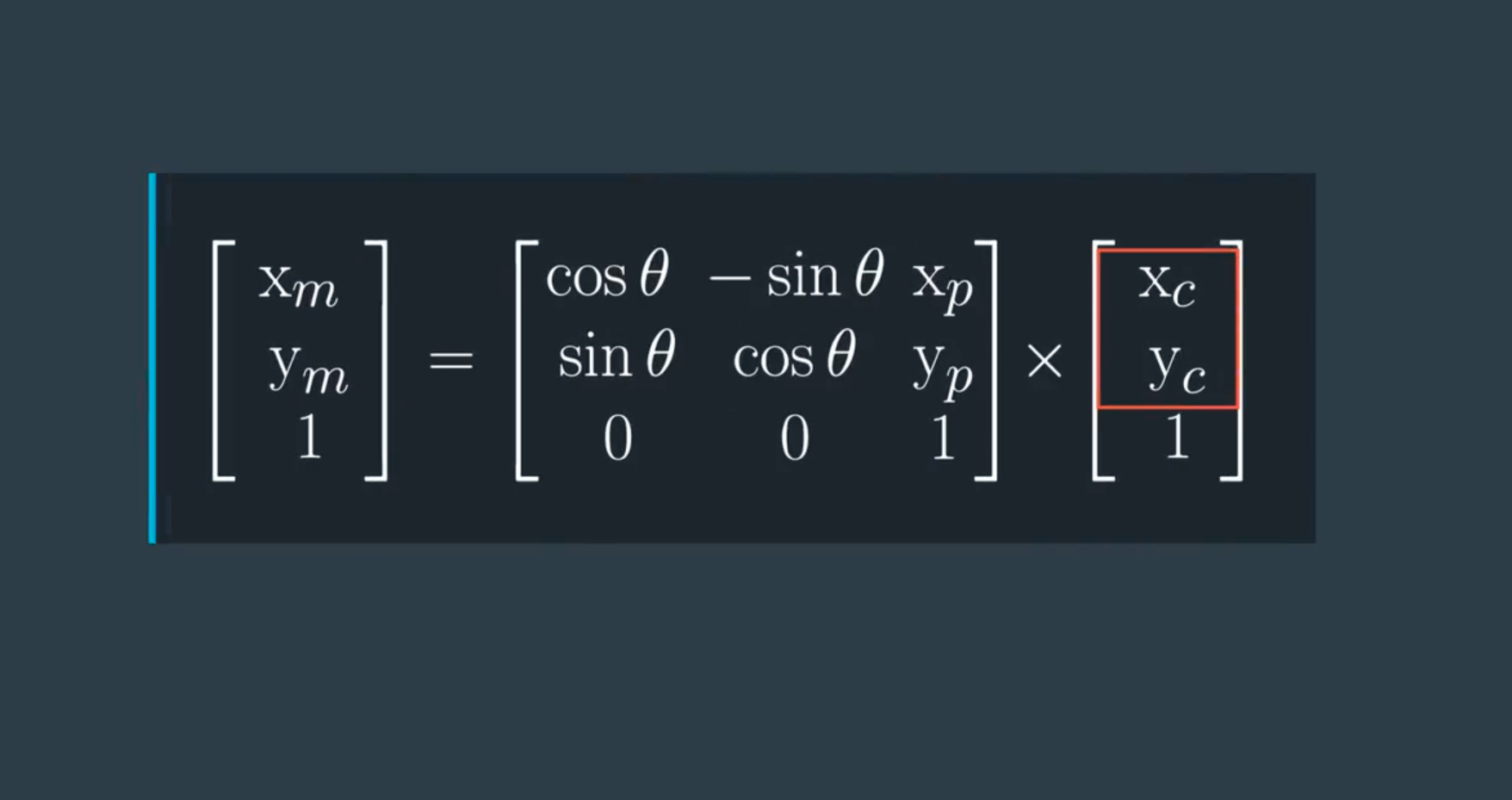


In the graph above we have a car (**ground truth position**) that observes three nearby landmarks, each one labeled OBS1, OBS2, OBS3. Each observation measurement has x, and y values in the car's coordinate system. We have a particle "P" (**estimated position of the car**) above with position (4,5) on the map with heading -90 degrees. The first task is to transform each observation marker from the vehicle's coordinates to the map's coordinates, with respect to our particle.





Car coordinate -> map coordinate( homogenouse transformation)좌표계 바꿀때

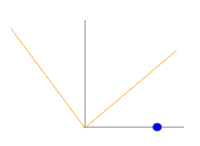


Here is another example that might help your intuition.

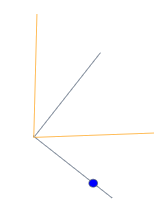
Referring to the figures below:

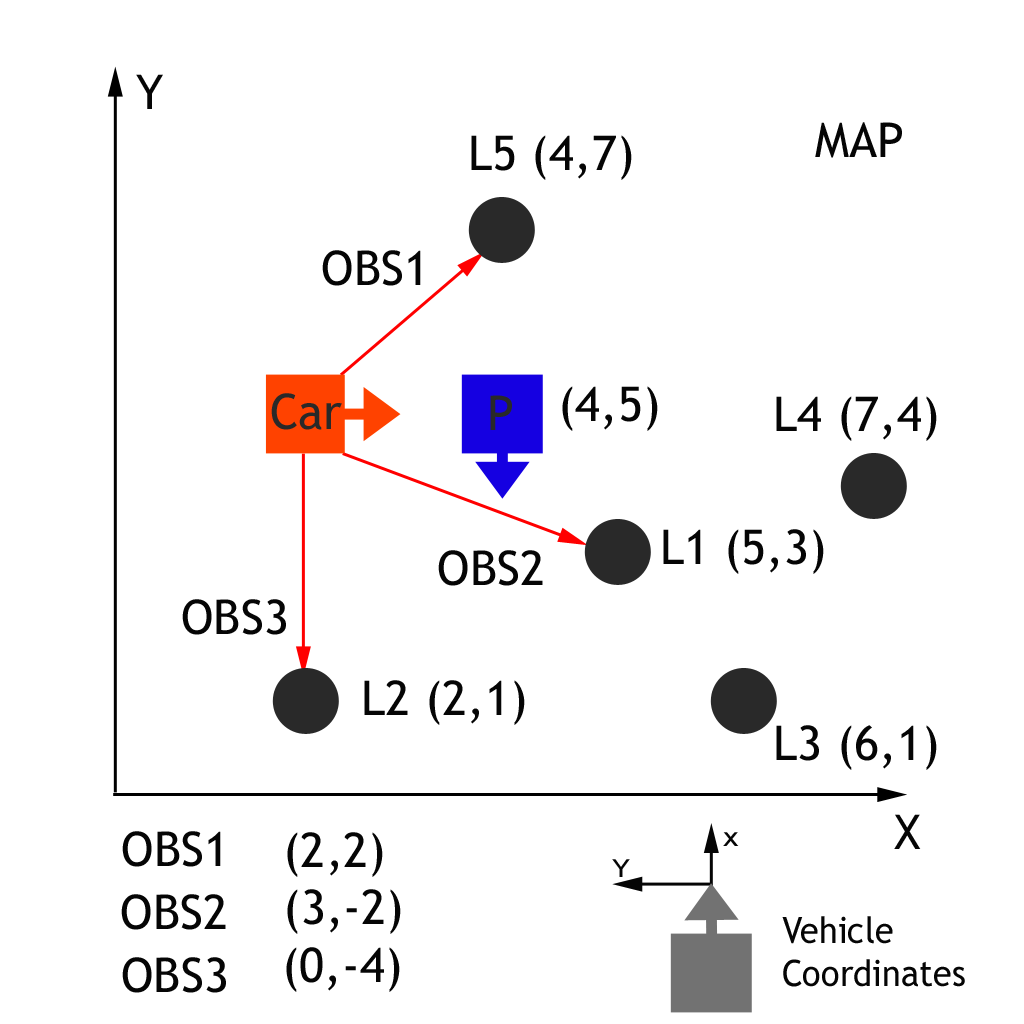
Suppose the map coordinate system (grey lines) and the vehicle coordinate system (orange lines) are offset, as depicted below. If we know the location of the observation in vehicle coordinates (grey lines), we would need to rotate the entire system, observation included, -45 degrees to find it in map coordinates (grey lines), Once this rotation is done, we can easily see the location of the observation in map coordinates.

**Particle (blue dot) in Map Frame (grey)**



**Particle (blue dot) in Vehicle Frame (orange)**





Map with Car Observations and Particle

In the graph above we have a car (**ground truth position**) that observes three nearby landmarks, each one labeled OBS1, OBS2, OBS3. Each observation measurement has x, and y values in the car's coordinate system. We have a particle "P" (**estimated position of the car**) above with position (4,5) on the map with heading -90 degrees. The first task is to transform each observation marker from the vehicle's coordinates to the map's coordinates, with respect to our particle.

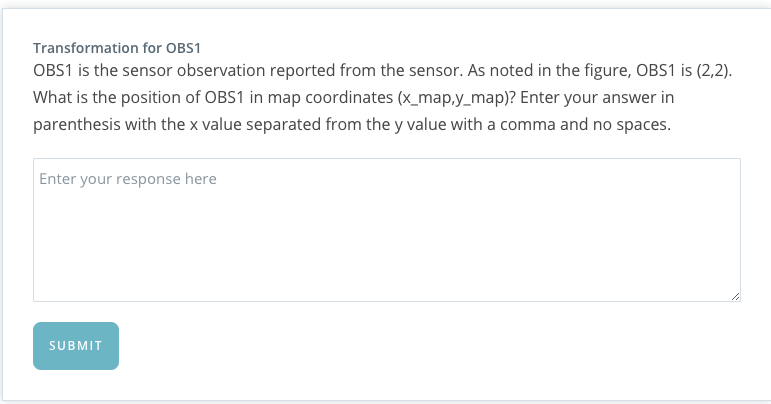
## Transformations

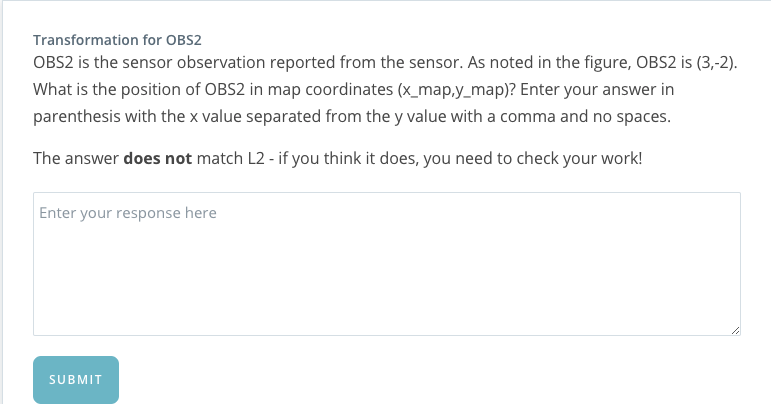
There is a generalized trigonometric function that given any particle position and heading along with any observation measurement (x\_obs,y\_obs) will output the transformed observation (x\_map,y\_map) for that particle. In other words, we need to map particle coordinates to map coordinates, by passing particle and observation coordinates through a function.

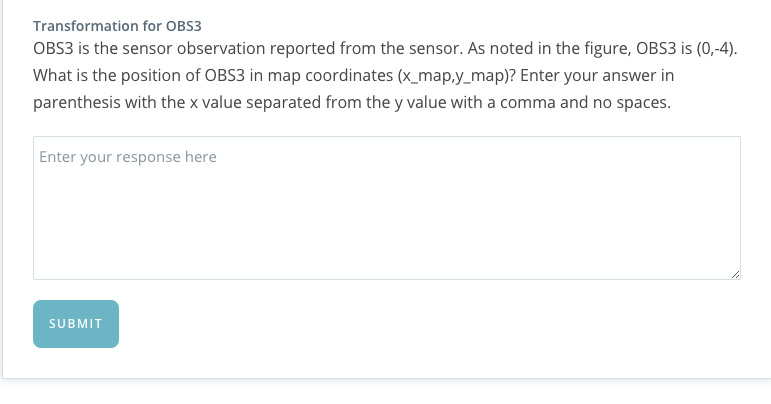
**Transformed Observation (x\_map,y\_map)** = func(x\_particle, y\_particle, heading\_particle, x\_obs, y\_obs)

You will need to derive and use this function to efficiently calculate particle observation transformations in the project.

* Some excellent resources discussing coordinate transformation and rotation matrices:
  + [Coordinate Transformations](http://farside.ph.utexas.edu/teaching/336k/Newtonhtml/node153.html)
  + [Coordinate Transformation Under Rotation](https://www.miniphysics.com/coordinate-transformation-under-rotation.html)

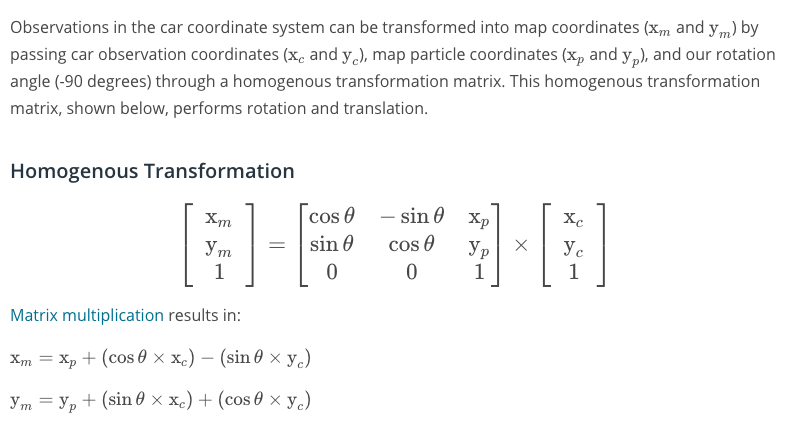






# Resources, Hints, and Tips

* Without implementation of localization methods the car does not know where it is within an acceptable level of precision. The car knows particle coordinates and observation coordinates. The objective is to use the particle coordinates and heading to transform the car's frame of reference to the map's frame of reference, associate the observations, then use the associated observations in the map domain to update the particle weight.
* Since we know the coordinates of the particle from the car's frame of reference we can use this information and a matrix rotation/translation to transform each observation from the car frame of reference to the map frame of reference. The particle is at (4,5) in the map coordinate system with a heading of -90 degrees. The figure indicates the heading by depicting the particle x-axis as pointing down (blue arrow). This is critical to understanding the matrix transformation we are about to perform.
* By convention we define the car coordinate system with x pointing up and y rotated from x by pi/2 (+90 degrees). This is another way of saying that y is perpendicular and to the left of x. When x is pointing down, we have the mirror of this, with y perpendicular to the right. To visualize this make an L with your left index finger and thumb with palm facing away from you, this is our map frame of reference. Point the thumb towards the ceiling, this is the car coordinate convention, now point your thumb down, this is the orientation of the particle at (4,5).
* Now consider the map frame of reference (make an L with your left index finger and thumb as above), this is a typical presentation of Cartesian coordinates, with x pointing right and y perpendicular to the left, pointing up. If we rotate our thumb down we have the particle orientation. To get back to the map orientation we must rotate counterclockwise by 90 degrees (thumb from pointing down back to pointing right). Try this a few times with your left hand. Notice that particle to map is a counterclockwise rotation (+90 degrees) nd map to particle is a clockwise rotation (-90 degrees).
* The most straight forward way of rotating and translating coordinates is through homogenous transformation matrix (see below) using the angle of rotation required to get to the particle’s frame from the map’s point of view, -90 degrees. This way we can use theta directly. The alternative, which we will not cover here is to is to use -theta and a transformation matrix from the particle frame to the map frame.
* [This video](https://youtu.be/h11ljFJeaLo) is a great resource for developing a deeper understanding of how to solve this transformation problem - it covers the rotation transformation, and from there you just need to perform a translation.



**Quiz Solutions**

**Observation 1 Solution**

**#include <cmath>**

**#include <iostream>**

**int** **main**() {

*// define coordinates and theta*

**double** x\_part, y\_part, x\_obs, y\_obs, theta;

x\_part = 4;

y\_part = 5;

x\_obs = 2;

y\_obs = 2;

theta = -M\_PI/2; *// -90 degrees*

*// transform to map x coordinate*

**double** x\_map;

x\_map = x\_part + (cos(theta) \* x\_obs) - (sin(theta) \* y\_obs);

*// transform to map y coordinate*

**double** y\_map;

y\_map = y\_part + (sin(theta) \* x\_obs) + (cos(theta) \* y\_obs);

*// (6,3)*

std::cout << **int**(round(x\_map)) << ", " << **int**(round((y\_map)) << std::endl;

**return** 0;

}

**Observation 2 Solution**

**#include <cmath>**

**#include <iostream>**

**int** **main**() {

*// define coordinates and theta*

**double** x\_part, y\_part, x\_obs, y\_obs, theta;

x\_part = 4;

y\_part = 5;

x\_obs = 3;

y\_obs = -2;

theta = -M\_PI/2; *// -90 degrees*

*// transform to map x coordinate*

**double** x\_map;

x\_map = x\_part + (cos(theta) \* x\_obs) - (sin(theta) \* y\_obs);

*// transform to map y coordinate*

**double** y\_map;

y\_map = y\_part + (sin(theta) \* x\_obs) + (cos(theta) \* y\_obs);

*// (2,2)*

std::cout << **int**(round(x\_map)) << ", " << **int**(round(y\_map)) << std::endl;

**return** 0;

}

**Observation 3 Solution**

**#include <cmath>**

**#include <iostream>**

**int** **main**() {

*// define coordinates and theta*

**double** x\_part, y\_part, x\_obs, y\_obs, theta;

x\_part = 4;

y\_part = 5;

x\_obs = 0;

y\_obs = -4;

theta = -M\_PI/2; *// -90 degrees*

*// transform to map x coordinate*

**double** x\_map;

x\_map = x\_part + (cos(theta) \* x\_obs) - (sin(theta) \* y\_obs);

*// transform to map y coordinate*

**double** y\_map;

y\_map = y\_part + (sin(theta) \* x\_obs) + (cos(theta) \* y\_obs);

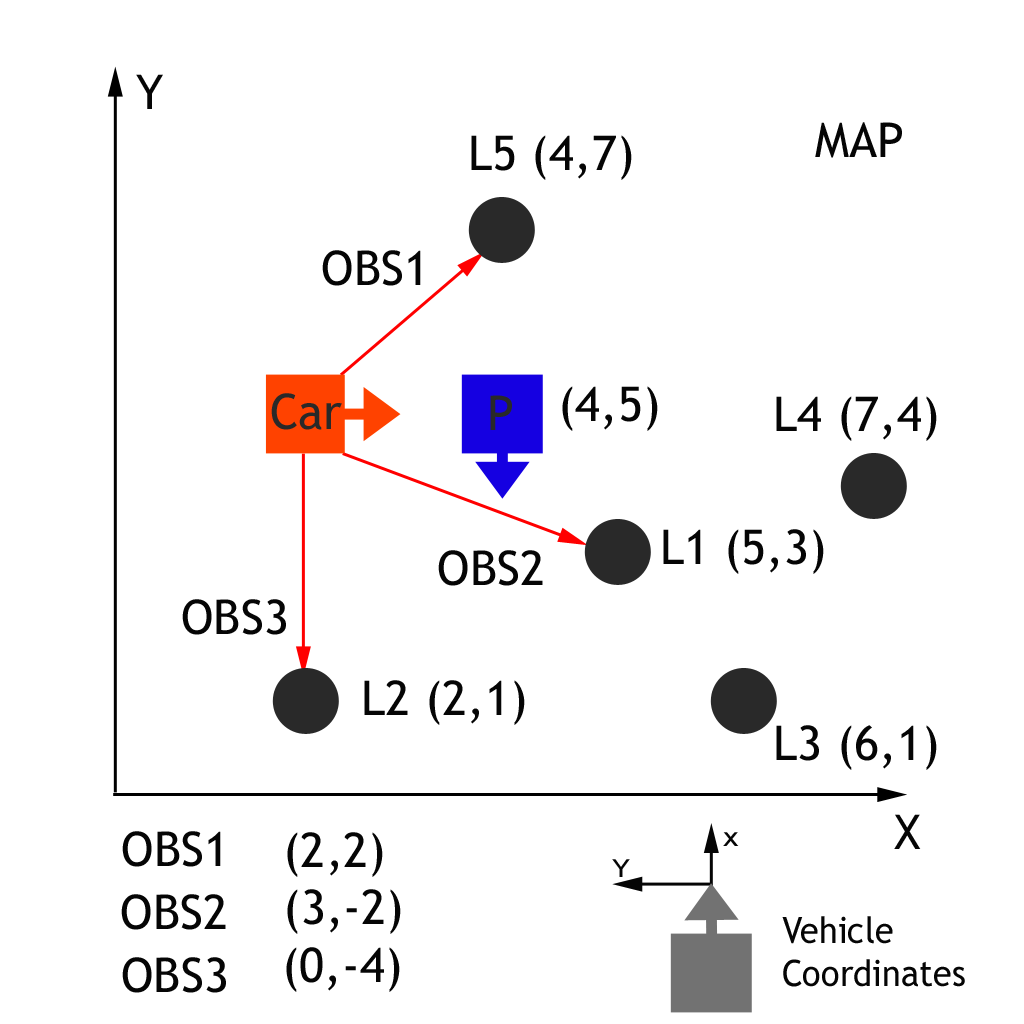
*// (0,5)*

std::cout << **int**(round(x\_map)) << ", " << **int**(round(y\_map)) << std::endl;

**return** 0;

}

**QUIZ ASSOCIATION**



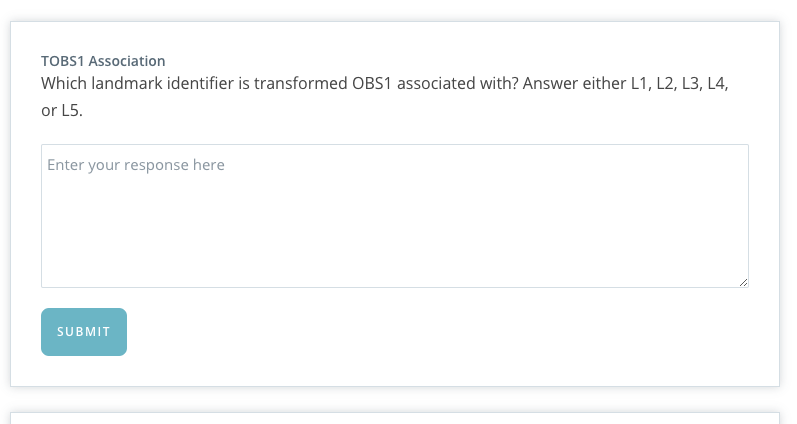
Map with Car Observations and Particle

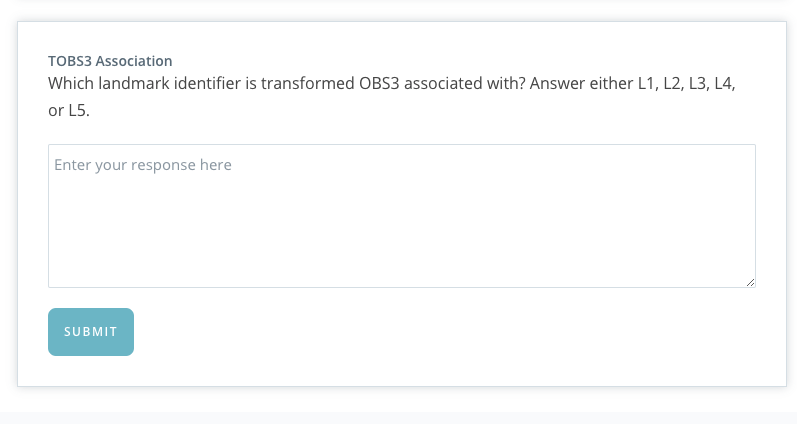
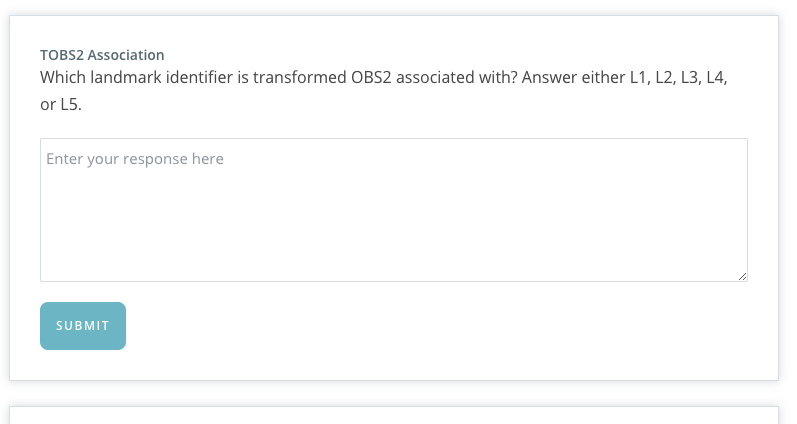
**Associations**

Now that observations have been transformed into the map's coordinate space, the next step is to associate each transformed observation with a land mark identifier. In the map exercise above we have 5 total landmarks each identified as L1, L2, L3, L4, L5, and each with a known map location. We need to associate each transformed observation TOBS1, TOBS2, TOBS3 with one of these 5 identifiers. To do this we must associate the closest landmark to each transformed observation.

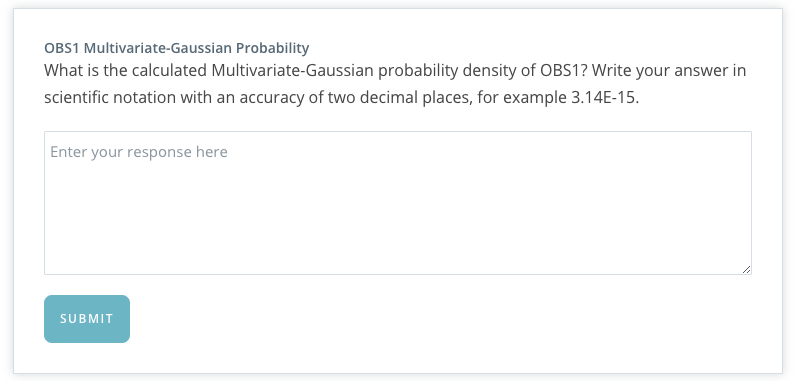
As a reminder:

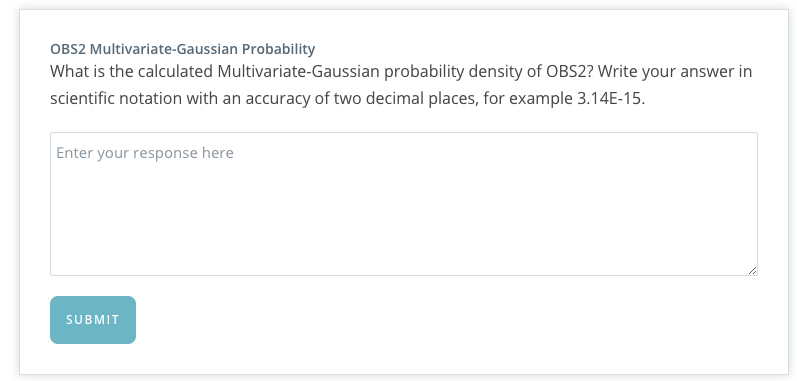
TOBS1 = (6,3), TOBS2 = (2,2) and TOBS3 = (0,5). [those are observation in map coordinate]

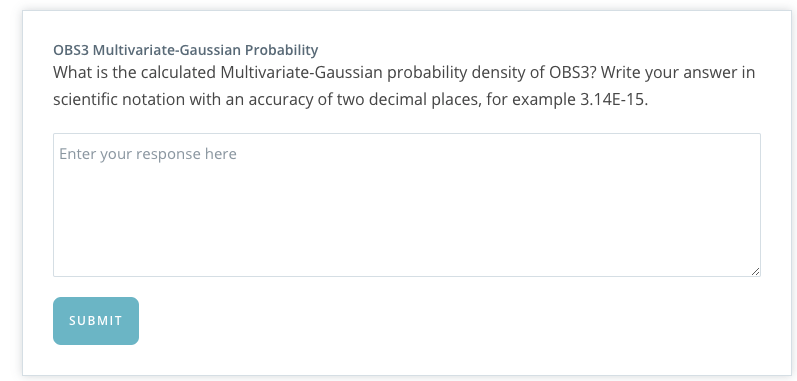


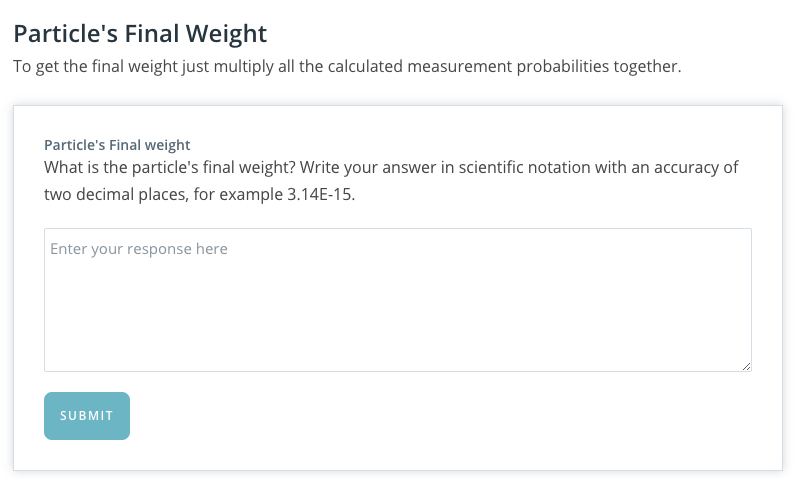
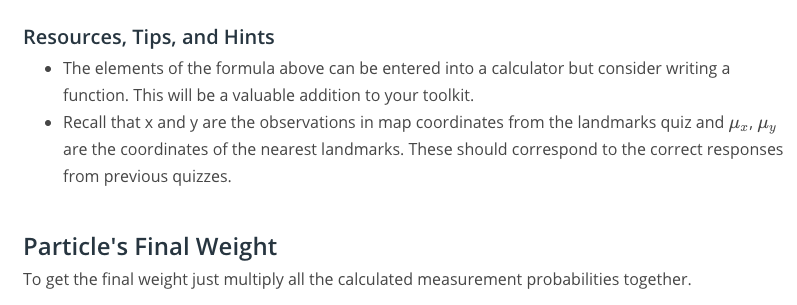


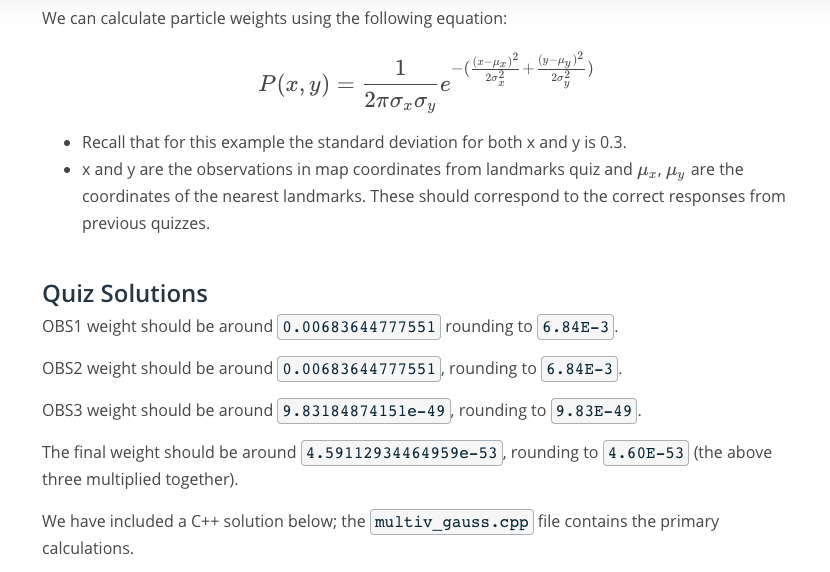












Main.cpp

-

#include <iostream>

#include "multiv\_gauss.h"

int main() {

// define inputs

double sig\_x, sig\_y, x\_obs, y\_obs, mu\_x, mu\_y;

// define outputs for observations

double weight1, weight2, weight3;

// final weight

double final\_weight;

// OBS1 values

// standard Deviation

sig\_x = 0.3;

sig\_y = 0.3;

// observation in map coordinate

x\_obs = 6;

y\_obs = 3;

// nearest particle

mu\_x = 5;

mu\_y = 3;

// Calculate OBS1 weight(Particle Weight)

weight1 = multiv\_prob(sig\_x, sig\_y, x\_obs, y\_obs, mu\_x, mu\_y);

// should be around 0.00683644777551 rounding to 6.84E-3

std::cout << "Weight1: " << weight1 << std::endl;

// OBS2 values

sig\_x = 0.3;

sig\_y = 0.3;

x\_obs = 2;

y\_obs = 2;

mu\_x = 2;

mu\_y = 1;

// Calculate OBS2 weight

weight2 = multiv\_prob(sig\_x, sig\_y, x\_obs, y\_obs, mu\_x, mu\_y);

// should be around 0.00683644777551 rounding to 6.84E-3

std::cout << "Weight2: " << weight2 << std::endl;

// OBS3 values

sig\_x = 0.3;

sig\_y = 0.3;

x\_obs = 0;

y\_obs = 5;

mu\_x = 2;

mu\_y = 1;

// Calculate OBS3 weight

weight3 = multiv\_prob(sig\_x, sig\_y, x\_obs, y\_obs, mu\_x, mu\_y);

// should be around 9.83184874151e-49 rounding to 9.83E-49

std::cout << "Weight3: " << weight3 << std::endl;

// Output final weight

final\_weight = weight1 \* weight2 \* weight3;

// 4.60E-53

std::cout << "Final weight: " << final\_weight << std::endl;

return 0;

}

-

Multiv\_gauss,cpp

-

#include "multiv\_gauss.h"

#include <cmath>

double multiv\_prob(double sig\_x, double sig\_y, double x\_obs, double y\_obs,

double mu\_x, double mu\_y) {

// calculate normalization term

double gauss\_norm;

gauss\_norm = 1 / (2 \* M\_PI \* sig\_x \* sig\_y);

// calculate exponent

double exponent;

exponent = (pow(x\_obs - mu\_x, 2) / (2 \* pow(sig\_x, 2)))

+ (pow(y\_obs - mu\_y, 2) / (2 \* pow(sig\_y, 2)));

// calculate weight using normalization terms and exponent

double weight;

weight = gauss\_norm \* exp(-exponent);

return weight;

}

-

Multiv\_gauss.h

-

#ifndef MULTIV\_GAUSS\_H

#define MULTIV\_GAUSS\_H

double multiv\_prob(double sig\_x, double sig\_y, double x\_obs, double y\_obs,

double mu\_x, double mu\_y);

#endif // MULTIV\_GAUSS\_H

-

### Explanation of Project Code

We are constantly updating and improving our projects so the code you see may be slightly different, such as M in the video now being named num\_particles. Refer to the repository for the most up to date instructions.

Go to the next lesson to read more about the project and get the link to the github repo.

NEXT

## Additional Resources on Localization

Nice work reaching the end of the localization content! While you still have the project left to do here, we're also providing some additional resources and recent research on the topic that you can come back to if you have time later on.

Reading research papers is a great way to get exposure to the latest and greatest in the field, as well as expand your learning. However, just like the project ahead, it's often best to learn by doing - if you find a paper that really excites you, try to implement it (or even something better) yourself!

##### Optional Reading

All of these are completely optional reading - you could spend hours reading through the entirety of these! We suggest moving onto the project first so you have what you’ve learned fresh on your mind, before coming back to check these out.

We've categorized these papers to hopefully help you narrow down which ones might be of interest, as well as highlighted a couple key reads by category by including their Abstract section, which summarizes the paper.

### Simultaneous Localization and Mapping (SLAM)

The below papers cover Simultaneous Localization and Mapping (SLAM) - which as the name suggests, combines localization and mapping into a single algorithm without a map created beforehand.

[Past, Present, and Future of Simultaneous Localization And Mapping: Towards the Robust-Perception Age](https://arxiv.org/abs/1606.05830) by C. Cadena, et. al.

***Abstract:****Simultaneous Localization and Mapping (SLAM) consists in the concurrent construction of a model of the environment (the map), and the estimation of the state of the robot moving within it. The SLAM community has made astonishing progress over the last 30 years, enabling large-scale real-world applications, and witnessing a steady transition of this technology to industry. We survey the current state of SLAM. We start by presenting what is now the de-facto standard formulation for SLAM. We then review related work, covering a broad set of topics including robustness and scalability in long-term mapping, metric and semantic representations for mapping, theoretical performance guarantees, active SLAM and exploration, and other new frontiers. [...]*

[Navigating the Landscape for Real-time Localisation and Mapping for Robotics and Virtual and Augmented Reality](https://arxiv.org/abs/1808.06352) by S. Saeedi, et. al.

***Abstract:****Visual understanding of 3D environments in real-time, at low power, is a huge computational challenge. Often referred to as SLAM (Simultaneous Localisation and Mapping), it is central to applications spanning domestic and industrial robotics, autonomous vehicles, virtual and augmented reality. This paper describes the results of a major research effort to assemble the algorithms, architectures, tools, and systems software needed to enable delivery of SLAM, by supporting applications specialists in selecting and configuring the appropriate algorithm and the appropriate hardware, and compilation pathway, to meet their performance, accuracy, and energy consumption goals. [...]*

### Other Methods

The below paper from Udacity's founder Sebastian Thrun, while from 2002, is still relevant for many different methods of mapping used today in robotics.

[Robotic Mapping: A Survey](http://robots.stanford.edu/papers/thrun.mapping-tr.pdf) by S. Thrun

***Abstract:****This article provides a comprehensive introduction into the field of robotic mapping, with a focus on indoor mapping. It describes and compares various probabilistic techniques, as they are presently being applied to a vast array of mobile robot mapping problems. The history of robotic mapping is also described, along with an extensive list of open research problems.*

NEXT