# Course 4 – Localization

## Introduction to Localization

### Localization in Robotics

Video 1.1

Resources for Additional Localization Knowledge:

Textbook: [Probabilistic Robotics](http://www.probabilistic-robotics.org/) by Sebastian Thrun,‎ Wolfram Burgard,‎ and Dieter Fox.

Udacity's [AI for RoboticsFree](https://www.udacity.com/course/artificial-intelligence-for-robotics--cs373) Course

### Localization Changes

Video 1.2

## Quiz

Select all of the correct statements regarding localization.

- [ ] In position tracking, the robot's initial pose is unknown.

- [ ] In global localization, the robot's initial pose is unknown.

- [ ] The position tracking problem is easier to solve than the global localization one.

- [ ] In the kidnapped robot problem, the robot is teleported to a different location.

- [ ] The global localization problem is harder to solve than the kidnapped robot problem.

### Overview

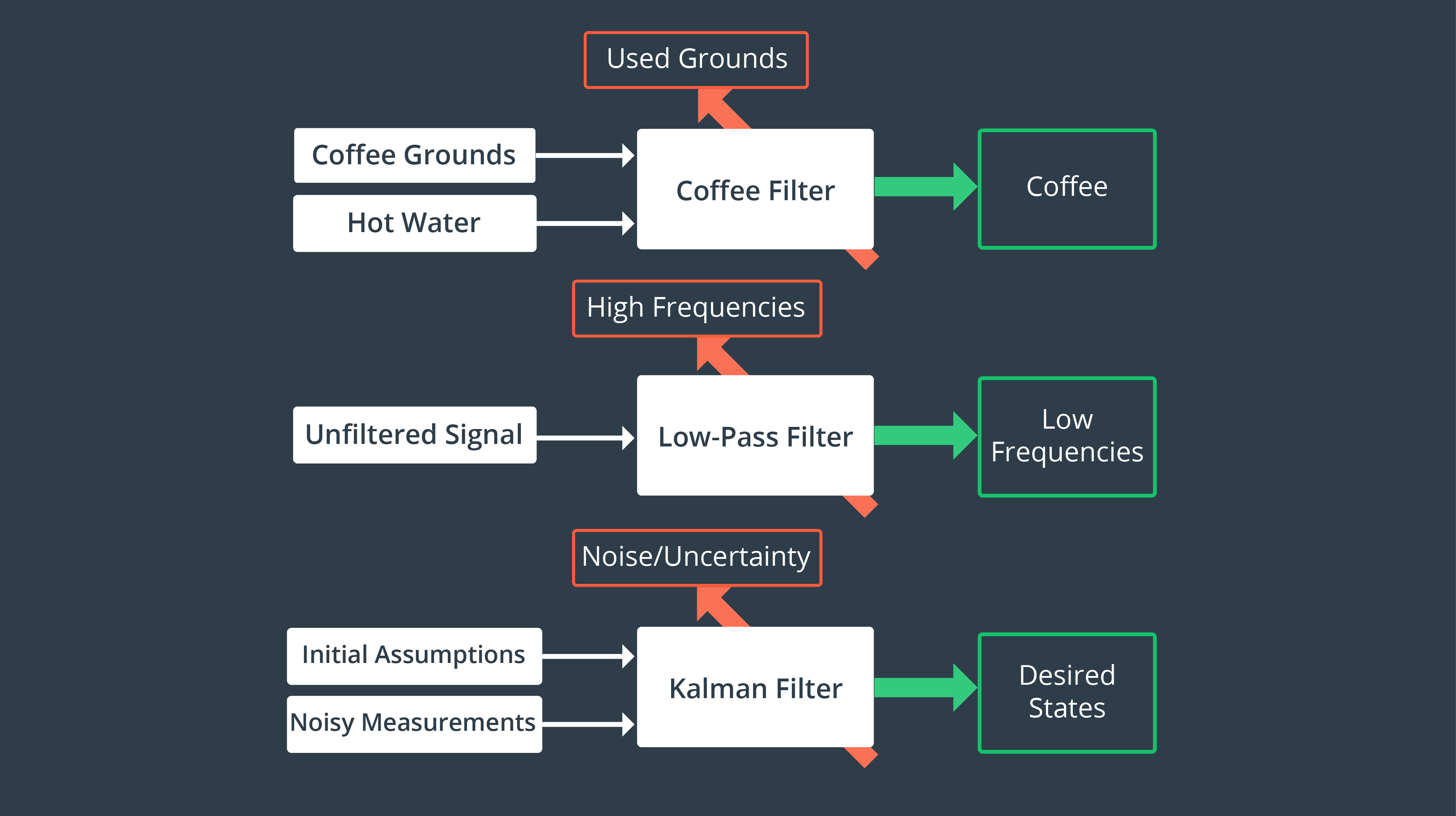
*Video 1.3*

## Kalman Filters

### Overview

*Video 2.1*

Another way of looking at a Kalman Filter is just like you’d look at any other filter. What does it take as an input, what does it filter out, and what important substance does it let through? The graphic below compares a household coffee filter, an engineering low-pass filter, and a Kalman filter.



### What's a Kalman Filter?

*Video 2.2*

### History

*Video 2.3*

### Applications

*Video 2.4*

### Variations

*Video 2.5*

**Instructor's Notes:**

[**UKF-Wikipedia**](https://en.wikipedia.org/wiki/Kalman_filter#Unscented_Kalman_filter)

[**UKF by Cyrill Stachniss, University of Freiburg**](http://ais.informatik.uni-freiburg.de/teaching/ws12/mapping/pdf/slam05-ukf.pdf)

### Robot Uncertainty

*Video 2.6*

You may notice that the distribution around the 20m mark is wider than at the 30m mark. Generally, as a robot moves, uncertainty in its position can increase due to factors like wheel slip or external forces, which broaden the distribution. However, this isn’t always a strict rule. For example, if the robot experienced less randomness (e.g., minimal slip or external disturbances) between the 20m and 30m marks, its position estimate might become more certain over that range, resulting in a narrower distribution at the 30m mark.

In real-world localization, a robot’s position estimate can become more concentrated around its true location as it gathers additional data. You’ll see this effect in action during the 'Where Am I' project at the end of this course.

### Kalman Filter Advantage

*Video 2.7*

### 1D Gaussian

At the basis of the Kalman Filter is the Gaussian distribution, sometimes referred to as a bell curve or normal distribution. Recall the rover example - after executing one motion, the rover’s location was represented by a Gaussian. It’s exact location was not certain, but the level of uncertainty was bounded. It was unlikely that the rover would be more than a few meters away from its target location, and it would be nearly impossible for it to show up at the 50 meter mark.

A long line of black squares

AI-generated content may be incorrect.

This is the role of a Kalman Filter - after a movement or a measurement update, it outputs a unimodal Gaussian distribution. This is its best guess at the true value of a parameter.

A Gaussian distribution is a probability distribution, which is a continuous function. The probability that a random variable, x, will take a value between x1*x*1​ and x2*x*2​ is given by the integral of the function from x1*x*1​ to x2*x*2​.

A math equation with numbers and symbols

AI-generated content may be incorrect.

In the image below, the probability of the rover being located between 8.7m and 9m is 7%.

A diagram of a normal distribution

AI-generated content may be incorrect.

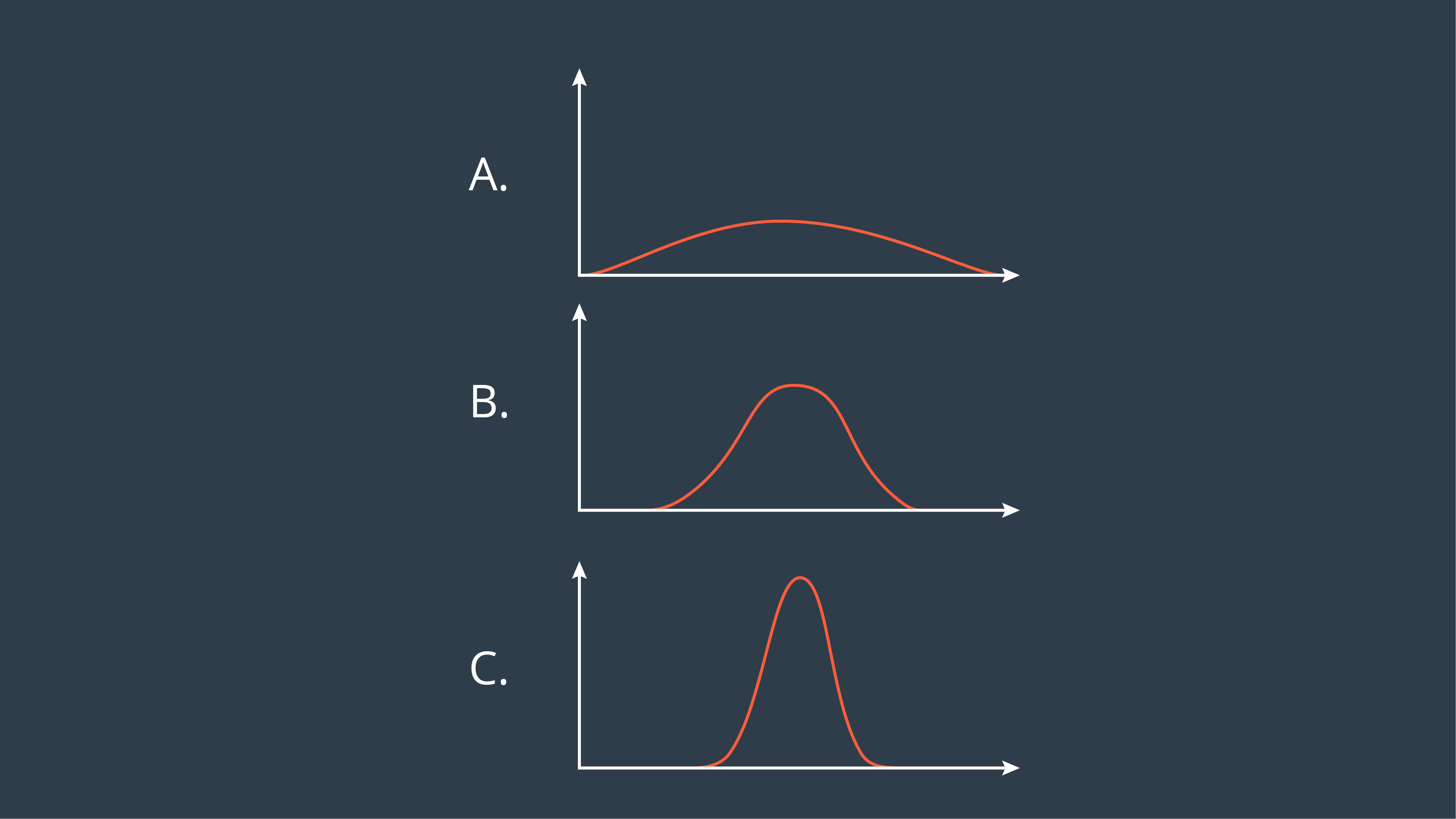
**Mean and Variance**

A Gaussian is characterized by two parameters - its mean (μ) and its variance (σ²). The mean is the most probable occurrence and lies at the centre of the function, and the variance relates to the width of the curve. The term unimodal implies a single peak present in the distribution.

Gaussian distributions are frequently abbreviated as N(x: μ, σ²), and will be referred to in this way throughout the coming lessons.

It's time for a quiz! Reference the image below to answer the quiz question.

Question 1 of 3 If you had to pick a Gaussian to represent the location of your rover, which of the following would you prefer?



The formula for the Gaussian distribution is printed below. Notice that the formula contains an exponential of a quadratic function. The quadratic compares the value of x to μ, and in the case that x=μ, the exponential is equal to 1 (e0=1*e*0=1). You’ll note here, that the constant in front of the exponential is a necessary normalizing factor.

A math equation with numbers and symbols

AI-generated content may be incorrect.

Just like with discrete probability, like a coin toss, the probabilities of all the options must sum to one. Therefore, the area underneath the function always sums to one.

A black text on a white background

AI-generated content may be incorrect.

Now that you are familiar with the formula, it’s time to code the Gaussian in C++. This will allow you to calculate the probability of a value occurring given a mean and a variance!

//main.cpp

#include <iostream>

#include <math.h>

using namespace std;

double f(double mu, double sigma2, double x)

{

    //Use mu, sigma2 (sigma squared), and x to code the 1-dimensional Gaussian

    //Put your code here

    //double prob =

    return prob;

}

int main()

{

    cout << f(10.0, 4.0, 8.0) << endl;

    return 0;

}

//solution.cpp

#include <iostream>

#include <math.h>

using namespace std;

double f(double mu, double sigma2, double x)

{

    //Use mu, sigma2 (sigma squared), and x to code the 1-dimensional Gaussian

    //Put your code here

    double prob = 1.0 / sqrt(2.0 \* M\_PI \* sigma2) \* exp(-0.5 \* pow((x - mu), 2.0) / sigma2);

    return prob;

}

int main()

{

    cout << f(10.0, 4.0, 8.0) << endl;

    return 0;

}

Great, you’ve coded the formula for a Gaussian distribution, now let’s make sure you know where it is applied!

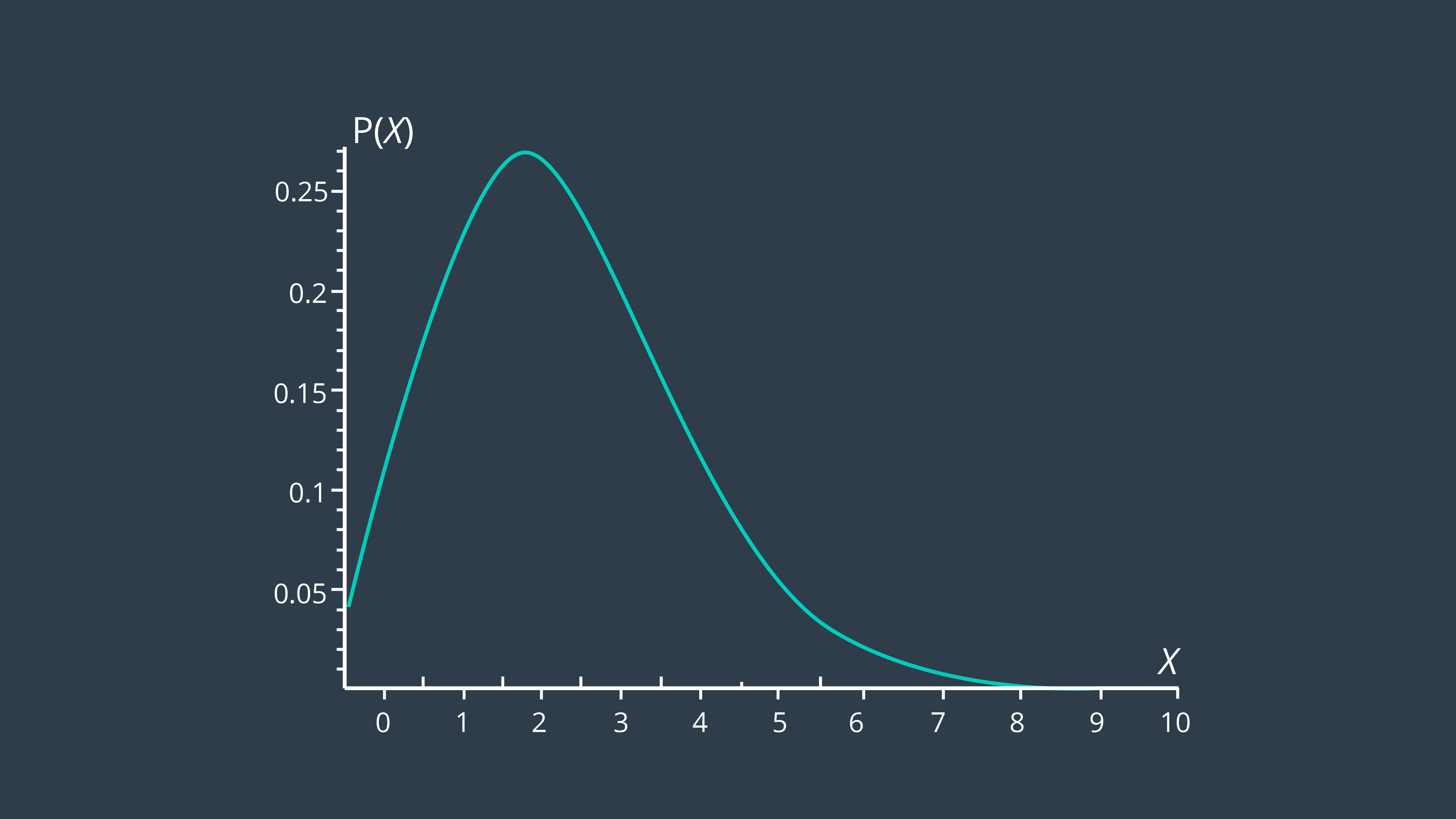
A screenshot of a computer

AI-generated content may be incorrect.

That’s right, the Kalman Filter treats all noise as unimodal Gaussian. In reality, that’s not the case. However, the algorithm is optimal if the noise is Gaussian. The term optimal expresses that the algorithm minimizes the mean square error of the estimated parameters.

**Distribution Quiz**

Reference the following probability distribution to answer the quiz question below.



Question 3 of 3 Can a state with this probability distribution be solved using the Kalman Filter? Y/N

That’s all the mathematics that you need to know for now. Let’s start designing Kalman Filters!

### Designing 1D Kalman Filters

*Video 2.9*

A white background with black text

AI-generated content may be incorrect.

### Measurement Update

*Video 2.10*

*μ*: Mean of the prior belief  
σ2*σ*2: Variance of the prior belief  
  
v*v*: Mean of the measurement  
r2*r*2: Variance of the measurement

**New Belief Quiz**

A diagram of a normal distribution

AI-generated content may be incorrect.

Question 1 of 2 Where do you think the robot's new belief will be? A/B/C

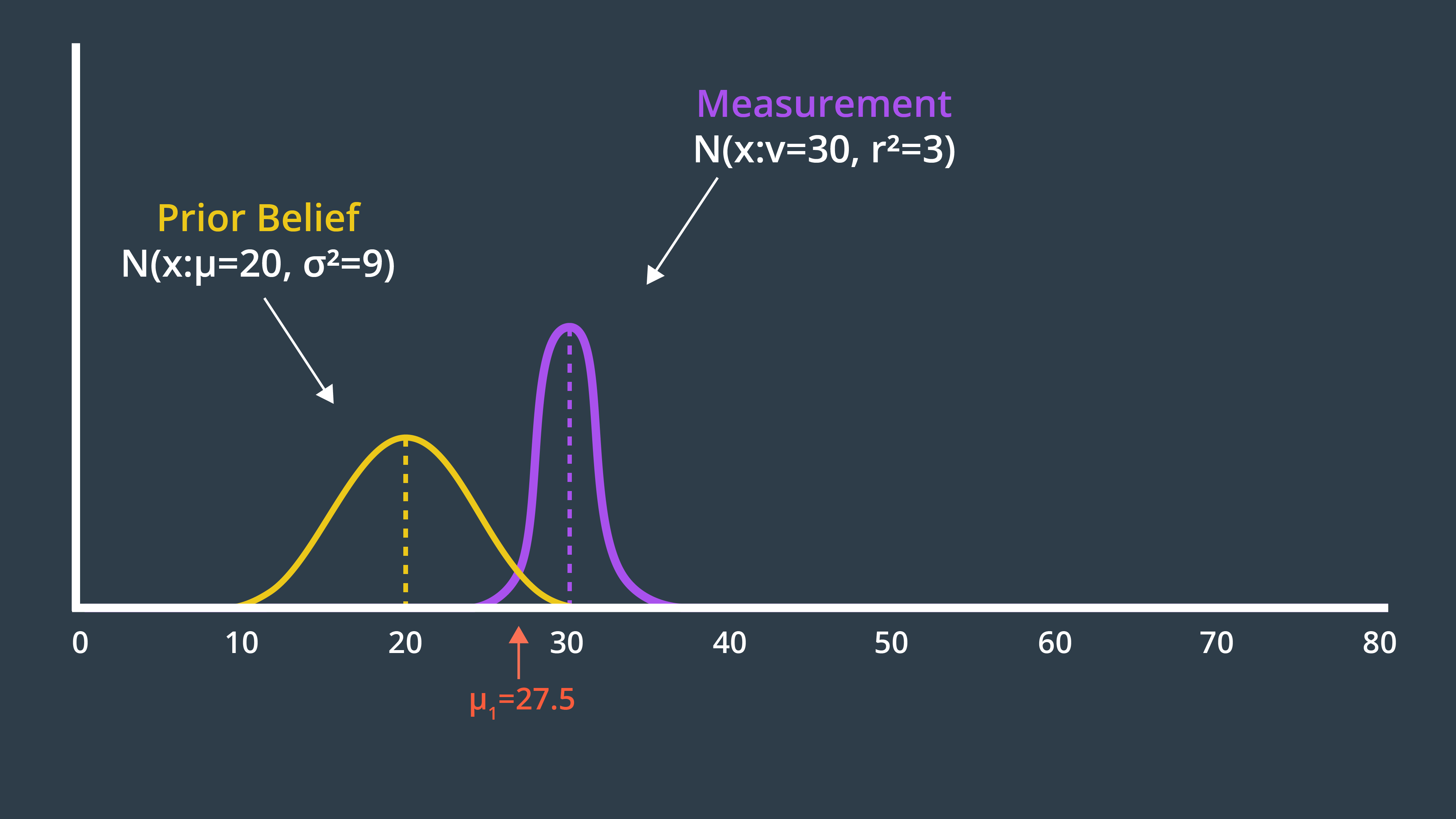
*Video 2.10.1*

The new mean is a weighted sum of the prior belief and measurement means. With uncertainty, a larger number represents a more uncertain probability distribution. However, the new mean should be biased towards the measurement update, which has a smaller standard deviation than the prior. How do we accomplish this?

A math equation with numbers and plus and two plus

AI-generated content may be incorrect.

The answer is - the uncertainty of the prior is multiplied by the mean of the measurement, to give it more weight, and similarly the uncertainty of the measurement is multiplied with the mean of the prior. Applying this formula to our example generates a new mean of 27.5, which we can label on our graph below.



**Variance Calculation**

Next, we need to determine the variance of the new state estimate.

A white box with black text

AI-generated content may be incorrect.

The two Gaussians provide us with more information together than either Gaussian offered alone. As a result, our new state estimate is more confident than our prior belief and our measurement. This means that it has a higher peak and is narrower. You can see this in the graph below.

A diagram of a normal distribution

AI-generated content may be incorrect.

The formula for the new variance is presented below.

A math equation with numbers and symbols

AI-generated content may be incorrect.

Entering the variances from our example into this formula produces a new variance of 2.25. The new state estimate, often called the posterior, is drawn below.

A diagram of a normal distribution

AI-generated content may be incorrect.

A white background with black text

AI-generated content may be incorrect.

**Programming Quiz**

In this C++ code, the **measurement update** function returns two values: the newly computed mean and variance. Usually, a ***tuple*** or ***struct*** should be used in C++ to return more than one value from a function and easily assign them later to multiple variables. For more information on ***tuples*** and ***structs*** take a look at this [**link**](https://dzone.com/articles/returning-multiple-values-from-functions-in-c).

//main.cpp

#include <iostream>

#include <math.h>

#include <tuple>

using namespace std;

double new\_mean, new\_var;

tuple<double, double> measurement\_update(double mean1, double var1, double mean2, double var2)

{

    new\_mean = //TODO: Code the measurment update mean function mu;

    new\_var =  //TODO: Code the measurment update variance function sigma square;

    return make\_tuple(new\_mean, new\_var);

}

int main()

{

    tie(new\_mean, new\_var) = measurement\_update(10, 8, 13, 2);

    printf("[%f, %f]", new\_mean, new\_var);

    return 0;

}

//solution.cpp

#include <iostream>

#include <math.h>

#include <tuple>

using namespace std;

double new\_mean, new\_var;

tuple<double, double> measurement\_update(double mean1, double var1, double mean2, double var2)

{

    new\_mean = (var2 \* mean1 + var1 \* mean2) / (var1 + var2);

    new\_var = 1 / (1 / var1 + 1 / var2);

    return make\_tuple(new\_mean, new\_var);

}

int main()

{

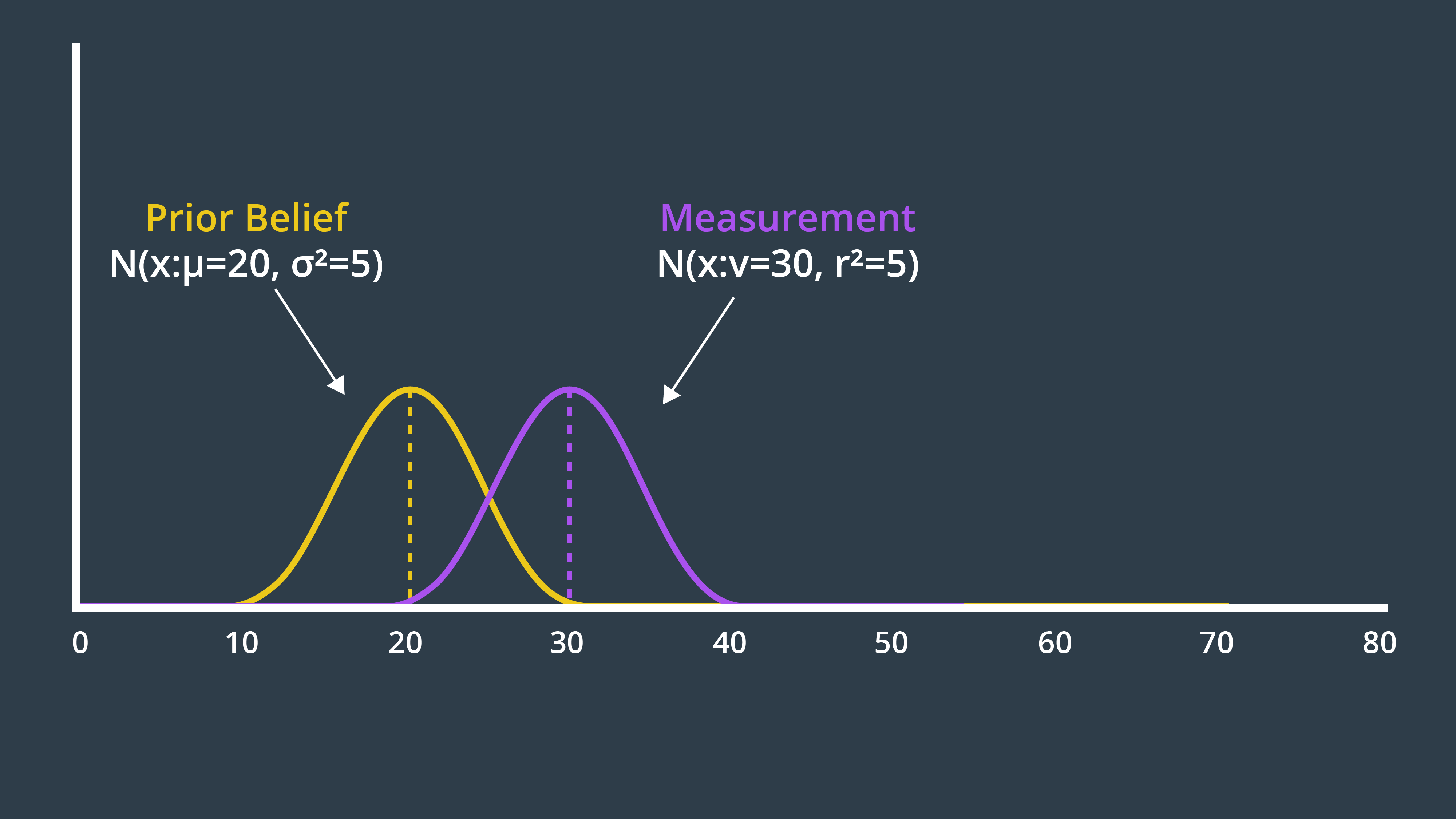
    tie(new\_mean, new\_var) = measurement\_update(10, 8, 13, 2);

    printf("[%f, %f]", new\_mean, new\_var);

    return 0;

}

I encourage you to think about what the posterior Gaussian would look like for the following example, and even calculate the exact values using your measurement\_update function.



### State Prediction

*Video 2.11*

*A black text on a white background

AI-generated content may be incorrect.*

//main.cpp

#include <iostream>

#include <math.h>

#include <tuple>

using namespace std;

double new\_mean, new\_var;

tuple<double, double> state\_prediction(double mean1, double var1, double mean2, double var2)

{

    new\_mean = //TODO: Code the state prediction mean function mu;

    new\_var =  //TODO: Code the state prediction variance function sigma square;

    return make\_tuple(new\_mean, new\_var);

}

int main()

{

    tie(new\_mean, new\_var) = state\_prediction(10, 4, 12, 4);

    printf("[%f, %f]", new\_mean, new\_var);

    return 0;

}

//solution.cpp

#include <iostream>

#include <math.h>

#include <tuple>

using namespace std;

double new\_mean, new\_var;

tuple<double, double> state\_prediction(double mean1, double var1, double mean2, double var2)

{

    new\_mean = mean1 + mean2;

    new\_var =  var1 + var2;

    return make\_tuple(new\_mean, new\_var);

}

int main()

{

    tie(new\_mean, new\_var) = state\_prediction(10, 4, 12, 4);

    printf("[%f, %f]", new\_mean, new\_var);

    return 0;

}

### 1D Kalman Filter

*Video 2.12*

**Note on Sigma (σ)**

In this video, we use motion\_sig and measurement\_sig to represent **variance**, which indicates uncertainty in motion and measurement. However, sigma (σ) typically means **standard deviation**, not variance. We apologize for this oversight. A more accurate naming would be:

* motion\_variance instead of motion\_sig
* measurement\_variance instead of measurement\_sig

In the programming quiz below, write the code that will iteratively go through the available measurements and motions, and apply a measurement update or a state prediction to each one of them.

//main.cpp

#include <iostream>

#include <math.h>

#include <tuple>

using namespace std;

double new\_mean, new\_var;

tuple<double, double> measurement\_update(double mean1, double var1, double mean2, double var2)

{

    new\_mean = (var2 \* mean1 + var1 \* mean2) / (var1 + var2);

    new\_var = 1 / (1 / var1 + 1 / var2);

    return make\_tuple(new\_mean, new\_var);

}

tuple<double, double> state\_prediction(double mean1, double var1, double mean2, double var2)

{

    new\_mean = mean1 + mean2;

    new\_var = var1 + var2;

    return make\_tuple(new\_mean, new\_var);

}

int main()

{

    //Measurements and measurement variance

    double measurements[5] = { 5, 6, 7, 9, 10 };

    double measurement\_sig = 4;

    //Motions and motion variance

    double motion[5] = { 1, 1, 2, 1, 1 };

    double motion\_sig = 2;

    //Initial state

    double mu = 0;

    double sig = 1000;

    //######TODO: Put your code here below this line######//

    // Loop through all the measurments

        // Apply a measurment update

        printf("update:  [%f, %f]\n", mu, sig);

        // Apply a state prediction

        printf("predict: [%f, %f]\n", mu, sig);

    return 0;

}

//solution.cpp

#include <iostream>

#include <math.h>

#include <tuple>

using namespace std;

double new\_mean, new\_var;

tuple<double, double> measurement\_update(double mean1, double var1, double mean2, double var2)

{

    new\_mean = (var2 \* mean1 + var1 \* mean2) / (var1 + var2);

    new\_var = 1 / (1 / var1 + 1 / var2);

    return make\_tuple(new\_mean, new\_var);

}

tuple<double, double> state\_prediction(double mean1, double var1, double mean2, double var2)

{

    new\_mean = mean1 + mean2;

    new\_var = var1 + var2;

    return make\_tuple(new\_mean, new\_var);

}

int main()

{

    //Measurements and measurement variance

    double measurements[5] = { 5, 6, 7, 9, 10 };

    double measurement\_sig = 4;

    //Motions and motion variance

    double motion[5] = { 1, 1, 2, 1, 1 };

    double motion\_sig = 2;

    //Initial state

    double mu = 0;

    double sig = 1000;

    for (int i = 0; i < sizeof(measurements) / sizeof(measurements[0]); i++) {

        tie(mu, sig) = measurement\_update(mu, sig, measurements[i], measurement\_sig);

        printf("update:  [%f, %f]\n", mu, sig);

        tie(mu, sig) = state\_prediction(mu, sig, motion[i], motion\_sig);

        printf("predict: [%f, %f]\n", mu, sig);

    }

    return 0;

}

**Additional Challenge**

You should try experimenting with different initial values - for instance changing the initial estimate to something absurd, and then adjusting the confidence value from 1000 down to a single digit number. At what point does the Kalman Filter start to be affected by the incorrect initial estimate? How robust is the filter to errors?

### Multivariate Gaussian

Most robots that we would be interested in modeling are moving in more than one dimension. For instance, a robot on a plane would have an x & y position.

The simple approach to take, would be to have a 1-dimensional Gaussian represent each dimension - one for the x-axis and one for the y-axis.

Do you see any problems with this?

A screenshot of a question

AI-generated content may be incorrect.

The image below depicts a two-dimensional Gaussian distribution.

A graph of a cone shaped graph

AI-generated content may be incorrect.

Let's dive into the details!

*Video 2.13*

**Formulas for the Multivariate Gaussian**

**A screenshot of a math problem

AI-generated content may be incorrect.**

### Intro to Multidimensional KF

*Video 2.14*

### Design of Multidimensional KF

Design of Multi-Dimensional Kalman Filters

From this point forward we will transition to using linear algebra, as it allows us to easily work with multi-dimensional problems. To begin with, let’s write the state prediction in linear algebra form.

**State Transition**

The formula below is the state transition function that advances the state from time *t* to time *t + 1*. It is just the relationship between the robot’s position, *x*, and velocity, *x*˙. Here, we will assume that the robot’s velocity is not changing.

A math symbols and symbols

AI-generated content may be incorrect.

We can express the same relationship in matrix form, as seen below. On the left, is the posterior state (denoted with the prime symbol, ′), and on the right are the state transition function and the prior state. This equation shows how the state changes over the time period, Δt. Note that we are only working with the means here; the covariance matrix will appear later.

A math symbols with numbers

AI-generated content may be incorrect.

The State Transition Function is denoted F*F*, and the formula can be written as so,



In reality, the equation should also account for process noise, as its own term in the equation. However, process noise is a Gaussian with a mean of 0, so the update equation for the mean need not include it.

A math symbols on a white background

AI-generated content may be incorrect.

**Sidenote:** While it is common to use Σ to represent the covariance of a Gaussian distribution in mathematics, it is more common to use the letter P*P* to represent the state covariance in localization.

If you multiply the state, *x*, by *F*, then the covariance will be affected by the square of *F*. In matrix form, this will look like so:



However, your intuition may suggest that it should be affected by more than just the state transition function. For instance, additional uncertainty may arise from the prediction itself. If so, you’re correct!

To calculate the posterior covariance, the prior covariance is multiplied by the state transition function squared, and *Q* added as an increase of uncertainty due to process noise. *Q* can account for a robot slowing down unexpectedly, or being drawn off course by an external influence.

A black and white text

AI-generated content may be incorrect.

Now we’ve updated the mean and the covariance as part of the state prediction.

**Quiz 1**

Now that you've seen how a simple state transition function is created, let's see if you can construct a more complicated one for the following problem:

You are tracking the position and velocity of a robot in two dimensions, x and y. The state is represented as so,

A mathematical equation with black text

AI-generated content may be incorrect.

Find the state update function, F, that will advance the state from time *t* to time *t + 1* based on the state transition equation below.



Try to work through this on paper before looking at the quiz options below.

A screenshot of a question

AI-generated content may be incorrect.

**Quiz 2**

You are tracking the position, velocity, and acceleration of a quadrotor in the vertical dimension, z. The state of the quadrotor can be represented as so,

A black and white image of a mathematical equation

AI-generated content may be incorrect.

Find the state update function, F, that will advance the state from time *t* to time *t + 1* based on the state transition equation below.



A screenshot of a question

AI-generated content may be incorrect.

**Measurement Update**

Next, we move onto the measurement update step. If we return to our original example, where we were tracking the position and velocity of a robot in the x-dimension, the robot was taking measurements of the location only (the velocity is a hidden state variable). Therefore the measurement function is very simple - a matrix containing a one and a zero. This matrix demonstrates how to map the state to the observation, *z*.

A close up of numbers

AI-generated content may be incorrect.

This matrix, called the Measurement Function, is denoted *H*.

For the measurement update step, there are a few formulas. First, we calculate the measurement residual, *y*. The measurement residual is the difference between the measurement and the expected measurement based on the prediction (ie. we are comparing where the measurement *tells us* we are vs. where we *think* we are). The measurement residual will be used later on in a formula.



Next, it's time to consider the measurement noise, denoted *R*. This formula maps the state prediction covariance into the measurement space and adds the measurement noise. The result, *S*, will be used in a subsequent equation to calculate the Kalman Gain.



These equations need not be memorized, instead they can be referred to in text or implemented in code for use and reuse.

**Kalman Gain**

Next, we calculate the Kalman Gain, K. As you will see in the next equation, the Kalman Gain determines how much weight should be placed on the state prediction, and how much on the measurement update. It is an averaging factor that changes depending on the uncertainty of the state prediction and measurement update.

A math equations with black text

AI-generated content may be incorrect.

These equations may look complicated and intimidating, but they do nothing more than calculate an average factor. Let’s work through a quick example to gain a better understanding of this. Feel free to pause the video and follow along in your own notebook!

*Video 2.15*

Notes:

* The second equation on the left should use *P*′ instead of *P* on the left side of the equation.
* The inverse of *H* matrix may not exist, but it is used for simplification. In practice, the Kalman filter operates with the matrix *HP*′*HT* which is square and invertible.

The last step in the Kalman Filter is to update the new state’s covariance using the Kalman Gain.



A screenshot of a math equation

AI-generated content may be incorrect.

The Kalman Filter can successfully recover from inaccurate initial estimates, but it is very important to estimate the noise parameters, Q and R, as accurately as possible - as they are used to determine which of the estimate or the measurement to believe more.

**Programming Exercise**

Now it’s your chance to code the multi-dimensional Kalman Filter in C++. The code below uses the C++ ***eigen*** library to define matrices and easily compute their inverse and transpose. Check out the ***eigen*** library full documentation [**here**](https://eigen.tuxfamily.org/dox/group__QuickRefPage.html) and go through some of their examples. Here's a list of useful commands that you'll need while working on this quiz:

* Initializing a 2x1 float matrix **K**: *MatrixXf K(2, 1);*
* Inserting values to matrix **K**: *K << 0, 0*
* Computing the transpose of matrix **K**: *K.transpose()*
* Computing the inverse of matrix **K**: *K.inverse()*

//main.cpp

#include <iostream>

#include <math.h>

#include <tuple>

#include "Core" // Eigen Library

#include "LU"   // Eigen Library

using namespace std;

using namespace Eigen;

float measurements[3] = { 1, 2, 3 };

tuple<MatrixXf, MatrixXf> kalman\_filter(MatrixXf x, MatrixXf P, MatrixXf u, MatrixXf F, MatrixXf H, MatrixXf R, MatrixXf I)

{

    for (int n = 0; n < sizeof(measurements) / sizeof(measurements[0]); n++) {

        //\*\*\*\*\*\* TODO: Kalman-filter function\*\*\*\*\*\*\*\*//

        // Measurement Update

        // Code the Measurement Update

        // Initialize and Compute Z, y, S, K, x, and P

        // Prediction

        // Code the Prediction

        // Compute x and P

    }

    return make\_tuple(x, P);

}

int main()

{

    MatrixXf x(2, 1);// Initial state (location and velocity)

    x << 0,

         0;

    MatrixXf P(2, 2);//Initial Uncertainty

    P << 100, 0,

         0, 100;

    MatrixXf u(2, 1);// External Motion

    u << 0,

         0;

    MatrixXf F(2, 2);//Next State Function

    F << 1, 1,

         0, 1;

    MatrixXf H(1, 2);//Measurement Function

    H << 1,

         0;

    MatrixXf R(1, 1); //Measurement Uncertainty

    R << 1;

    MatrixXf I(2, 2);// Identity Matrix

    I << 1, 0,

         0, 1;

    tie(x, P) = kalman\_filter(x, P, u, F, H, R, I);

    cout << "x= " << x << endl;

    cout << "P= " << P << endl;

    return 0;

}

//solution.cpp

#include <iostream>

#include <math.h>

#include <tuple>

#include "Core" // Eigen Library

#include "LU"   // Eigen Library

using namespace std;

using namespace Eigen;

float measurements[3] = { 1, 2, 3 };

tuple<MatrixXf, MatrixXf> kalman\_filter(MatrixXf x, MatrixXf P, MatrixXf u, MatrixXf F, MatrixXf H, MatrixXf R, MatrixXf I)

{

    for (int n = 0; n < sizeof(measurements) / sizeof(measurements[0]); n++) {

        // Measurement Update

        MatrixXf Z(1, 1);

        Z << measurements[n];

        MatrixXf y(1, 1);

        y << Z - (H \* x);

        MatrixXf S(1, 1);

        S << H \* P \* H.transpose() + R;

        MatrixXf K(2, 1);

        K << P \* H.transpose() \* S.inverse();

        x << x + (K \* y);

        P << (I - (K \* H)) \* P;

        // Prediction

        x << (F \* x) + u;

        P << F \* P \* F.transpose();

    }

    return make\_tuple(x, P);

}

int main()

{

    MatrixXf x(2, 1);// Initial state (location and velocity)

    x << 0,

         0;

    MatrixXf P(2, 2);//Initial Uncertainty

    P << 100, 0,

         0, 100;

    MatrixXf u(2, 1);// External Motion

    u << 0,

         0;

    MatrixXf F(2, 2);//Next State Function

    F << 1, 1,

         0, 1;

    MatrixXf H(1, 2);//Measurement Function

    H << 1,

         0;

    MatrixXf R(1, 1); //Measurement Uncertainty

    R << 1;

    MatrixXf I(2, 2);// Identity Matrix

    I << 1, 0,

         0, 1;

    tie(x, P) = kalman\_filter(x, P, u, F, H, R, I);

    cout << "x= " << x << endl;

    cout << "P= " << P << endl;

    return 0;

}

### Introduction to EKF

*Video 2.16*

**Additional Resources:**

[**Taylor series Wikipedia page**](https://en.wikipedia.org/wiki/Taylor_series)

**Summary**

The Kalman Filter is applicable to problems with linear motion and measurement functions. This is limiting, as much of the real world is nonlinear.

A nonlinear function can be used to update the mean of a function,



but not the variance, as this would result in a non-Gaussian distribution which is *much* more computationally expensive to work with. To update the variance, the Extended Kalman Filter linearizes the nonlinear function f(x) over a small section and calls it F. This linearization, F, is then used to update the state's variance.

A black and white text

AI-generated content may be incorrect.

The linear approximation can be obtained by using the first two terms of the Taylor Series of the function centered around the mean.

A math equation with numbers and symbols

AI-generated content may be incorrect.

### EKF

**Multi-dimensional Extended Kalman Filter**

Now you’ve seen the fundamentals behind the Extended Kalman Filter. The mechanics are not too different from the Kalman Filter, with the exception of needing to linearize a nonlinear motion or measurement function to be able to update the variance.

You’ve seen how this can be done for a state prediction or measurement function that is of one-dimension, but now it’s time to explore how to linearize functions with multiple dimensions. To do this, we will be using multi-dimensional Taylor series.

A white text with black text

AI-generated content may be incorrect.

**Example Application**

This will make more sense in context, so let’s look at a specific example. Let’s say that we are tracking the x-y coordinate of an object. This is to say that our state is a vector x, with state variables x and y.

A black and white image of a mathematical equation

AI-generated content may be incorrect.

However, our sensor does not allow us to measure the x and y coordinates of the object directly. Instead, our sensor measures the distance from the robot to the object, r, as well as the angle between r and the x-axis, θ.

A black and white image of a mathematical equation

AI-generated content may be incorrect.

It is important to notice that our state is using a Cartesian representation of the world, while the measurements are in a polar representation. How will this affect our measurement function?

Our measurement function maps the state to the observation, as so,

A black and white text

AI-generated content may be incorrect.

Thus, our measurement function must map from Cartesian to polar coordinates. But there is no matrix, H, that will successfully make this conversion, as the relationship between Cartesian and polar coordinates is nonlinear.

A math equations with black text

AI-generated content may be incorrect.

For this reason, instead of using the measurement residual equation *y*=*z*−*Hx*′ that you had seen before, the mapping must be made with a dedicated function, h(x').

A math equation with black text

AI-generated content may be incorrect.

Then the measurement residual equation becomes  *y*=*z*−*h*(*x*′).

Our measurement covariance matrix cannot be updated the same way, as it would turn into a non-Gaussian distribution (as seen in the previous video). Let's calculate a linearization, H, and use it instead. The Taylor series for the function h(x), centered about the mean μ, is defined below.

A black and yellow math equation

AI-generated content may be incorrect.

The Jacobian, *Df*(*μ*), is defined below. But let's call it H since it's the linearization of our measurement function, h(x).

A black and white image of a mathematical equation

AI-generated content may be incorrect.

If you were to compute each of those partial derivatives, the matrix would reduce to the following,

A black and white math equations

AI-generated content may be incorrect.

It's this matrix, H, that can then be used to update the state's covariance.

To summarize the flow of calculations for the Extended Kalman Filter, it's worth revisiting the equations to see what has changed and what has remained the same.

A screenshot of a math equation

AI-generated content may be incorrect.

**Summary**

Phew, that got complicated quickly! Here are the key take-aways about Extended Kalman Filters:

* The Kalman Filter cannot be used when the measurement and/or state transition functions are nonlinear, since this would result in a non-Gaussian distribution.
* Instead, we take a local linear approximation and use this approximation to update the covariance of the estimate. The linear approximation is made using the first terms of the Taylor Series, which includes the first derivative of the function.
* In the multi-dimensional case, taking the first derivative isn't as easy as there are multiple state variables and multiple dimensions. Here we employ a Jacobian, which is a matrix of partial derivatives, containing the partial derivative of each dimension with respect to each state variable.

While it's important to understand the underlying math to employ the Kalman Filter, don't feel the need to memorize these equations. Chances are, whatever software package or programming language you're working with will have libraries that allow you to apply the Kalman Filter, or at the very least perform linear algebra calculations (such as matrix multiplication and calculating the Jacobian).

### EKF Example

Let's look at another example of a vehicle taking measurements - this time, a quadrotor! This quadrotor is a bit simplified - it's motion is constrained to the y-axis. Therefore, it's state can be defined by the following vector,

A mathematical equation with black text

AI-generated content may be incorrect.

that is: its roll angle, its velocity, and its position.

Imagine you have a quadrotor, such as the one in the image below. This quadrotor would like to know the distance between it and the wall. This is an important measurement to have if the quadrotor would like to traverse the inside of a room, or outside of a building, while maintaining a safe distance from the wall.

To estimate this distance, the quadrotor is equipped with a range finger.

A blue and red line with a flying device

AI-generated content may be incorrect.

Shown in blue, are the true distances from an arbitrary point on the left to the quadrotor, and to the wall.

A screenshot of a computer

AI-generated content may be incorrect.

That's right - while the quadrotor is hovering perpendicular to the wall, the measurement, h(x)*h*(*x*), is equal to wall−y*wall*−*y*.

Now, what would happen if the quadrotor were to roll to some angle ϕ*ϕ*? What is a more general equation for the measurement that takes into account the roll angle?

A diagram of a wall-y

AI-generated content may be incorrect.

A screenshot of a math problem

AI-generated content may be incorrect.

Applying some basic trigonometry, we've now determined the measurement model for this quadrotor's range finder.  
A math equation with numbers and symbols

AI-generated content may be incorrect.

The function has a cosine in it's denominator, making this function non-linear. This means that we will need to use the Extended Kalman Filter for our estimation, and in the process, linearize the function.

**Calculating H**

To apply the Extended Kalman Filter, we will need to calculate H, the Jacobian of the measurement model that we defined above. This won't be too strenuous since the measurement function is a 1x1 matrix.

Without calculating the partial derivatives, which of the following is the correct Jacobian for the measurement model?

A screenshot of a math test

AI-generated content may be incorrect.

Calculating the three partial derivatives will result in the following,

A mathematical equations on a white background

AI-generated content may be incorrect.

When implementing the Extended Kalman Filter in code, there are software libraries that can take the partial derivative of a function, simplifying your implementation. There are, of course, also EKF implementations readily available too. However, it's *always* helpful to understand the inner workings of an algorithm and apply it intelligently to the problem at hand.

After calculating H,

A black text on a white background

AI-generated content may be incorrect.

it can be used in the EKF equations to update the covariance of the state.

As a reminder, the equations are provided below:

A screenshot of a math equation

AI-generated content may be incorrect.

### Recap

*Video 2.19*

## Lab: Kalman Filters

### Introduction

Video 3.1

### Sensor Fusion

Video 3.2

### Catkin Workspace

Before you start downloading the different ROS packages, you need to create a catkin\_ws to hold them in. If you already have a workspace in your **/home** directory, it is recommended that you keep a copy of it by renaming it to catkin\_ws\_saved. By doing so, you’ll avoid any possible conflict with pre-installed packages.

**Udacity Workspace**

For this lab, you'll get a chance to work in the Udacity Workspace. Thus, move to the next concept, enable GPU, and GO TO DESKTOP.

Now, follow these steps to create your catkin\_ws and perform a system update:

A screenshot of a computer program

AI-generated content may be incorrect.

### Udacity Workspace

* To follow along with the lab's instructions, use local VM image (Ubuntu 16.04 LTS) running on your VMWare/VirtualBox.
* Once you log into the VM image, open a Terminal window.
* You're now ready to follow along in your development environment with this lab!

### TurtleBot Gazebo Package

**Notice**: The commands provided here are compatible with ROS Kinetic or Melodic and may not work in that workspace. The online workspace provided earlier, as it runs ROS Noetic. For best results, use a local workspace with ROS Kinetic or Melodic. If you prefer to use ROS Noetic, you’ll need to look up the appropriate installation commands for TurtleBot3.

*Video 3.5*

**Turtlebot package:**

Access this [**link**](http://wiki.ros.org/turtlebot_gazebo) and go through the turtlebot\_gazebo documentation. After that, follow the instructions listed below to clone the turtlebot\_gazebo package in the src directory of your catkin\_ws and build it.

While scrolling through these instructions, you will notice that some statements are a bit different than the one presented in the videos. The reason behind that is the different environment used. In the videos, a **Virtual Machine** booted with LUbuntu is used. Here in the description, you are presented with instructions on how to replicate the steps in the **Udacity Workspace**. The only difference between what you see and what is listed is the directory of the catkin\_ws. In VM, you can store it anywhere you want, preferably in the root ~ directory. Whereas, in the workspace, you will have to store it under /home so it won't get deleted after a reboot.

A screenshot of a computer program

AI-generated content may be incorrect.

### Robot Pose EKF Package

*Video 3.6*

**EKF package:**

Access this [**link**](http://wiki.ros.org/robot_pose_ekf) and go through the robot\_pose\_ekf documentation.

A computer screen shot of a computer

AI-generated content may be incorrect.

A screenshot of a computer program

AI-generated content may be incorrect.

A screenshot of a computer program

AI-generated content may be incorrect.

A screenshot of a chat

AI-generated content may be incorrect.

A diagram of a company

AI-generated content may be incorrect.

### Odometry to Trajectory Package

*Video 3.7*

*A screenshot of a computer program

AI-generated content may be incorrect.*

A diagram of a diagram

AI-generated content may be incorrect.

Now that you’ve launched the nodes, open a new terminal and run the ***rqt graph***. In the active nodes and topics, you’ll now see the estimated ***3D pose*** since one of the nodes in the ***odom to trajectory*** package is subscribing to it. You’ll also notice that the ***trajectories*** are still invisible since we are not yet subscribed to them!

A white rectangular object with a black border

AI-generated content may be incorrect.

### TurtleBot Teleop Package

*Video 3.8*

**Teleop package:**

Access this [**link**](http://wiki.ros.org/turtlebot_teleop) and go through the turtlebot\_teleop documentation.

A screenshot of a computer program

AI-generated content may be incorrect.

### Rviz Package

Video 3.9

A screenshot of a computer

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.

A screenshot of a computer program

AI-generated content may be incorrect.

<launch>

  <!--RVIZ-->

  <node pkg="rviz" type="rviz" name="rviz" args="-d ~/catkin\_ws/src/EKFLab.rviz"/>

</launch>

### Main Launch

Video 3.10

A diagram of a product

AI-generated content may be incorrect.

The ***main.launch*** file located inside a ***main*** package. This launch file is a combination of all the 5 nodes we interfaced so far.

A screenshot of a computer

AI-generated content may be incorrect.

A screenshot of a computer program

AI-generated content may be incorrect.

### Rqt Multiplot

Video 3.11

**Instructions for Installing and Running the *rqt\_multiplot* ROS plugin:**

**GitHub documentation:**

Access this [**link**](https://github.com/ethz-asl/rqt_multiplot_plugin) and go through the ***rqt\_multiplot*** ROS plugin documentation.

A screenshot of a computer program

AI-generated content may be incorrect.

### Outro

Video 3.12

**Lab GitHub Repo:**

Access this [**link**](https://github.com/udacity/RoboND-EKFLab) and grab the lab from GitHub.

## Monte Carlo Localization

### Introduction

Video 4.1

**Additional References:**

Robust Monte Carlo Localization for Mobile Robots [**paper**](http://robots.stanford.edu/papers/thrun.robust-mcl.pdf) by Sebastian Thrun.

### What's MCL?

Video 4.2

A screenshot of a question

AI-generated content may be incorrect.

### Power of MCL

Video 4.3

MCL vs EKF:

A table with check marks and ticks

AI-generated content may be incorrect.

A screenshot of a question

AI-generated content may be incorrect.

### Particle Filters

Video 4.4

A screenshot of a question

AI-generated content may be incorrect.

### Bayes Filtering

The powerful Monte Carlo localization algorithm estimates the posterior distribution of a robot’s position and orientation based on sensory information. This process is known as a recursive Bayes filter.

Using a Bayes filtering approach, roboticists can estimate the **state** of a **dynamical system** from sensor **measurements**.

In mobile robot localization, it’s important to be acquainted with the following definitions:

* **Dynamical system**: The mobile robot and its environment
* **State**: The robot’s pose, including its position and orientation.
* **Measurements**: Perception data(e.g. laser scanners) and odometry data(e.g. rotary encoders)

The goal of Bayes filtering is to estimate a probability density over the state space conditioned on the measurements. The probability density, or also known as **posterior** is called the **belief** and is denoted as:

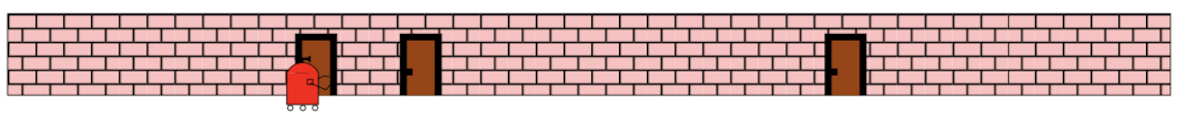
A screenshot of a math problem

AI-generated content may be incorrect.

A math equations on a white background

AI-generated content may be incorrect.

**Quiz:**



This robot is located inside of a 1D hallway which has three doors. The robot doesn't know where it is located in this hallway, but it has sensors onboard that can tell it, with some amount of precision, whether it is standing in front of a door, or in front of a wall. The robot also has the ability to move around - with some precision provided by its odometry data. Neither the sensors nor the movement is perfectly accurate, but the robot aims to locate itself in this hallway.

The mobile robot is now moving in the 1D hallway and collecting odometry and perception data. With the odometry data, the robot is keeping track of its current position. Whereas, with the perception data, the robot is identifying the presence of doors.

In this quiz, we are aiming to calculate the state of the robot, given its measurements. This is known by the belief: **P(Xt|Z)**!

**Given**:

* **P(POS)**: The probability of the robot being at the actual position
* **P(DOOR|POS)**: The probability of the robot seeing the door given that it’s in the actual position
* **P(DOOR|¬POS)**: The probability of the robot seeing the door given that it’s not in the actual position

**Compute**:

* **P(POS|DOOR)**: The belief or the probability of the robot being at the actual position given that it’s seeing the door.

//main.cpp

#include <iostream>

using namespace std;

int main() {

    //Given P(POS), P(DOOR|POS) and P(DOOR|¬POS)

    double a = 0.0002 ; //P(POS) = 0.002

    double b = 0.6    ; //P(DOOR|POS) = 0.6

    double c = 0.05   ; //P(DOOR|¬POS) = 0.05

    //TODO: Compute P(¬POS) and P(POS|DOOR)

    double d =                   //P(¬POS)

    double e =                   //P(POS|DOOR)

    //Print Result

    cout << "P(POS|DOOR)= " <<    e    << endl;

    return 0;

}

//solution.cpp

#include <iostream>

using namespace std;

int main() {

    //Given P(POS), P(DOOR|POS) and P(DOOR|¬POS)

    double a = 0.0002 ; //P(POS) = 0.002

    double b = 0.6    ; //P(DOOR|POS) = 0.6

    double c = 0.05   ; //P(DOOR|¬POS) = 0.05

    //TODO: Compute P(¬POS) and P(POS|DOOR)

    double d = 1-a ;                  //P(¬POS)

    double e =  (b\*a)/((a\*b)+(d\*c)) ; //P(POS|DOOR)

    //Print Result

    cout << "P(POS|DOOR)= " <<    e    << endl;

    return 0;

}

### MCL: The Algorithm

Video 4.6

A screenshot of a computer

AI-generated content may be incorrect.

A screenshot of a white background

AI-generated content may be incorrect.

### MCL in Action

Video 4.7

MCL vs EKF in Action

1- MCL:

A screenshot of a graph

AI-generated content may be incorrect.

**At time**:

* **t=1**, Particles are drawn randomly and uniformly over the entire pose space.
* **t=2**, Measurement is updated and an importance weight is assigned to each particle.
* **t=3**, Motion is updated and a new particle set with uniform weights and high number of particles around the three most likely places is obtained in resampling.
* **t=4**, Measurement assigns non-uniform weight to the particle set.
* **t=5**, Motion is updated and a new resampling step is about to start.

2- EKF:

A diagram of a red brick wall

AI-generated content may be incorrect.

**At time**:

* **t=1**, Initial belief represented by a Gaussian distribution around the first door.
* **t=2**, Motion is updated and the new belief is represented by a shifted Gaussian of increased weight.
* **t=3**, Measurement is updated and the robot is more certain of its location. The new posterior is represented by a Gaussian with a small variance.
* **t=4**, Motion is updated and the uncertainty increases.

A screenshot of a computer

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.

## Build MCL in C++

### Introduction

Video 5.1

### Robot Class

Video 5.2

**Robot Class**

Go through the Robot class and the functions in the C++ quiz section and try to understand the role of each one of them. After reviewing the robot class, print “I am ready for coding the MCL!” in the main function.

While scrolling through the C++ code, you'll notice some statements and functions commented out. These statements and functions are part of the matplotlib python library and are later used to graph and visualize the results. After you finish coding the MCL algorithm, you'll be asked to interface with this function on the Workspace and generate images to visualize the process of the MCL algorithm.

#### Codeblock

//#include "src/matplotlibcpp.h"//Graph Library

#include <iostream>

#include <string>

#include <math.h>

#include <vector>

#include <stdexcept> // throw errors

#include <random> //C++ 11 Random Numbers

//namespace plt = matplotlibcpp;

using namespace std;

// Landmarks

double landmarks[8][2] = { { 20.0, 20.0 }, { 20.0, 80.0 }, { 20.0, 50.0 },

    { 50.0, 20.0 }, { 50.0, 80.0 }, { 80.0, 80.0 },

    { 80.0, 20.0 }, { 80.0, 50.0 } };

// Map size in meters

double world\_size = 100.0;

// Random Generators

random\_device rd;

mt19937 gen(rd());

// Global Functions

double mod(double first\_term, double second\_term);

double gen\_real\_random();

class Robot {

public:

    Robot()

    {

        // Constructor

        x = gen\_real\_random() \* world\_size; // robot's x coordinate

        y = gen\_real\_random() \* world\_size; // robot's y coordinate

        orient = gen\_real\_random() \* 2.0 \* M\_PI; // robot's orientation

        forward\_noise = 0.0; //noise of the forward movement

        turn\_noise = 0.0; //noise of the turn

        sense\_noise = 0.0; //noise of the sensing

    }

    void set(double new\_x, double new\_y, double new\_orient)

    {

        // Set robot new position and orientation

        if (new\_x < 0 || new\_x >= world\_size)

            throw std::invalid\_argument("X coordinate out of bound");

        if (new\_y < 0 || new\_y >= world\_size)

            throw std::invalid\_argument("Y coordinate out of bound");

        if (new\_orient < 0 || new\_orient >= 2 \* M\_PI)

            throw std::invalid\_argument("Orientation must be in [0..2pi]");

        x = new\_x;

        y = new\_y;

        orient = new\_orient;

    }

    void set\_noise(double new\_forward\_noise, double new\_turn\_noise, double new\_sense\_noise)

    {

        // Simulate noise, often useful in particle filters

        forward\_noise = new\_forward\_noise;

        turn\_noise = new\_turn\_noise;

        sense\_noise = new\_sense\_noise;

    }

    vector<double> sense()

    {

        // Measure the distances from the robot toward the landmarks

        vector<double> z(sizeof(landmarks) / sizeof(landmarks[0]));

        double dist;

        for (int i = 0; i < sizeof(landmarks) / sizeof(landmarks[0]); i++) {

            dist = sqrt(pow((x - landmarks[i][0]), 2) + pow((y - landmarks[i][1]), 2));

            dist += gen\_gauss\_random(0.0, sense\_noise);

            z[i] = dist;

        }

        return z;

    }

    Robot move(double turn, double forward)

    {

        if (forward < 0)

            throw std::invalid\_argument("Robot cannot move backward");

        // turn, and add randomness to the turning command

        orient = orient + turn + gen\_gauss\_random(0.0, turn\_noise);

        orient = mod(orient, 2 \* M\_PI);

        // move, and add randomness to the motion command

        double dist = forward + gen\_gauss\_random(0.0, forward\_noise);

        x = x + (cos(orient) \* dist);

        y = y + (sin(orient) \* dist);

        // cyclic truncate

        x = mod(x, world\_size);

        y = mod(y, world\_size);

        // set particle

        Robot res;

        res.set(x, y, orient);

        res.set\_noise(forward\_noise, turn\_noise, sense\_noise);

        return res;

    }

    string show\_pose()

    {

        // Returns the robot current position and orientation in a string format

        return "[x=" + to\_string(x) + " y=" + to\_string(y) + " orient=" + to\_string(orient) + "]";

    }

    string read\_sensors()

    {

        // Returns all the distances from the robot toward the landmarks

        vector<double> z = sense();

        string readings = "[";

        for (int i = 0; i < z.size(); i++) {

            readings += to\_string(z[i]) + " ";

        }

        readings[readings.size() - 1] = ']';

        return readings;

    }

    double measurement\_prob(vector<double> measurement)

    {

        // Calculates how likely a measurement should be

        double prob = 1.0;

        double dist;

        for (int i = 0; i < sizeof(landmarks) / sizeof(landmarks[0]); i++) {

            dist = sqrt(pow((x - landmarks[i][0]), 2) + pow((y - landmarks[i][1]), 2));

            prob \*= gaussian(dist, sense\_noise, measurement[i]);

        }

        return prob;

    }

    double x, y, orient; //robot poses

    double forward\_noise, turn\_noise, sense\_noise; //robot noises

private:

    double gen\_gauss\_random(double mean, double variance)

    {

        // Gaussian random

        normal\_distribution<double> gauss\_dist(mean, variance);

        return gauss\_dist(gen);

    }

    double gaussian(double mu, double sigma, double x)

    {

        // Probability of x for 1-dim Gaussian with mean mu and var. sigma

        return exp(-(pow((mu - x), 2)) / (pow(sigma, 2)) / 2.0) / sqrt(2.0 \* M\_PI \* (pow(sigma, 2)));

    }

};

// Functions

double gen\_real\_random()

{

    // Generate real random between 0 and 1

    uniform\_real\_distribution<double> real\_dist(0.0, 1.0); //Real

    return real\_dist(gen);

}

double mod(double first\_term, double second\_term)

{

    // Compute the modulus

    return first\_term - (second\_term)\*floor(first\_term / (second\_term));

}

double evaluation(Robot r, Robot p[], int n)

{

    //Calculate the mean error of the system

    double sum = 0.0;

    for (int i = 0; i < n; i++) {

        //the second part is because of world's cyclicity

        double dx = mod((p[i].x - r.x + (world\_size / 2.0)), world\_size) - (world\_size / 2.0);

        double dy = mod((p[i].y - r.y + (world\_size / 2.0)), world\_size) - (world\_size / 2.0);

        double err = sqrt(pow(dx, 2) + pow(dy, 2));

        sum += err;

    }

    return sum / n;

}

double max(double arr[], int n)

{

    // Identify the max element in an array

    double max = 0;

    for (int i = 0; i < n; i++) {

        if (arr[i] > max)

            max = arr[i];

    }

    return max;

}

/\*

void visualization(int n, Robot robot, int step, Robot p[], Robot pr[])

{

    //Draw the robot, landmarks, particles and resampled particles on a graph

    //Graph Format

    plt::title("MCL, step " + to\_string(step));

    plt::xlim(0, 100);

    plt::ylim(0, 100);

    //Draw particles in green

    for (int i = 0; i < n; i++) {

        plt::plot({ p[i].x }, { p[i].y }, "go");

    }

    //Draw resampled particles in yellow

    for (int i = 0; i < n; i++) {

        plt::plot({ pr[i].x }, { pr[i].y }, "yo");

    }

    //Draw landmarks in red

    for (int i = 0; i < sizeof(landmarks) / sizeof(landmarks[0]); i++) {

        plt::plot({ landmarks[i][0] }, { landmarks[i][1] }, "ro");

    }

    //Draw robot position in blue

    plt::plot({ robot.x }, { robot.y }, "bo");

    //Save the image and close the plot

    plt::save("./Images/Step" + to\_string(step) + ".png");

    plt::clf();

}

\*/

//####   DON'T MODIFY ANYTHING ABOVE HERE! ENTER CODE BELOW ####

int main()

{

    // TODO: Print "I am ready for coding the MCL!"

    return 0;

}

### First Interaction

Video 5.3

A screenshot of a computer

AI-generated content may be incorrect.

#### main.cpp

//#include "src/matplotlibcpp.h"//Graph Library

#include <iostream>

#include <string>

#include <math.h>

#include <vector>

#include <stdexcept> // throw errors

#include <random> //C++ 11 Random Numbers

//namespace plt = matplotlibcpp;

using namespace std;

// Landmarks

double landmarks[8][2] = { { 20.0, 20.0 }, { 20.0, 80.0 }, { 20.0, 50.0 },

    { 50.0, 20.0 }, { 50.0, 80.0 }, { 80.0, 80.0 },

    { 80.0, 20.0 }, { 80.0, 50.0 } };

// Map size in meters

double world\_size = 100.0;

// Random Generators

random\_device rd;

mt19937 gen(rd());

// Global Functions

double mod(double first\_term, double second\_term);

double gen\_real\_random();

class Robot {

public:

    Robot()

    {

        // Constructor

        x = gen\_real\_random() \* world\_size; // robot's x coordinate

        y = gen\_real\_random() \* world\_size; // robot's y coordinate

        orient = gen\_real\_random() \* 2.0 \* M\_PI; // robot's orientation

        forward\_noise = 0.0; //noise of the forward movement

        turn\_noise = 0.0; //noise of the turn

        sense\_noise = 0.0; //noise of the sensing

    }

    void set(double new\_x, double new\_y, double new\_orient)

    {

        // Set robot new position and orientation

        if (new\_x < 0 || new\_x >= world\_size)

            throw std::invalid\_argument("X coordinate out of bound");

        if (new\_y < 0 || new\_y >= world\_size)

            throw std::invalid\_argument("Y coordinate out of bound");

        if (new\_orient < 0 || new\_orient >= 2 \* M\_PI)

            throw std::invalid\_argument("Orientation must be in [0..2pi]");

        x = new\_x;

        y = new\_y;

        orient = new\_orient;

    }

    void set\_noise(double new\_forward\_noise, double new\_turn\_noise, double new\_sense\_noise)

    {

        // Simulate noise, often useful in particle filters

        forward\_noise = new\_forward\_noise;

        turn\_noise = new\_turn\_noise;

        sense\_noise = new\_sense\_noise;

    }

    vector<double> sense()

    {

        // Measure the distances from the robot toward the landmarks

        vector<double> z(sizeof(landmarks) / sizeof(landmarks[0]));

        double dist;

        for (int i = 0; i < sizeof(landmarks) / sizeof(landmarks[0]); i++) {

            dist = sqrt(pow((x - landmarks[i][0]), 2) + pow((y - landmarks[i][1]), 2));

            dist += gen\_gauss\_random(0.0, sense\_noise);

            z[i] = dist;

        }

        return z;

    }

    Robot move(double turn, double forward)

    {

        if (forward < 0)

            throw std::invalid\_argument("Robot cannot move backward");

        // turn, and add randomness to the turning command

        orient = orient + turn + gen\_gauss\_random(0.0, turn\_noise);

        orient = mod(orient, 2 \* M\_PI);

        // move, and add randomness to the motion command

        double dist = forward + gen\_gauss\_random(0.0, forward\_noise);

        x = x + (cos(orient) \* dist);

        y = y + (sin(orient) \* dist);

        // cyclic truncate

        x = mod(x, world\_size);

        y = mod(y, world\_size);

        // set particle

        Robot res;

        res.set(x, y, orient);

        res.set\_noise(forward\_noise, turn\_noise, sense\_noise);

        return res;

    }

    string show\_pose()

    {

        // Returns the robot current position and orientation in a string format

        return "[x=" + to\_string(x) + " y=" + to\_string(y) + " orient=" + to\_string(orient) + "]";

    }

    string read\_sensors()

    {

        // Returns all the distances from the robot toward the landmarks

        vector<double> z = sense();

        string readings = "[";

        for (int i = 0; i < z.size(); i++) {

            readings += to\_string(z[i]) + " ";

        }

        readings[readings.size() - 1] = ']';

        return readings;

    }

    double measurement\_prob(vector<double> measurement)

    {

        // Calculates how likely a measurement should be

        double prob = 1.0;

        double dist;

        for (int i = 0; i < sizeof(landmarks) / sizeof(landmarks[0]); i++) {

            dist = sqrt(pow((x - landmarks[i][0]), 2) + pow((y - landmarks[i][1]), 2));

            prob \*= gaussian(dist, sense\_noise, measurement[i]);

        }

        return prob;

    }

    double x, y, orient; //robot poses

    double forward\_noise, turn\_noise, sense\_noise; //robot noises

private:

    double gen\_gauss\_random(double mean, double variance)

    {

        // Gaussian random

        normal\_distribution<double> gauss\_dist(mean, variance);

        return gauss\_dist(gen);

    }

    double gaussian(double mu, double sigma, double x)

    {

        // Probability of x for 1-dim Gaussian with mean mu and var. sigma

        return exp(-(pow((mu - x), 2)) / (pow(sigma, 2)) / 2.0) / sqrt(2.0 \* M\_PI \* (pow(sigma, 2)));

    }

};

// Functions

double gen\_real\_random()

{

    // Generate real random between 0 and 1

    uniform\_real\_distribution<double> real\_dist(0.0, 1.0); //Real

    return real\_dist(gen);

}

double mod(double first\_term, double second\_term)

{

    // Compute the modulus

    return first\_term - (second\_term)\*floor(first\_term / (second\_term));

}

double evaluation(Robot r, Robot p[], int n)

{

    //Calculate the mean error of the system

    double sum = 0.0;

    for (int i = 0; i < n; i++) {

        //the second part is because of world's cyclicity

        double dx = mod((p[i].x - r.x + (world\_size / 2.0)), world\_size) - (world\_size / 2.0);

        double dy = mod((p[i].y - r.y + (world\_size / 2.0)), world\_size) - (world\_size / 2.0);

        double err = sqrt(pow(dx, 2) + pow(dy, 2));

        sum += err;

    }

    return sum / n;

}

double max(double arr[], int n)

{

    // Identify the max element in an array

    double max = 0;

    for (int i = 0; i < n; i++) {

        if (arr[i] > max)

            max = arr[i];

    }

    return max;

}

/\*

void visualization(int n, Robot robot, int step, Robot p[], Robot pr[])

{

    //Draw the robot, landmarks, particles and resampled particles on a graph

    //Graph Format

    plt::title("MCL, step " + to\_string(step));

    plt::xlim(0, 100);

    plt::ylim(0, 100);

    //Draw particles in green

    for (int i = 0; i < n; i++) {

        plt::plot({ p[i].x }, { p[i].y }, "go");

    }

    //Draw resampled particles in yellow

    for (int i = 0; i < n; i++) {

        plt::plot({ pr[i].x }, { pr[i].y }, "yo");

    }

    //Draw landmarks in red

    for (int i = 0; i < sizeof(landmarks) / sizeof(landmarks[0]); i++) {

        plt::plot({ landmarks[i][0] }, { landmarks[i][1] }, "ro");

    }

    //Draw robot position in blue

    plt::plot({ robot.x }, { robot.y }, "bo");

    //Save the image and close the plot

    plt::save("./Images/Step" + to\_string(step) + ".png");

    plt::clf();

}

\*/

//####   DON'T MODIFY ANYTHING ABOVE HERE! ENTER CODE BELOW ####

int main()

{

    // Instantiating a robot object from the Robot class

    Robot myrobot;

    // TODO: Set robot new position to x=10.0, y=10.0 and orientation=0

    // Fill in the position and orientation values in myrobot.set() function

    myrobot.set();

    // Printing out the new robot position and orientation

    cout << myrobot.show\_pose() << endl;

    // TODO: Rotate the robot by PI/2.0 and then move him forward by 10.0

    // Use M\_PI for the pi value

    myrobot.move();

    // TODO: Print out the new robot position and orientation

    // Printing the distance from the robot toward the eight landmarks

    cout << myrobot.read\_sensors() << endl;

    return 0;

}

#### solution.cpp

//#include "src/matplotlibcpp.h"//Graph Library

#include <iostream>

#include <string>

#include <math.h>

#include <vector>

#include <stdexcept> // throw errors

#include <random> //C++ 11 Random Numbers

//namespace plt = matplotlibcpp;

using namespace std;

// Landmarks

double landmarks[8][2] = { { 20.0, 20.0 }, { 20.0, 80.0 }, { 20.0, 50.0 },

    { 50.0, 20.0 }, { 50.0, 80.0 }, { 80.0, 80.0 },

    { 80.0, 20.0 }, { 80.0, 50.0 } };

// Map size in meters

double world\_size = 100.0;

// Random Generators

random\_device rd;

mt19937 gen(rd());

// Global Functions

double mod(double first\_term, double second\_term);

double gen\_real\_random();

class Robot {

public:

    Robot()

    {

        // Constructor

        x = gen\_real\_random() \* world\_size; // robot's x coordinate

        y = gen\_real\_random() \* world\_size; // robot's y coordinate

        orient = gen\_real\_random() \* 2.0 \* M\_PI; // robot's orientation

        forward\_noise = 0.0; //noise of the forward movement

        turn\_noise = 0.0; //noise of the turn

        sense\_noise = 0.0; //noise of the sensing

    }

    void set(double new\_x, double new\_y, double new\_orient)

    {

        // Set robot new position and orientation

        if (new\_x < 0 || new\_x >= world\_size)

            throw std::invalid\_argument("X coordinate out of bound");

        if (new\_y < 0 || new\_y >= world\_size)

            throw std::invalid\_argument("Y coordinate out of bound");

        if (new\_orient < 0 || new\_orient >= 2 \* M\_PI)

            throw std::invalid\_argument("Orientation must be in [0..2pi]");

        x = new\_x;

        y = new\_y;

        orient = new\_orient;

    }

    void set\_noise(double new\_forward\_noise, double new\_turn\_noise, double new\_sense\_noise)

    {

        // Simulate noise, often useful in particle filters

        forward\_noise = new\_forward\_noise;

        turn\_noise = new\_turn\_noise;

        sense\_noise = new\_sense\_noise;

    }

    vector<double> sense()

    {

        // Measure the distances from the robot toward the landmarks

        vector<double> z(sizeof(landmarks) / sizeof(landmarks[0]));

        double dist;

        for (int i = 0; i < sizeof(landmarks) / sizeof(landmarks[0]); i++) {

            dist = sqrt(pow((x - landmarks[i][0]), 2) + pow((y - landmarks[i][1]), 2));

            dist += gen\_gauss\_random(0.0, sense\_noise);

            z[i] = dist;

        }

        return z;

    }

    Robot move(double turn, double forward)

    {

        if (forward < 0)

            throw std::invalid\_argument("Robot cannot move backward");

        // turn, and add randomness to the turning command

        orient = orient + turn + gen\_gauss\_random(0.0, turn\_noise);

        orient = mod(orient, 2 \* M\_PI);

        // move, and add randomness to the motion command

        double dist = forward + gen\_gauss\_random(0.0, forward\_noise);

        x = x + (cos(orient) \* dist);

        y = y + (sin(orient) \* dist);

        // cyclic truncate

        x = mod(x, world\_size);

        y = mod(y, world\_size);

        // set particle

        Robot res;

        res.set(x, y, orient);

        res.set\_noise(forward\_noise, turn\_noise, sense\_noise);

        return res;

    }

    string show\_pose()

    {

        // Returns the robot current position and orientation in a string format

        return "[x=" + to\_string(x) + " y=" + to\_string(y) + " orient=" + to\_string(orient) + "]";

    }

    string read\_sensors()

    {

        // Returns all the distances from the robot toward the landmarks

        vector<double> z = sense();

        string readings = "[";

        for (int i = 0; i < z.size(); i++) {

            readings += to\_string(z[i]) + " ";

        }

        readings[readings.size() - 1] = ']';

        return readings;

    }

    double measurement\_prob(vector<double> measurement)

    {

        // Calculates how likely a measurement should be

        double prob = 1.0;

        double dist;

        for (int i = 0; i < sizeof(landmarks) / sizeof(landmarks[0]); i++) {

            dist = sqrt(pow((x - landmarks[i][0]), 2) + pow((y - landmarks[i][1]), 2));

            prob \*= gaussian(dist, sense\_noise, measurement[i]);

        }

        return prob;

    }

    double x, y, orient; //robot poses

    double forward\_noise, turn\_noise, sense\_noise; //robot noises

private:

    double gen\_gauss\_random(double mean, double variance)

    {

        // Gaussian random

        normal\_distribution<double> gauss\_dist(mean, variance);

        return gauss\_dist(gen);

    }

    double gaussian(double mu, double sigma, double x)

    {

        // Probability of x for 1-dim Gaussian with mean mu and var. sigma

        return exp(-(pow((mu - x), 2)) / (pow(sigma, 2)) / 2.0) / sqrt(2.0 \* M\_PI \* (pow(sigma, 2)));

    }

};

// Functions

double gen\_real\_random()

{

    // Generate real random between 0 and 1

    uniform\_real\_distribution<double> real\_dist(0.0, 1.0); //Real

    return real\_dist(gen);

}

double mod(double first\_term, double second\_term)

{

    // Compute the modulus

    return first\_term - (second\_term)\*floor(first\_term / (second\_term));

}

double evaluation(Robot r, Robot p[], int n)

{

    //Calculate the mean error of the system

    double sum = 0.0;

    for (int i = 0; i < n; i++) {

        //the second part is because of world's cyclicity

        double dx = mod((p[i].x - r.x + (world\_size / 2.0)), world\_size) - (world\_size / 2.0);

        double dy = mod((p[i].y - r.y + (world\_size / 2.0)), world\_size) - (world\_size / 2.0);

        double err = sqrt(pow(dx, 2) + pow(dy, 2));

        sum += err;

    }

    return sum / n;

}

double max(double arr[], int n)

{

    // Identify the max element in an array

    double max = 0;

    for (int i = 0; i < n; i++) {

        if (arr[i] > max)

            max = arr[i];

    }

    return max;

}

/\*

void visualization(int n, Robot robot, int step, Robot p[], Robot pr[])

{

    //Draw the robot, landmarks, particles and resampled particles on a graph

    //Graph Format

    plt::title("MCL, step " + to\_string(step));

    plt::xlim(0, 100);

    plt::ylim(0, 100);

    //Draw particles in green

    for (int i = 0; i < n; i++) {

        plt::plot({ p[i].x }, { p[i].y }, "go");

    }

    //Draw resampled particles in yellow

    for (int i = 0; i < n; i++) {

        plt::plot({ pr[i].x }, { pr[i].y }, "yo");

    }

    //Draw landmarks in red

    for (int i = 0; i < sizeof(landmarks) / sizeof(landmarks[0]); i++) {

        plt::plot({ landmarks[i][0] }, { landmarks[i][1] }, "ro");

    }

    //Draw robot position in blue

    plt::plot({ robot.x }, { robot.y }, "bo");

    //Save the image and close the plot

    plt::save("./Images/Step" + to\_string(step) + ".png");

    plt::clf();

}

\*/

//####   DON'T MODIFY ANYTHING ABOVE HERE! ENTER CODE BELOW ####

int main()

{

    // Instantiating a robot object from the Robot class

    Robot myrobot;

    // Set robot new position to x=10.0, y=10.0 and orientation=0

    // Fill in the position and orientation values in myrobot.set() function

    myrobot.set(10.0, 10.0, 0);

    // Printing out the new robot position and orientation

    cout << myrobot.show\_pose() << endl;

    // Rotate the robot by PI/2.0 and then move him forward by 10.0

    // Use M\_PI for the pi value

    myrobot.move(M\_PI / 2.0, 10.0);

    // Print out the new robot position and orientation

    cout << myrobot.show\_pose() << endl;

    // Printing the distance from the robot toward the eight landmarks

    cout << myrobot.read\_sensors() << endl;

    return 0;

}

### Motion and Sensing

A screenshot of a computer

AI-generated content may be incorrect.

#### main.cpp

//#include "src/matplotlibcpp.h"//Graph Library

#include <iostream>

#include <string>

#include <math.h>

#include <vector>

#include <stdexcept> // throw errors

#include <random> //C++ 11 Random Numbers

//namespace plt = matplotlibcpp;

using namespace std;

// Landmarks

double landmarks[8][2] = { { 20.0, 20.0 }, { 20.0, 80.0 }, { 20.0, 50.0 },

    { 50.0, 20.0 }, { 50.0, 80.0 }, { 80.0, 80.0 },

    { 80.0, 20.0 }, { 80.0, 50.0 } };

// Map size in meters

double world\_size = 100.0;

// Random Generators

random\_device rd;

mt19937 gen(rd());

// Global Functions

double mod(double first\_term, double second\_term);

double gen\_real\_random();

class Robot {

public:

    Robot()

    {

        // Constructor

        x = gen\_real\_random() \* world\_size; // robot's x coordinate

        y = gen\_real\_random() \* world\_size; // robot's y coordinate

        orient = gen\_real\_random() \* 2.0 \* M\_PI; // robot's orientation

        forward\_noise = 0.0; //noise of the forward movement

        turn\_noise = 0.0; //noise of the turn

        sense\_noise = 0.0; //noise of the sensing

    }

    void set(double new\_x, double new\_y, double new\_orient)

    {

        // Set robot new position and orientation

        if (new\_x < 0 || new\_x >= world\_size)

            throw std::invalid\_argument("X coordinate out of bound");

        if (new\_y < 0 || new\_y >= world\_size)

            throw std::invalid\_argument("Y coordinate out of bound");

        if (new\_orient < 0 || new\_orient >= 2 \* M\_PI)

            throw std::invalid\_argument("Orientation must be in [0..2pi]");

        x = new\_x;

        y = new\_y;

        orient = new\_orient;

    }

    void set\_noise(double new\_forward\_noise, double new\_turn\_noise, double new\_sense\_noise)

    {

        // Simulate noise, often useful in particle filters

        forward\_noise = new\_forward\_noise;

        turn\_noise = new\_turn\_noise;

        sense\_noise = new\_sense\_noise;

    }

    vector<double> sense()

    {

        // Measure the distances from the robot toward the landmarks

        vector<double> z(sizeof(landmarks) / sizeof(landmarks[0]));

        double dist;

        for (int i = 0; i < sizeof(landmarks) / sizeof(landmarks[0]); i++) {

            dist = sqrt(pow((x - landmarks[i][0]), 2) + pow((y - landmarks[i][1]), 2));

            dist += gen\_gauss\_random(0.0, sense\_noise);

            z[i] = dist;

        }

        return z;

    }

    Robot move(double turn, double forward)

    {

        if (forward < 0)

            throw std::invalid\_argument("Robot cannot move backward");

        // turn, and add randomness to the turning command

        orient = orient + turn + gen\_gauss\_random(0.0, turn\_noise);

        orient = mod(orient, 2 \* M\_PI);

        // move, and add randomness to the motion command

        double dist = forward + gen\_gauss\_random(0.0, forward\_noise);

        x = x + (cos(orient) \* dist);

        y = y + (sin(orient) \* dist);

        // cyclic truncate

        x = mod(x, world\_size);

        y = mod(y, world\_size);

        // set particle

        Robot res;

        res.set(x, y, orient);

        res.set\_noise(forward\_noise, turn\_noise, sense\_noise);

        return res;

    }

    string show\_pose()

    {

        // Returns the robot current position and orientation in a string format

        return "[x=" + to\_string(x) + " y=" + to\_string(y) + " orient=" + to\_string(orient) + "]";

    }

    string read\_sensors()

    {

        // Returns all the distances from the robot toward the landmarks

        vector<double> z = sense();

        string readings = "[";

        for (int i = 0; i < z.size(); i++) {

            readings += to\_string(z[i]) + " ";

        }

        readings[readings.size() - 1] = ']';

        return readings;

    }

    double measurement\_prob(vector<double> measurement)

    {

        // Calculates how likely a measurement should be

        double prob = 1.0;

        double dist;

        for (int i = 0; i < sizeof(landmarks) / sizeof(landmarks[0]); i++) {

            dist = sqrt(pow((x - landmarks[i][0]), 2) + pow((y - landmarks[i][1]), 2));

            prob \*= gaussian(dist, sense\_noise, measurement[i]);

        }

        return prob;

    }

    double x, y, orient; //robot poses

    double forward\_noise, turn\_noise, sense\_noise; //robot noises

private:

    double gen\_gauss\_random(double mean, double variance)

    {

        // Gaussian random

        normal\_distribution<double> gauss\_dist(mean, variance);

        return gauss\_dist(gen);

    }

    double gaussian(double mu, double sigma, double x)

    {

        // Probability of x for 1-dim Gaussian with mean mu and var. sigma

        return exp(-(pow((mu - x), 2)) / (pow(sigma, 2)) / 2.0) / sqrt(2.0 \* M\_PI \* (pow(sigma, 2)));

    }

};

// Functions

double gen\_real\_random()

{

    // Generate real random between 0 and 1

    uniform\_real\_distribution<double> real\_dist(0.0, 1.0); //Real

    return real\_dist(gen);

}

double mod(double first\_term, double second\_term)

{

    // Compute the modulus

    return first\_term - (second\_term)\*floor(first\_term / (second\_term));

}

double evaluation(Robot r, Robot p[], int n)

{

    //Calculate the mean error of the system

    double sum = 0.0;

    for (int i = 0; i < n; i++) {

        //the second part is because of world's cyclicity

        double dx = mod((p[i].x - r.x + (world\_size / 2.0)), world\_size) - (world\_size / 2.0);

        double dy = mod((p[i].y - r.y + (world\_size / 2.0)), world\_size) - (world\_size / 2.0);

        double err = sqrt(pow(dx, 2) + pow(dy, 2));

        sum += err;

    }

    return sum / n;

}

double max(double arr[], int n)

{

    // Identify the max element in an array

    double max = 0;

    for (int i = 0; i < n; i++) {

        if (arr[i] > max)

            max = arr[i];

    }

    return max;

}

/\*

void visualization(int n, Robot robot, int step, Robot p[], Robot pr[])

{

    //Draw the robot, landmarks, particles and resampled particles on a graph

    //Graph Format

    plt::title("MCL, step " + to\_string(step));

    plt::xlim(0, 100);

    plt::ylim(0, 100);

    //Draw particles in green

    for (int i = 0; i < n; i++) {

        plt::plot({ p[i].x }, { p[i].y }, "go");

    }

    //Draw resampled particles in yellow

    for (int i = 0; i < n; i++) {

        plt::plot({ pr[i].x }, { pr[i].y }, "yo");

    }

    //Draw landmarks in red

    for (int i = 0; i < sizeof(landmarks) / sizeof(landmarks[0]); i++) {

        plt::plot({ landmarks[i][0] }, { landmarks[i][1] }, "ro");

    }

    //Draw robot position in blue

    plt::plot({ robot.x }, { robot.y }, "bo");

    //Save the image and close the plot

    plt::save("./Images/Step" + to\_string(step) + ".png");

    plt::clf();

}

\*/

//####   DON'T MODIFY ANYTHING ABOVE HERE! ENTER CODE BELOW ####

int main()

{

    // TODO: Instantiate a robot object from the Robot class

    // TODO: Set robot new position to x=30.0, y=50.0 and orientation=PI/2

    // TODO: Turn clockwise by PI/2 and move by 15 meters

    // TODO: Print the distance from the robot toward the eight landmarks

    // TODO: Turn clockwise by PI/2 and move by 10 meters

    // TODO: Print the distance from the robot toward the eight landmarks

    return 0;

}

#### solution.cpp

//#include "src/matplotlibcpp.h"//Graph Library

#include <iostream>

#include <string>

#include <math.h>

#include <vector>

#include <stdexcept> // throw errors

#include <random> //C++ 11 Random Numbers

//namespace plt = matplotlibcpp;

using namespace std;

// Landmarks

double landmarks[8][2] = { { 20.0, 20.0 }, { 20.0, 80.0 }, { 20.0, 50.0 },

    { 50.0, 20.0 }, { 50.0, 80.0 }, { 80.0, 80.0 },

    { 80.0, 20.0 }, { 80.0, 50.0 } };

// Map size in meters

double world\_size = 100.0;

// Random Generators

random\_device rd;

mt19937 gen(rd());

// Global Functions

double mod(double first\_term, double second\_term);

double gen\_real\_random();

class Robot {

public:

    Robot()

    {

        // Constructor

        x = gen\_real\_random() \* world\_size; // robot's x coordinate

        y = gen\_real\_random() \* world\_size; // robot's y coordinate

        orient = gen\_real\_random() \* 2.0 \* M\_PI; // robot's orientation

        forward\_noise = 0.0; //noise of the forward movement

        turn\_noise = 0.0; //noise of the turn

        sense\_noise = 0.0; //noise of the sensing

    }

    void set(double new\_x, double new\_y, double new\_orient)

    {

        // Set robot new position and orientation

        if (new\_x < 0 || new\_x >= world\_size)

            throw std::invalid\_argument("X coordinate out of bound");

        if (new\_y < 0 || new\_y >= world\_size)

            throw std::invalid\_argument("Y coordinate out of bound");

        if (new\_orient < 0 || new\_orient >= 2 \* M\_PI)

            throw std::invalid\_argument("Orientation must be in [0..2pi]");

        x = new\_x;

        y = new\_y;

        orient = new\_orient;

    }

    void set\_noise(double new\_forward\_noise, double new\_turn\_noise, double new\_sense\_noise)

    {

        // Simulate noise, often useful in particle filters

        forward\_noise = new\_forward\_noise;

        turn\_noise = new\_turn\_noise;

        sense\_noise = new\_sense\_noise;

    }

    vector<double> sense()

    {

        // Measure the distances from the robot toward the landmarks

        vector<double> z(sizeof(landmarks) / sizeof(landmarks[0]));

        double dist;

        for (int i = 0; i < sizeof(landmarks) / sizeof(landmarks[0]); i++) {

            dist = sqrt(pow((x - landmarks[i][0]), 2) + pow((y - landmarks[i][1]), 2));

            dist += gen\_gauss\_random(0.0, sense\_noise);

            z[i] = dist;

        }

        return z;

    }

    Robot move(double turn, double forward)

    {

        if (forward < 0)

            throw std::invalid\_argument("Robot cannot move backward");

        // turn, and add randomness to the turning command

        orient = orient + turn + gen\_gauss\_random(0.0, turn\_noise);

        orient = mod(orient, 2 \* M\_PI);

        // move, and add randomness to the motion command

        double dist = forward + gen\_gauss\_random(0.0, forward\_noise);

        x = x + (cos(orient) \* dist);

        y = y + (sin(orient) \* dist);

        // cyclic truncate

        x = mod(x, world\_size);

        y = mod(y, world\_size);

        // set particle

        Robot res;

        res.set(x, y, orient);

        res.set\_noise(forward\_noise, turn\_noise, sense\_noise);

        return res;

    }

    string show\_pose()

    {

        // Returns the robot current position and orientation in a string format

        return "[x=" + to\_string(x) + " y=" + to\_string(y) + " orient=" + to\_string(orient) + "]";

    }

    string read\_sensors()

    {

        // Returns all the distances from the robot toward the landmarks

        vector<double> z = sense();

        string readings = "[";

        for (int i = 0; i < z.size(); i++) {

            readings += to\_string(z[i]) + " ";

        }

        readings[readings.size() - 1] = ']';

        return readings;

    }

    double measurement\_prob(vector<double> measurement)

    {

        // Calculates how likely a measurement should be

        double prob = 1.0;

        double dist;

        for (int i = 0; i < sizeof(landmarks) / sizeof(landmarks[0]); i++) {

            dist = sqrt(pow((x - landmarks[i][0]), 2) + pow((y - landmarks[i][1]), 2));

            prob \*= gaussian(dist, sense\_noise, measurement[i]);

        }

        return prob;

    }

    double x, y, orient; //robot poses

    double forward\_noise, turn\_noise, sense\_noise; //robot noises

private:

    double gen\_gauss\_random(double mean, double variance)

    {

        // Gaussian random

        normal\_distribution<double> gauss\_dist(mean, variance);

        return gauss\_dist(gen);

    }

    double gaussian(double mu, double sigma, double x)

    {

        // Probability of x for 1-dim Gaussian with mean mu and var. sigma

        return exp(-(pow((mu - x), 2)) / (pow(sigma, 2)) / 2.0) / sqrt(2.0 \* M\_PI \* (pow(sigma, 2)));

    }

};

// Functions

double gen\_real\_random()

{

    // Generate real random between 0 and 1

    uniform\_real\_distribution<double> real\_dist(0.0, 1.0); //Real

    return real\_dist(gen);

}

double mod(double first\_term, double second\_term)

{

    // Compute the modulus

    return first\_term - (second\_term)\*floor(first\_term / (second\_term));

}

double evaluation(Robot r, Robot p[], int n)

{

    //Calculate the mean error of the system

    double sum = 0.0;

    for (int i = 0; i < n; i++) {

        //the second part is because of world's cyclicity

        double dx = mod((p[i].x - r.x + (world\_size / 2.0)), world\_size) - (world\_size / 2.0);

        double dy = mod((p[i].y - r.y + (world\_size / 2.0)), world\_size) - (world\_size / 2.0);

        double err = sqrt(pow(dx, 2) + pow(dy, 2));

        sum += err;

    }

    return sum / n;

}

double max(double arr[], int n)

{

    // Identify the max element in an array

    double max = 0;

    for (int i = 0; i < n; i++) {

        if (arr[i] > max)

            max = arr[i];

    }

    return max;

}

/\*

void visualization(int n, Robot robot, int step, Robot p[], Robot pr[])

{

    //Draw the robot, landmarks, particles and resampled particles on a graph

    //Graph Format

    plt::title("MCL, step " + to\_string(step));

    plt::xlim(0, 100);

    plt::ylim(0, 100);

    //Draw particles in green

    for (int i = 0; i < n; i++) {

        plt::plot({ p[i].x }, { p[i].y }, "go");

    }

    //Draw resampled particles in yellow

    for (int i = 0; i < n; i++) {

        plt::plot({ pr[i].x }, { pr[i].y }, "yo");

    }

    //Draw landmarks in red

    for (int i = 0; i < sizeof(landmarks) / sizeof(landmarks[0]); i++) {

        plt::plot({ landmarks[i][0] }, { landmarks[i][1] }, "ro");

    }

    //Draw robot position in blue

    plt::plot({ robot.x }, { robot.y }, "bo");

    //Save the image and close the plot

    plt::save("./Images/Step" + to\_string(step) + ".png");

    plt::clf();

}

\*/

//####   DON'T MODIFY ANYTHING ABOVE HERE! ENTER CODE BELOW ####

int main()

{

    // Instantiate a robot object from the Robot class

    Robot myrobot;

    // Set robot new position to x=30.0, y=50.0 and orientation=PI/2.0

    myrobot.set(30.0, 50.0, M\_PI / 2.0);

    // Turn clockwise by PI/2.0 and move by 15.0 meters

    myrobot.move(-M\_PI / 2.0, 15.0);

    // Print the distance from the robot toward the eight landmarks

    cout << myrobot.read\_sensors() << endl;

    // Turn clockwise by PI/2.0 and move by 10.0 meters

    myrobot.move(-M\_PI / 2.0, 10.0);

    // Print the distance from the robot toward the eight landmarks

    cout << myrobot.read\_sensors() << endl;

    return 0;

}

### Noise

Video 5.5

**Noise**

You’ll now alter the robot’s **pose** and **measurement** values to **noisy** ones. Add the following noise values:

* Forward Noise=5.0
* Turn Noise=0.1
* Sense Noise=5.0.

Scroll down to the main function and follow the instructions.

Since you are working with random values, this program will output different numbers each time you run it!

#### main.cpp

//#include "src/matplotlibcpp.h"//Graph Library

#include <iostream>

#include <string>

#include <math.h>

#include <vector>

#include <stdexcept> // throw errors

#include <random> //C++ 11 Random Numbers

//namespace plt = matplotlibcpp;

using namespace std;

// Landmarks

double landmarks[8][2] = { { 20.0, 20.0 }, { 20.0, 80.0 }, { 20.0, 50.0 },

    { 50.0, 20.0 }, { 50.0, 80.0 }, { 80.0, 80.0 },

    { 80.0, 20.0 }, { 80.0, 50.0 } };

// Map size in meters

double world\_size = 100.0;

// Random Generators

random\_device rd;

mt19937 gen(rd());

// Global Functions

double mod(double first\_term, double second\_term);

double gen\_real\_random();

class Robot {

public:

    Robot()

    {

        // Constructor

        x = gen\_real\_random() \* world\_size; // robot's x coordinate

        y = gen\_real\_random() \* world\_size; // robot's y coordinate

        orient = gen\_real\_random() \* 2.0 \* M\_PI; // robot's orientation

        forward\_noise = 0.0; //noise of the forward movement

        turn\_noise = 0.0; //noise of the turn

        sense\_noise = 0.0; //noise of the sensing

    }

    void set(double new\_x, double new\_y, double new\_orient)

    {

        // Set robot new position and orientation

        if (new\_x < 0 || new\_x >= world\_size)

            throw std::invalid\_argument("X coordinate out of bound");

        if (new\_y < 0 || new\_y >= world\_size)

            throw std::invalid\_argument("Y coordinate out of bound");

        if (new\_orient < 0 || new\_orient >= 2 \* M\_PI)

            throw std::invalid\_argument("Orientation must be in [0..2pi]");

        x = new\_x;

        y = new\_y;

        orient = new\_orient;

    }

    void set\_noise(double new\_forward\_noise, double new\_turn\_noise, double new\_sense\_noise)

    {

        // Simulate noise, often useful in particle filters

        forward\_noise = new\_forward\_noise;

        turn\_noise = new\_turn\_noise;

        sense\_noise = new\_sense\_noise;

    }

    vector<double> sense()

    {

        // Measure the distances from the robot toward the landmarks

        vector<double> z(sizeof(landmarks) / sizeof(landmarks[0]));

        double dist;

        for (int i = 0; i < sizeof(landmarks) / sizeof(landmarks[0]); i++) {

            dist = sqrt(pow((x - landmarks[i][0]), 2) + pow((y - landmarks[i][1]), 2));

            dist += gen\_gauss\_random(0.0, sense\_noise);

            z[i] = dist;

        }

        return z;

    }

    Robot move(double turn, double forward)

    {

        if (forward < 0)

            throw std::invalid\_argument("Robot cannot move backward");

        // turn, and add randomness to the turning command

        orient = orient + turn + gen\_gauss\_random(0.0, turn\_noise);

        orient = mod(orient, 2 \* M\_PI);

        // move, and add randomness to the motion command

        double dist = forward + gen\_gauss\_random(0.0, forward\_noise);

        x = x + (cos(orient) \* dist);

        y = y + (sin(orient) \* dist);

        // cyclic truncate

        x = mod(x, world\_size);

        y = mod(y, world\_size);

        // set particle

        Robot res;

        res.set(x, y, orient);

        res.set\_noise(forward\_noise, turn\_noise, sense\_noise);

        return res;

    }

    string show\_pose()

    {

        // Returns the robot current position and orientation in a string format

        return "[x=" + to\_string(x) + " y=" + to\_string(y) + " orient=" + to\_string(orient) + "]";

    }

    string read\_sensors()

    {

        // Returns all the distances from the robot toward the landmarks

        vector<double> z = sense();

        string readings = "[";

        for (int i = 0; i < z.size(); i++) {

            readings += to\_string(z[i]) + " ";

        }

        readings[readings.size() - 1] = ']';

        return readings;

    }

    double measurement\_prob(vector<double> measurement)

    {

        // Calculates how likely a measurement should be

        double prob = 1.0;

        double dist;

        for (int i = 0; i < sizeof(landmarks) / sizeof(landmarks[0]); i++) {

            dist = sqrt(pow((x - landmarks[i][0]), 2) + pow((y - landmarks[i][1]), 2));

            prob \*= gaussian(dist, sense\_noise, measurement[i]);

        }

        return prob;

    }

    double x, y, orient; //robot poses

    double forward\_noise, turn\_noise, sense\_noise; //robot noises

private:

    double gen\_gauss\_random(double mean, double variance)

    {

        // Gaussian random

        normal\_distribution<double> gauss\_dist(mean, variance);

        return gauss\_dist(gen);

    }

    double gaussian(double mu, double sigma, double x)

    {

        // Probability of x for 1-dim Gaussian with mean mu and var. sigma

        return exp(-(pow((mu - x), 2)) / (pow(sigma, 2)) / 2.0) / sqrt(2.0 \* M\_PI \* (pow(sigma, 2)));

    }

};

// Functions

double gen\_real\_random()

{

    // Generate real random between 0 and 1

    uniform\_real\_distribution<double> real\_dist(0.0, 1.0); //Real

    return real\_dist(gen);

}

double mod(double first\_term, double second\_term)

{

    // Compute the modulus

    return first\_term - (second\_term)\*floor(first\_term / (second\_term));

}

double evaluation(Robot r, Robot p[], int n)

{

    //Calculate the mean error of the system

    double sum = 0.0;

    for (int i = 0; i < n; i++) {

        //the second part is because of world's cyclicity

        double dx = mod((p[i].x - r.x + (world\_size / 2.0)), world\_size) - (world\_size / 2.0);

        double dy = mod((p[i].y - r.y + (world\_size / 2.0)), world\_size) - (world\_size / 2.0);

        double err = sqrt(pow(dx, 2) + pow(dy, 2));

        sum += err;

    }

    return sum / n;

}

double max(double arr[], int n)

{

    // Identify the max element in an array

    double max = 0;

    for (int i = 0; i < n; i++) {

        if (arr[i] > max)

            max = arr[i];

    }

    return max;

}

/\*

void visualization(int n, Robot robot, int step, Robot p[], Robot pr[])

{

    //Draw the robot, landmarks, particles and resampled particles on a graph

    //Graph Format

    plt::title("MCL, step " + to\_string(step));

    plt::xlim(0, 100);

    plt::ylim(0, 100);

    //Draw particles in green

    for (int i = 0; i < n; i++) {

        plt::plot({ p[i].x }, { p[i].y }, "go");

    }

    //Draw resampled particles in yellow

    for (int i = 0; i < n; i++) {

        plt::plot({ pr[i].x }, { pr[i].y }, "yo");

    }

    //Draw landmarks in red

    for (int i = 0; i < sizeof(landmarks) / sizeof(landmarks[0]); i++) {

        plt::plot({ landmarks[i][0] }, { landmarks[i][1] }, "ro");

    }

    //Draw robot position in blue

    plt::plot({ robot.x }, { robot.y }, "bo");

    //Save the image and close the plot

    plt::save("./Images/Step" + to\_string(step) + ".png");

    plt::clf();

}

\*/

//####   DON'T MODIFY ANYTHING ABOVE HERE! ENTER CODE BELOW ####

int main()

{

    Robot myrobot;

    // TODO: Simulate Noise

    // Forward Noise=5.0, Turn Noise=0.1,Sense Noise=5.0

    myrobot.set\_noise(Forward\_Noise, Turn\_Noise, Sense\_Noise);

    myrobot.set(30.0, 50.0, M\_PI / 2.0);

    myrobot.move(-M\_PI / 2.0, 15.0);

    cout << myrobot.read\_sensors() << endl;

    myrobot.move(-M\_PI / 2.0, 10.0);

    cout << myrobot.read\_sensors() << endl;

    return 0;

}

#### solution.cpp

//#include "src/matplotlibcpp.h"//Graph Library

#include <iostream>

#include <string>

#include <math.h>

#include <vector>

#include <stdexcept> // throw errors

#include <random> //C++ 11 Random Numbers

//namespace plt = matplotlibcpp;

using namespace std;

// Landmarks

double landmarks[8][2] = { { 20.0, 20.0 }, { 20.0, 80.0 }, { 20.0, 50.0 },

    { 50.0, 20.0 }, { 50.0, 80.0 }, { 80.0, 80.0 },

    { 80.0, 20.0 }, { 80.0, 50.0 } };

// Map size in meters

double world\_size = 100.0;

// Random Generators

random\_device rd;

mt19937 gen(rd());

// Global Functions

double mod(double first\_term, double second\_term);

double gen\_real\_random();

class Robot {

public:

    Robot()

    {

        // Constructor

        x = gen\_real\_random() \* world\_size; // robot's x coordinate

        y = gen\_real\_random() \* world\_size; // robot's y coordinate

        orient = gen\_real\_random() \* 2.0 \* M\_PI; // robot's orientation

        forward\_noise = 0.0; //noise of the forward movement

        turn\_noise = 0.0; //noise of the turn

        sense\_noise = 0.0; //noise of the sensing

    }

    void set(double new\_x, double new\_y, double new\_orient)

    {

        // Set robot new position and orientation

        if (new\_x < 0 || new\_x >= world\_size)

            throw std::invalid\_argument("X coordinate out of bound");

        if (new\_y < 0 || new\_y >= world\_size)

            throw std::invalid\_argument("Y coordinate out of bound");

        if (new\_orient < 0 || new\_orient >= 2 \* M\_PI)

            throw std::invalid\_argument("Orientation must be in [0..2pi]");

        x = new\_x;

        y = new\_y;

        orient = new\_orient;

    }

    void set\_noise(double new\_forward\_noise, double new\_turn\_noise, double new\_sense\_noise)

    {

        // Simulate noise, often useful in particle filters

        forward\_noise = new\_forward\_noise;

        turn\_noise = new\_turn\_noise;

        sense\_noise = new\_sense\_noise;

    }

    vector<double> sense()

    {

        // Measure the distances from the robot toward the landmarks

        vector<double> z(sizeof(landmarks) / sizeof(landmarks[0]));

        double dist;

        for (int i = 0; i < sizeof(landmarks) / sizeof(landmarks[0]); i++) {

            dist = sqrt(pow((x - landmarks[i][0]), 2) + pow((y - landmarks[i][1]), 2));

            dist += gen\_gauss\_random(0.0, sense\_noise);

            z[i] = dist;

        }

        return z;

    }

    Robot move(double turn, double forward)

    {

        if (forward < 0)

            throw std::invalid\_argument("Robot cannot move backward");

        // turn, and add randomness to the turning command

        orient = orient + turn + gen\_gauss\_random(0.0, turn\_noise);

        orient = mod(orient, 2 \* M\_PI);

        // move, and add randomness to the motion command

        double dist = forward + gen\_gauss\_random(0.0, forward\_noise);

        x = x + (cos(orient) \* dist);

        y = y + (sin(orient) \* dist);

        // cyclic truncate

        x = mod(x, world\_size);

        y = mod(y, world\_size);

        // set particle

        Robot res;

        res.set(x, y, orient);

        res.set\_noise(forward\_noise, turn\_noise, sense\_noise);

        return res;

    }

    string show\_pose()

    {

        // Returns the robot current position and orientation in a string format

        return "[x=" + to\_string(x) + " y=" + to\_string(y) + " orient=" + to\_string(orient) + "]";

    }

    string read\_sensors()

    {

        // Returns all the distances from the robot toward the landmarks

        vector<double> z = sense();

        string readings = "[";

        for (int i = 0; i < z.size(); i++) {

            readings += to\_string(z[i]) + " ";

        }

        readings[readings.size() - 1] = ']';

        return readings;

    }

    double measurement\_prob(vector<double> measurement)

    {

        // Calculates how likely a measurement should be

        double prob = 1.0;

        double dist;

        for (int i = 0; i < sizeof(landmarks) / sizeof(landmarks[0]); i++) {

            dist = sqrt(pow((x - landmarks[i][0]), 2) + pow((y - landmarks[i][1]), 2));

            prob \*= gaussian(dist, sense\_noise, measurement[i]);

        }

        return prob;

    }

    double x, y, orient; //robot poses

    double forward\_noise, turn\_noise, sense\_noise; //robot noises

private:

    double gen\_gauss\_random(double mean, double variance)

    {

        // Gaussian random

        normal\_distribution<double> gauss\_dist(mean, variance);

        return gauss\_dist(gen);

    }

    double gaussian(double mu, double sigma, double x)

    {

        // Probability of x for 1-dim Gaussian with mean mu and var. sigma

        return exp(-(pow((mu - x), 2)) / (pow(sigma, 2)) / 2.0) / sqrt(2.0 \* M\_PI \* (pow(sigma, 2)));

    }

};

// Functions

double gen\_real\_random()

{

    // Generate real random between 0 and 1

    uniform\_real\_distribution<double> real\_dist(0.0, 1.0); //Real

    return real\_dist(gen);

}

double mod(double first\_term, double second\_term)

{

    // Compute the modulus

    return first\_term - (second\_term)\*floor(first\_term / (second\_term));

}

double evaluation(Robot r, Robot p[], int n)

{

    //Calculate the mean error of the system

    double sum = 0.0;

    for (int i = 0; i < n; i++) {

        //the second part is because of world's cyclicity

        double dx = mod((p[i].x - r.x + (world\_size / 2.0)), world\_size) - (world\_size / 2.0);

        double dy = mod((p[i].y - r.y + (world\_size / 2.0)), world\_size) - (world\_size / 2.0);

        double err = sqrt(pow(dx, 2) + pow(dy, 2));

        sum += err;

    }

    return sum / n;

}

double max(double arr[], int n)

{

    // Identify the max element in an array

    double max = 0;

    for (int i = 0; i < n; i++) {

        if (arr[i] > max)

            max = arr[i];

    }

    return max;

}

/\*

void visualization(int n, Robot robot, int step, Robot p[], Robot pr[])

{

    //Draw the robot, landmarks, particles and resampled particles on a graph

    //Graph Format

    plt::title("MCL, step " + to\_string(step));

    plt::xlim(0, 100);

    plt::ylim(0, 100);

    //Draw particles in green

    for (int i = 0; i < n; i++) {

        plt::plot({ p[i].x }, { p[i].y }, "go");

    }

    //Draw resampled particles in yellow

    for (int i = 0; i < n; i++) {

        plt::plot({ pr[i].x }, { pr[i].y }, "yo");

    }

    //Draw landmarks in red

    for (int i = 0; i < sizeof(landmarks) / sizeof(landmarks[0]); i++) {

        plt::plot({ landmarks[i][0] }, { landmarks[i][1] }, "ro");

    }

    //Draw robot position in blue

    plt::plot({ robot.x }, { robot.y }, "bo");

    //Save the image and close the plot

    plt::save("./Images/Step" + to\_string(step) + ".png");

    plt::clf();

}

\*/

//####   DON'T MODIFY ANYTHING ABOVE HERE! ENTER CODE BELOW ####

int main()

{

    Robot myrobot;

    // Simulate Noise

    // Forward Noise=5.0, Turn Noise=0.1,Sense Noise=5.0

    myrobot.set\_noise(5.0, 0.1, 5.0);

    myrobot.set(30.0, 50.0, M\_PI / 2.0);

    myrobot.move(-M\_PI / 2.0, 15.0);

    cout << myrobot.read\_sensors() << endl;

    myrobot.move(-M\_PI / 2.0, 10.0);

    cout << myrobot.read\_sensors() << endl;

    return 0;

}

### Particle Filter

Video 5.6

A screenshot of a computer

AI-generated content may be incorrect.

#### main.cpp

//#include "src/matplotlibcpp.h"//Graph Library

#include <iostream>

#include <string>

#include <math.h>

#include <vector>

#include <stdexcept> // throw errors

#include <random> //C++ 11 Random Numbers

//namespace plt = matplotlibcpp;

using namespace std;

// Landmarks

double landmarks[8][2] = { { 20.0, 20.0 }, { 20.0, 80.0 }, { 20.0, 50.0 },

    { 50.0, 20.0 }, { 50.0, 80.0 }, { 80.0, 80.0 },

    { 80.0, 20.0 }, { 80.0, 50.0 } };

// Map size in meters

double world\_size = 100.0;

// Random Generators

random\_device rd;

mt19937 gen(rd());

// Global Functions

double mod(double first\_term, double second\_term);

double gen\_real\_random();

class Robot {

public:

    Robot()

    {

        // Constructor

        x = gen\_real\_random() \* world\_size; // robot's x coordinate

        y = gen\_real\_random() \* world\_size; // robot's y coordinate

        orient = gen\_real\_random() \* 2.0 \* M\_PI; // robot's orientation

        forward\_noise = 0.0; //noise of the forward movement

        turn\_noise = 0.0; //noise of the turn

        sense\_noise = 0.0; //noise of the sensing

    }

    void set(double new\_x, double new\_y, double new\_orient)

    {

        // Set robot new position and orientation

        if (new\_x < 0 || new\_x >= world\_size)

            throw std::invalid\_argument("X coordinate out of bound");

        if (new\_y < 0 || new\_y >= world\_size)

            throw std::invalid\_argument("Y coordinate out of bound");

        if (new\_orient < 0 || new\_orient >= 2 \* M\_PI)

            throw std::invalid\_argument("Orientation must be in [0..2pi]");

        x = new\_x;

        y = new\_y;

        orient = new\_orient;

    }

    void set\_noise(double new\_forward\_noise, double new\_turn\_noise, double new\_sense\_noise)

    {

        // Simulate noise, often useful in particle filters

        forward\_noise = new\_forward\_noise;

        turn\_noise = new\_turn\_noise;

        sense\_noise = new\_sense\_noise;

    }

    vector<double> sense()

    {

        // Measure the distances from the robot toward the landmarks

        vector<double> z(sizeof(landmarks) / sizeof(landmarks[0]));

        double dist;

        for (int i = 0; i < sizeof(landmarks) / sizeof(landmarks[0]); i++) {

            dist = sqrt(pow((x - landmarks[i][0]), 2) + pow((y - landmarks[i][1]), 2));

            dist += gen\_gauss\_random(0.0, sense\_noise);

            z[i] = dist;

        }

        return z;

    }

    Robot move(double turn, double forward)

    {

        if (forward < 0)

            throw std::invalid\_argument("Robot cannot move backward");

        // turn, and add randomness to the turning command

        orient = orient + turn + gen\_gauss\_random(0.0, turn\_noise);

        orient = mod(orient, 2 \* M\_PI);

        // move, and add randomness to the motion command

        double dist = forward + gen\_gauss\_random(0.0, forward\_noise);

        x = x + (cos(orient) \* dist);

        y = y + (sin(orient) \* dist);

        // cyclic truncate

        x = mod(x, world\_size);

        y = mod(y, world\_size);

        // set particle

        Robot res;

        res.set(x, y, orient);

        res.set\_noise(forward\_noise, turn\_noise, sense\_noise);

        return res;

    }

    string show\_pose()

    {

        // Returns the robot current position and orientation in a string format

        return "[x=" + to\_string(x) + " y=" + to\_string(y) + " orient=" + to\_string(orient) + "]";

    }

    string read\_sensors()

    {

        // Returns all the distances from the robot toward the landmarks

        vector<double> z = sense();

        string readings = "[";

        for (int i = 0; i < z.size(); i++) {

            readings += to\_string(z[i]) + " ";

        }

        readings[readings.size() - 1] = ']';

        return readings;

    }

    double measurement\_prob(vector<double> measurement)

    {

        // Calculates how likely a measurement should be

        double prob = 1.0;

        double dist;

        for (int i = 0; i < sizeof(landmarks) / sizeof(landmarks[0]); i++) {

            dist = sqrt(pow((x - landmarks[i][0]), 2) + pow((y - landmarks[i][1]), 2));

            prob \*= gaussian(dist, sense\_noise, measurement[i]);

        }

        return prob;

    }

    double x, y, orient; //robot poses

    double forward\_noise, turn\_noise, sense\_noise; //robot noises

private:

    double gen\_gauss\_random(double mean, double variance)

    {

        // Gaussian random

        normal\_distribution<double> gauss\_dist(mean, variance);

        return gauss\_dist(gen);

    }

    double gaussian(double mu, double sigma, double x)

    {

        // Probability of x for 1-dim Gaussian with mean mu and var. sigma

        return exp(-(pow((mu - x), 2)) / (pow(sigma, 2)) / 2.0) / sqrt(2.0 \* M\_PI \* (pow(sigma, 2)));

    }

};

// Functions

double gen\_real\_random()

{

    // Generate real random between 0 and 1

    uniform\_real\_distribution<double> real\_dist(0.0, 1.0); //Real

    return real\_dist(gen);

}

double mod(double first\_term, double second\_term)

{

    // Compute the modulus

    return first\_term - (second\_term)\*floor(first\_term / (second\_term));

}

double evaluation(Robot r, Robot p[], int n)

{

    //Calculate the mean error of the system

    double sum = 0.0;

    for (int i = 0; i < n; i++) {

        //the second part is because of world's cyclicity

        double dx = mod((p[i].x - r.x + (world\_size / 2.0)), world\_size) - (world\_size / 2.0);

        double dy = mod((p[i].y - r.y + (world\_size / 2.0)), world\_size) - (world\_size / 2.0);

        double err = sqrt(pow(dx, 2) + pow(dy, 2));

        sum += err;

    }

    return sum / n;

}

double max(double arr[], int n)

{

    // Identify the max element in an array

    double max = 0;

    for (int i = 0; i < n; i++) {

        if (arr[i] > max)

            max = arr[i];

    }

    return max;

}

/\*

void visualization(int n, Robot robot, int step, Robot p[], Robot pr[])

{

    //Draw the robot, landmarks, particles and resampled particles on a graph

    //Graph Format

    plt::title("MCL, step " + to\_string(step));

    plt::xlim(0, 100);

    plt::ylim(0, 100);

    //Draw particles in green

    for (int i = 0; i < n; i++) {

        plt::plot({ p[i].x }, { p[i].y }, "go");

    }

    //Draw resampled particles in yellow

    for (int i = 0; i < n; i++) {

        plt::plot({ pr[i].x }, { pr[i].y }, "yo");

    }

    //Draw landmarks in red

    for (int i = 0; i < sizeof(landmarks) / sizeof(landmarks[0]); i++) {

        plt::plot({ landmarks[i][0] }, { landmarks[i][1] }, "ro");

    }

    //Draw robot position in blue

    plt::plot({ robot.x }, { robot.y }, "bo");

    //Save the image and close the plot

    plt::save("./Images/Step" + to\_string(step) + ".png");

    plt::clf();

}

\*/

int main()

{

    //Practice Interfacing with Robot Class

    Robot myrobot;

    myrobot.set\_noise(5.0, 0.1, 5.0);

    myrobot.set(30.0, 50.0, M\_PI / 2.0);

    myrobot.move(-M\_PI / 2.0, 15.0);

    //cout << myrobot.read\_sensors() << endl;

    myrobot.move(-M\_PI / 2.0, 10.0);

    //cout << myrobot.read\_sensors() << endl;

    //####   DON'T MODIFY ANYTHING ABOVE HERE! ENTER CODE BELOW ####

    // Instantiating 1000 Particles each with a random position and orientation

    int n = 1000;

    Robot p[n];

    //TODO: Your job is to loop over the set of particles

    //TODO: For each particle add noise: Forward\_Noise=0.05, Turn\_Noise=0.05, and Sense\_Noise=5.0

    //TODO: And print its pose on a single line

    return 0;

}

#### solution.cpp

//#include "src/matplotlibcpp.h"//Graph Library

#include <iostream>

#include <string>

#include <math.h>

#include <vector>

#include <stdexcept> // throw errors

#include <random> //C++ 11 Random Numbers

//namespace plt = matplotlibcpp;

using namespace std;

// Landmarks

double landmarks[8][2] = { { 20.0, 20.0 }, { 20.0, 80.0 }, { 20.0, 50.0 },

    { 50.0, 20.0 }, { 50.0, 80.0 }, { 80.0, 80.0 },

    { 80.0, 20.0 }, { 80.0, 50.0 } };

// Map size in meters

double world\_size = 100.0;

// Random Generators

random\_device rd;

mt19937 gen(rd());

// Global Functions

double mod(double first\_term, double second\_term);

double gen\_real\_random();

class Robot {

public:

    Robot()

    {

        // Constructor

        x = gen\_real\_random() \* world\_size; // robot's x coordinate

        y = gen\_real\_random() \* world\_size; // robot's y coordinate

        orient = gen\_real\_random() \* 2.0 \* M\_PI; // robot's orientation

        forward\_noise = 0.0; //noise of the forward movement

        turn\_noise = 0.0; //noise of the turn

        sense\_noise = 0.0; //noise of the sensing

    }

    void set(double new\_x, double new\_y, double new\_orient)

    {

        // Set robot new position and orientation

        if (new\_x < 0 || new\_x >= world\_size)

            throw std::invalid\_argument("X coordinate out of bound");

        if (new\_y < 0 || new\_y >= world\_size)

            throw std::invalid\_argument("Y coordinate out of bound");

        if (new\_orient < 0 || new\_orient >= 2 \* M\_PI)

            throw std::invalid\_argument("Orientation must be in [0..2pi]");

        x = new\_x;

        y = new\_y;

        orient = new\_orient;

    }

    void set\_noise(double new\_forward\_noise, double new\_turn\_noise, double new\_sense\_noise)

    {

        // Simulate noise, often useful in particle filters

        forward\_noise = new\_forward\_noise;

        turn\_noise = new\_turn\_noise;

        sense\_noise = new\_sense\_noise;

    }

    vector<double> sense()

    {

        // Measure the distances from the robot toward the landmarks

        vector<double> z(sizeof(landmarks) / sizeof(landmarks[0]));

        double dist;

        for (int i = 0; i < sizeof(landmarks) / sizeof(landmarks[0]); i++) {

            dist = sqrt(pow((x - landmarks[i][0]), 2) + pow((y - landmarks[i][1]), 2));

            dist += gen\_gauss\_random(0.0, sense\_noise);

            z[i] = dist;

        }

        return z;

    }

    Robot move(double turn, double forward)

    {

        if (forward < 0)

            throw std::invalid\_argument("Robot cannot move backward");

        // turn, and add randomness to the turning command

        orient = orient + turn + gen\_gauss\_random(0.0, turn\_noise);

        orient = mod(orient, 2 \* M\_PI);

        // move, and add randomness to the motion command

        double dist = forward + gen\_gauss\_random(0.0, forward\_noise);

        x = x + (cos(orient) \* dist);

        y = y + (sin(orient) \* dist);

        // cyclic truncate

        x = mod(x, world\_size);

        y = mod(y, world\_size);

        // set particle

        Robot res;

        res.set(x, y, orient);

        res.set\_noise(forward\_noise, turn\_noise, sense\_noise);

        return res;

    }

    string show\_pose()

    {

        // Returns the robot current position and orientation in a string format

        return "[x=" + to\_string(x) + " y=" + to\_string(y) + " orient=" + to\_string(orient) + "]";

    }

    string read\_sensors()

    {

        // Returns all the distances from the robot toward the landmarks

        vector<double> z = sense();

        string readings = "[";

        for (int i = 0; i < z.size(); i++) {

            readings += to\_string(z[i]) + " ";

        }

        readings[readings.size() - 1] = ']';

        return readings;

    }

    double measurement\_prob(vector<double> measurement)

    {

        // Calculates how likely a measurement should be

        double prob = 1.0;

        double dist;

        for (int i = 0; i < sizeof(landmarks) / sizeof(landmarks[0]); i++) {

            dist = sqrt(pow((x - landmarks[i][0]), 2) + pow((y - landmarks[i][1]), 2));

            prob \*= gaussian(dist, sense\_noise, measurement[i]);

        }

        return prob;

    }

    double x, y, orient; //robot poses

    double forward\_noise, turn\_noise, sense\_noise; //robot noises

private:

    double gen\_gauss\_random(double mean, double variance)

    {

        // Gaussian random

        normal\_distribution<double> gauss\_dist(mean, variance);

        return gauss\_dist(gen);

    }

    double gaussian(double mu, double sigma, double x)

    {

        // Probability of x for 1-dim Gaussian with mean mu and var. sigma

        return exp(-(pow((mu - x), 2)) / (pow(sigma, 2)) / 2.0) / sqrt(2.0 \* M\_PI \* (pow(sigma, 2)));

    }

};

// Functions

double gen\_real\_random()

{

    // Generate real random between 0 and 1

    uniform\_real\_distribution<double> real\_dist(0.0, 1.0); //Real

    return real\_dist(gen);

}

double mod(double first\_term, double second\_term)

{

    // Compute the modulus

    return first\_term - (second\_term)\*floor(first\_term / (second\_term));

}

double evaluation(Robot r, Robot p[], int n)

{

    //Calculate the mean error of the system

    double sum = 0.0;

    for (int i = 0; i < n; i++) {

        //the second part is because of world's cyclicity

        double dx = mod((p[i].x - r.x + (world\_size / 2.0)), world\_size) - (world\_size / 2.0);

        double dy = mod((p[i].y - r.y + (world\_size / 2.0)), world\_size) - (world\_size / 2.0);

        double err = sqrt(pow(dx, 2) + pow(dy, 2));

        sum += err;

    }

    return sum / n;

}

double max(double arr[], int n)

{

    // Identify the max element in an array

    double max = 0;

    for (int i = 0; i < n; i++) {

        if (arr[i] > max)

            max = arr[i];

    }

    return max;

}

/\*

void visualization(int n, Robot robot, int step, Robot p[], Robot pr[])

{

    //Draw the robot, landmarks, particles and resampled particles on a graph

    //Graph Format

    plt::title("MCL, step " + to\_string(step));

    plt::xlim(0, 100);

    plt::ylim(0, 100);

    //Draw particles in green

    for (int i = 0; i < n; i++) {

        plt::plot({ p[i].x }, { p[i].y }, "go");

    }

    //Draw resampled particles in yellow

    for (int i = 0; i < n; i++) {

        plt::plot({ pr[i].x }, { pr[i].y }, "yo");

    }

    //Draw landmarks in red

    for (int i = 0; i < sizeof(landmarks) / sizeof(landmarks[0]); i++) {

        plt::plot({ landmarks[i][0] }, { landmarks[i][1] }, "ro");

    }

    //Draw robot position in blue

    plt::plot({ robot.x }, { robot.y }, "bo");

    //Save the image and close the plot

    plt::save("./Images/Step" + to\_string(step) + ".png");

    plt::clf();

}

\*/

int main()

{

    //Practice Interfacing with Robot Class

    Robot myrobot;

    myrobot.set\_noise(5.0, 0.1, 5.0);

    myrobot.set(30.0, 50.0, M\_PI / 2.0);

    myrobot.move(-M\_PI / 2.0, 15.0);

    //cout << myrobot.read\_sensors() << endl;

    myrobot.move(-M\_PI / 2.0, 10.0);

    //cout << myrobot.read\_sensors() << endl;

    //####   DON'T MODIFY ANYTHING ABOVE HERE! ENTER CODE BELOW ####

    // Instantiating 1000 Particles each with a random position and orientation

    int n = 1000;

    Robot p[n];

    //Your job is to loop over the set of particles

    //For each particle add noise: Forward\_Noise=0.05, Turn\_Noise=0.05, and Sense\_Noise=5.0

    //And print its pose on a single line

    for (int i = 0; i < n; i++) {

        p[i].set\_noise(0.05, 0.05, 5.0);

        cout << p[i].show\_pose() << endl;

    }

    return 0;

}

#### A white background with black text AI-generated content may be incorrect.

#### main.cpp

//#include "src/matplotlibcpp.h"//Graph Library

#include <iostream>

#include <string>

#include <math.h>

#include <vector>

#include <stdexcept> // throw errors

#include <random> //C++ 11 Random Numbers

//namespace plt = matplotlibcpp;

using namespace std;

// Landmarks

double landmarks[8][2] = { { 20.0, 20.0 }, { 20.0, 80.0 }, { 20.0, 50.0 },

    { 50.0, 20.0 }, { 50.0, 80.0 }, { 80.0, 80.0 },

    { 80.0, 20.0 }, { 80.0, 50.0 } };

// Map size in meters

double world\_size = 100.0;

// Random Generators

random\_device rd;

mt19937 gen(rd());

// Global Functions

double mod(double first\_term, double second\_term);

double gen\_real\_random();

class Robot {

public:

    Robot()

    {

        // Constructor

        x = gen\_real\_random() \* world\_size; // robot's x coordinate

        y = gen\_real\_random() \* world\_size; // robot's y coordinate

        orient = gen\_real\_random() \* 2.0 \* M\_PI; // robot's orientation

        forward\_noise = 0.0; //noise of the forward movement

        turn\_noise = 0.0; //noise of the turn

        sense\_noise = 0.0; //noise of the sensing

    }

    void set(double new\_x, double new\_y, double new\_orient)

    {

        // Set robot new position and orientation

        if (new\_x < 0 || new\_x >= world\_size)

            throw std::invalid\_argument("X coordinate out of bound");

        if (new\_y < 0 || new\_y >= world\_size)

            throw std::invalid\_argument("Y coordinate out of bound");

        if (new\_orient < 0 || new\_orient >= 2 \* M\_PI)

            throw std::invalid\_argument("Orientation must be in [0..2pi]");

        x = new\_x;

        y = new\_y;

        orient = new\_orient;

    }

    void set\_noise(double new\_forward\_noise, double new\_turn\_noise, double new\_sense\_noise)

    {

        // Simulate noise, often useful in particle filters

        forward\_noise = new\_forward\_noise;

        turn\_noise = new\_turn\_noise;

        sense\_noise = new\_sense\_noise;

    }

    vector<double> sense()

    {

        // Measure the distances from the robot toward the landmarks

        vector<double> z(sizeof(landmarks) / sizeof(landmarks[0]));

        double dist;

        for (int i = 0; i < sizeof(landmarks) / sizeof(landmarks[0]); i++) {

            dist = sqrt(pow((x - landmarks[i][0]), 2) + pow((y - landmarks[i][1]), 2));

            dist += gen\_gauss\_random(0.0, sense\_noise);

            z[i] = dist;

        }

        return z;

    }

    Robot move(double turn, double forward)

    {

        if (forward < 0)

            throw std::invalid\_argument("Robot cannot move backward");

        // turn, and add randomness to the turning command

        orient = orient + turn + gen\_gauss\_random(0.0, turn\_noise);

        orient = mod(orient, 2 \* M\_PI);

        // move, and add randomness to the motion command

        double dist = forward + gen\_gauss\_random(0.0, forward\_noise);

        x = x + (cos(orient) \* dist);

        y = y + (sin(orient) \* dist);

        // cyclic truncate

        x = mod(x, world\_size);

        y = mod(y, world\_size);

        // set particle

        Robot res;

        res.set(x, y, orient);

        res.set\_noise(forward\_noise, turn\_noise, sense\_noise);

        return res;

    }

    string show\_pose()

    {

        // Returns the robot current position and orientation in a string format

        return "[x=" + to\_string(x) + " y=" + to\_string(y) + " orient=" + to\_string(orient) + "]";

    }

    string read\_sensors()

    {

        // Returns all the distances from the robot toward the landmarks

        vector<double> z = sense();

        string readings = "[";

        for (int i = 0; i < z.size(); i++) {

            readings += to\_string(z[i]) + " ";

        }

        readings[readings.size() - 1] = ']';

        return readings;

    }

    double measurement\_prob(vector<double> measurement)

    {

        // Calculates how likely a measurement should be

        double prob = 1.0;

        double dist;

        for (int i = 0; i < sizeof(landmarks) / sizeof(landmarks[0]); i++) {

            dist = sqrt(pow((x - landmarks[i][0]), 2) + pow((y - landmarks[i][1]), 2));

            prob \*= gaussian(dist, sense\_noise, measurement[i]);

        }

        return prob;

    }

    double x, y, orient; //robot poses

    double forward\_noise, turn\_noise, sense\_noise; //robot noises

private:

    double gen\_gauss\_random(double mean, double variance)

    {

        // Gaussian random

        normal\_distribution<double> gauss\_dist(mean, variance);

        return gauss\_dist(gen);

    }

    double gaussian(double mu, double sigma, double x)

    {

        // Probability of x for 1-dim Gaussian with mean mu and var. sigma

        return exp(-(pow((mu - x), 2)) / (pow(sigma, 2)) / 2.0) / sqrt(2.0 \* M\_PI \* (pow(sigma, 2)));

    }

};

// Functions

double gen\_real\_random()

{

    // Generate real random between 0 and 1

    uniform\_real\_distribution<double> real\_dist(0.0, 1.0); //Real

    return real\_dist(gen);

}

double mod(double first\_term, double second\_term)

{

    // Compute the modulus

    return first\_term - (second\_term)\*floor(first\_term / (second\_term));

}

double evaluation(Robot r, Robot p[], int n)

{

    //Calculate the mean error of the system

    double sum = 0.0;

    for (int i = 0; i < n; i++) {

        //the second part is because of world's cyclicity

        double dx = mod((p[i].x - r.x + (world\_size / 2.0)), world\_size) - (world\_size / 2.0);

        double dy = mod((p[i].y - r.y + (world\_size / 2.0)), world\_size) - (world\_size / 2.0);

        double err = sqrt(pow(dx, 2) + pow(dy, 2));

        sum += err;

    }

    return sum / n;

}

double max(double arr[], int n)

{

    // Identify the max element in an array

    double max = 0;

    for (int i = 0; i < n; i++) {

        if (arr[i] > max)

            max = arr[i];

    }

    return max;

}

/\*

void visualization(int n, Robot robot, int step, Robot p[], Robot pr[])

{

    //Draw the robot, landmarks, particles and resampled particles on a graph

    //Graph Format

    plt::title("MCL, step " + to\_string(step));

    plt::xlim(0, 100);

    plt::ylim(0, 100);

    //Draw particles in green

    for (int i = 0; i < n; i++) {

        plt::plot({ p[i].x }, { p[i].y }, "go");

    }

    //Draw resampled particles in yellow

    for (int i = 0; i < n; i++) {

        plt::plot({ pr[i].x }, { pr[i].y }, "yo");

    }

    //Draw landmarks in red

    for (int i = 0; i < sizeof(landmarks) / sizeof(landmarks[0]); i++) {

        plt::plot({ landmarks[i][0] }, { landmarks[i][1] }, "ro");

    }

    //Draw robot position in blue

    plt::plot({ robot.x }, { robot.y }, "bo");

    //Save the image and close the plot

    plt::save("./Images/Step" + to\_string(step) + ".png");

    plt::clf();

}

\*/

int main()

{

    //Practice Interfacing with Robot Class

    Robot myrobot;

    myrobot.set\_noise(5.0, 0.1, 5.0);

    myrobot.set(30.0, 50.0, M\_PI / 2.0);

    myrobot.move(-M\_PI / 2.0, 15.0);

    //cout << myrobot.read\_sensors() << endl;

    myrobot.move(-M\_PI / 2.0, 10.0);

    //cout << myrobot.read\_sensors() << endl;

    // Create a set of particles

    int n = 1000;

    Robot p[n];

    for (int i = 0; i < n; i++) {

        p[i].set\_noise(0.05, 0.05, 5.0);

        //cout << p[i].show\_pose() << endl;

    }

    //####   DON'T MODIFY ANYTHING ABOVE HERE! ENTER CODE BELOW ####

    //Now, simulate motion for each particle

    //TODO: Create a new particle set 'p2'

    //TODO: Rotate each particle by 0.1 and move it forward by 5.0

    //TODO: Assign 'p2' to 'p' and print the particle poses, each on a single line

    return 0;

}

#### solution.cpp

//#include "src/matplotlibcpp.h"//Graph Library

#include <iostream>

#include <string>

#include <math.h>

#include <vector>

#include <stdexcept> // throw errors

#include <random> //C++ 11 Random Numbers

//namespace plt = matplotlibcpp;

using namespace std;

// Landmarks

double landmarks[8][2] = { { 20.0, 20.0 }, { 20.0, 80.0 }, { 20.0, 50.0 },

    { 50.0, 20.0 }, { 50.0, 80.0 }, { 80.0, 80.0 },

    { 80.0, 20.0 }, { 80.0, 50.0 } };

// Map size in meters

double world\_size = 100.0;

// Random Generators

random\_device rd;

mt19937 gen(rd());

// Global Functions

double mod(double first\_term, double second\_term);

double gen\_real\_random();

class Robot {

public:

    Robot()

    {

        // Constructor

        x = gen\_real\_random() \* world\_size; // robot's x coordinate

        y = gen\_real\_random() \* world\_size; // robot's y coordinate

        orient = gen\_real\_random() \* 2.0 \* M\_PI; // robot's orientation

        forward\_noise = 0.0; //noise of the forward movement

        turn\_noise = 0.0; //noise of the turn

        sense\_noise = 0.0; //noise of the sensing

    }

    void set(double new\_x, double new\_y, double new\_orient)

    {

        // Set robot new position and orientation

        if (new\_x < 0 || new\_x >= world\_size)

            throw std::invalid\_argument("X coordinate out of bound");

        if (new\_y < 0 || new\_y >= world\_size)

            throw std::invalid\_argument("Y coordinate out of bound");

        if (new\_orient < 0 || new\_orient >= 2 \* M\_PI)

            throw std::invalid\_argument("Orientation must be in [0..2pi]");

        x = new\_x;

        y = new\_y;

        orient = new\_orient;

    }

    void set\_noise(double new\_forward\_noise, double new\_turn\_noise, double new\_sense\_noise)

    {

        // Simulate noise, often useful in particle filters

        forward\_noise = new\_forward\_noise;

        turn\_noise = new\_turn\_noise;

        sense\_noise = new\_sense\_noise;

    }

    vector<double> sense()

    {

        // Measure the distances from the robot toward the landmarks

        vector<double> z(sizeof(landmarks) / sizeof(landmarks[0]));

        double dist;

        for (int i = 0; i < sizeof(landmarks) / sizeof(landmarks[0]); i++) {

            dist = sqrt(pow((x - landmarks[i][0]), 2) + pow((y - landmarks[i][1]), 2));

            dist += gen\_gauss\_random(0.0, sense\_noise);

            z[i] = dist;

        }

        return z;

    }

    Robot move(double turn, double forward)

    {

        if (forward < 0)

            throw std::invalid\_argument("Robot cannot move backward");

        // turn, and add randomness to the turning command

        orient = orient + turn + gen\_gauss\_random(0.0, turn\_noise);

        orient = mod(orient, 2 \* M\_PI);

        // move, and add randomness to the motion command

        double dist = forward + gen\_gauss\_random(0.0, forward\_noise);

        x = x + (cos(orient) \* dist);

        y = y + (sin(orient) \* dist);

        // cyclic truncate

        x = mod(x, world\_size);

        y = mod(y, world\_size);

        // set particle

        Robot res;

        res.set(x, y, orient);

        res.set\_noise(forward\_noise, turn\_noise, sense\_noise);

        return res;

    }

    string show\_pose()

    {

        // Returns the robot current position and orientation in a string format

        return "[x=" + to\_string(x) + " y=" + to\_string(y) + " orient=" + to\_string(orient) + "]";

    }

    string read\_sensors()

    {

        // Returns all the distances from the robot toward the landmarks

        vector<double> z = sense();

        string readings = "[";

        for (int i = 0; i < z.size(); i++) {

            readings += to\_string(z[i]) + " ";

        }

        readings[readings.size() - 1] = ']';

        return readings;

    }

    double measurement\_prob(vector<double> measurement)

    {

        // Calculates how likely a measurement should be

        double prob = 1.0;

        double dist;

        for (int i = 0; i < sizeof(landmarks) / sizeof(landmarks[0]); i++) {

            dist = sqrt(pow((x - landmarks[i][0]), 2) + pow((y - landmarks[i][1]), 2));

            prob \*= gaussian(dist, sense\_noise, measurement[i]);

        }

        return prob;

    }

    double x, y, orient; //robot poses

    double forward\_noise, turn\_noise, sense\_noise; //robot noises

private:

    double gen\_gauss\_random(double mean, double variance)

    {

        // Gaussian random

        normal\_distribution<double> gauss\_dist(mean, variance);

        return gauss\_dist(gen);

    }

    double gaussian(double mu, double sigma, double x)

    {

        // Probability of x for 1-dim Gaussian with mean mu and var. sigma

        return exp(-(pow((mu - x), 2)) / (pow(sigma, 2)) / 2.0) / sqrt(2.0 \* M\_PI \* (pow(sigma, 2)));

    }

};

// Functions

double gen\_real\_random()

{

    // Generate real random between 0 and 1

    uniform\_real\_distribution<double> real\_dist(0.0, 1.0); //Real

    return real\_dist(gen);

}

double mod(double first\_term, double second\_term)

{

    // Compute the modulus

    return first\_term - (second\_term)\*floor(first\_term / (second\_term));

}

double evaluation(Robot r, Robot p[], int n)

{

    //Calculate the mean error of the system

    double sum = 0.0;

    for (int i = 0; i < n; i++) {

        //the second part is because of world's cyclicity

        double dx = mod((p[i].x - r.x + (world\_size / 2.0)), world\_size) - (world\_size / 2.0);

        double dy = mod((p[i].y - r.y + (world\_size / 2.0)), world\_size) - (world\_size / 2.0);

        double err = sqrt(pow(dx, 2) + pow(dy, 2));

        sum += err;

    }

    return sum / n;

}

double max(double arr[], int n)

{

    // Identify the max element in an array

    double max = 0;

    for (int i = 0; i < n; i++) {

        if (arr[i] > max)

            max = arr[i];

    }

    return max;

}

/\*

void visualization(int n, Robot robot, int step, Robot p[], Robot pr[])

{

    //Draw the robot, landmarks, particles and resampled particles on a graph

    //Graph Format

    plt::title("MCL, step " + to\_string(step));

    plt::xlim(0, 100);

    plt::ylim(0, 100);

    //Draw particles in green

    for (int i = 0; i < n; i++) {

        plt::plot({ p[i].x }, { p[i].y }, "go");

    }

    //Draw resampled particles in yellow

    for (int i = 0; i < n; i++) {

        plt::plot({ pr[i].x }, { pr[i].y }, "yo");

    }

    //Draw landmarks in red

    for (int i = 0; i < sizeof(landmarks) / sizeof(landmarks[0]); i++) {

        plt::plot({ landmarks[i][0] }, { landmarks[i][1] }, "ro");

    }

    //Draw robot position in blue

    plt::plot({ robot.x }, { robot.y }, "bo");

    //Save the image and close the plot

    plt::save("./Images/Step" + to\_string(step) + ".png");

    plt::clf();

}

\*/

int main()

{

    //Practice Interfacing with Robot Class

    Robot myrobot;

    myrobot.set\_noise(5.0, 0.1, 5.0);

    myrobot.set(30.0, 50.0, M\_PI / 2.0);

    myrobot.move(-M\_PI / 2.0, 15.0);

    //cout << myrobot.read\_sensors() << endl;

    myrobot.move(-M\_PI / 2.0, 10.0);

    //cout << myrobot.read\_sensors() << endl;

    // Create a set of particles

    int n = 1000;

    Robot p[n];

    for (int i = 0; i < n; i++) {

        p[i].set\_noise(0.05, 0.05, 5.0);

        //cout << p[i].show\_pose() << endl;

    }

    //####   DON'T MODIFY ANYTHING ABOVE HERE! ENTER CODE BELOW ####

    //Now, simulate motion for each particle

    //Create a new particle set 'p2'

    //Rotate each particle by 0.1 and move it forward by 5.0

    //Assign p2 to p and print the particle poses, each on a single line

    Robot p2[n];

    for (int i = 0; i < n; i++) {

        p2[i] = p[i].move(0.1, 5.0);

        p[i] = p2[i];

        cout << p[i].show\_pose() << endl;

    }

    return 0;

}

### Importance Weight

Video 5.7

A white text with black text

AI-generated content may be incorrect.

#### main.cpp

//#include "src/matplotlibcpp.h"//Graph Library

#include <iostream>

#include <string>

#include <math.h>

#include <vector>

#include <stdexcept> // throw errors

#include <random> //C++ 11 Random Numbers

//namespace plt = matplotlibcpp;

using namespace std;

// Landmarks

double landmarks[8][2] = { { 20.0, 20.0 }, { 20.0, 80.0 }, { 20.0, 50.0 },

    { 50.0, 20.0 }, { 50.0, 80.0 }, { 80.0, 80.0 },

    { 80.0, 20.0 }, { 80.0, 50.0 } };

// Map size in meters

double world\_size = 100.0;

// Random Generators

random\_device rd;

mt19937 gen(rd());

// Global Functions

double mod(double first\_term, double second\_term);

double gen\_real\_random();

class Robot {

public:

    Robot()

    {

        // Constructor

        x = gen\_real\_random() \* world\_size; // robot's x coordinate

        y = gen\_real\_random() \* world\_size; // robot's y coordinate

        orient = gen\_real\_random() \* 2.0 \* M\_PI; // robot's orientation

        forward\_noise = 0.0; //noise of the forward movement

        turn\_noise = 0.0; //noise of the turn

        sense\_noise = 0.0; //noise of the sensing

    }

    void set(double new\_x, double new\_y, double new\_orient)

    {

        // Set robot new position and orientation

        if (new\_x < 0 || new\_x >= world\_size)

            throw std::invalid\_argument("X coordinate out of bound");

        if (new\_y < 0 || new\_y >= world\_size)

            throw std::invalid\_argument("Y coordinate out of bound");

        if (new\_orient < 0 || new\_orient >= 2 \* M\_PI)

            throw std::invalid\_argument("Orientation must be in [0..2pi]");

        x = new\_x;

        y = new\_y;

        orient = new\_orient;

    }

    void set\_noise(double new\_forward\_noise, double new\_turn\_noise, double new\_sense\_noise)

    {

        // Simulate noise, often useful in particle filters

        forward\_noise = new\_forward\_noise;

        turn\_noise = new\_turn\_noise;

        sense\_noise = new\_sense\_noise;

    }

    vector<double> sense()

    {

        // Measure the distances from the robot toward the landmarks

        vector<double> z(sizeof(landmarks) / sizeof(landmarks[0]));

        double dist;

        for (int i = 0; i < sizeof(landmarks) / sizeof(landmarks[0]); i++) {

            dist = sqrt(pow((x - landmarks[i][0]), 2) + pow((y - landmarks[i][1]), 2));

            dist += gen\_gauss\_random(0.0, sense\_noise);

            z[i] = dist;

        }

        return z;

    }

    Robot move(double turn, double forward)

    {

        if (forward < 0)

            throw std::invalid\_argument("Robot cannot move backward");

        // turn, and add randomness to the turning command

        orient = orient + turn + gen\_gauss\_random(0.0, turn\_noise);

        orient = mod(orient, 2 \* M\_PI);

        // move, and add randomness to the motion command

        double dist = forward + gen\_gauss\_random(0.0, forward\_noise);

        x = x + (cos(orient) \* dist);

        y = y + (sin(orient) \* dist);

        // cyclic truncate

        x = mod(x, world\_size);

        y = mod(y, world\_size);

        // set particle

        Robot res;

        res.set(x, y, orient);

        res.set\_noise(forward\_noise, turn\_noise, sense\_noise);

        return res;

    }

    string show\_pose()

    {

        // Returns the robot current position and orientation in a string format

        return "[x=" + to\_string(x) + " y=" + to\_string(y) + " orient=" + to\_string(orient) + "]";

    }

    string read\_sensors()

    {

        // Returns all the distances from the robot toward the landmarks

        vector<double> z = sense();

        string readings = "[";

        for (int i = 0; i < z.size(); i++) {

            readings += to\_string(z[i]) + " ";

        }

        readings[readings.size() - 1] = ']';

        return readings;

    }

    double measurement\_prob(vector<double> measurement)

    {

        // Calculates how likely a measurement should be

        double prob = 1.0;

        double dist;

        for (int i = 0; i < sizeof(landmarks) / sizeof(landmarks[0]); i++) {

            dist = sqrt(pow((x - landmarks[i][0]), 2) + pow((y - landmarks[i][1]), 2));

            prob \*= gaussian(dist, sense\_noise, measurement[i]);

        }

        return prob;

    }

    double x, y, orient; //robot poses

    double forward\_noise, turn\_noise, sense\_noise; //robot noises

private:

    double gen\_gauss\_random(double mean, double variance)

    {

        // Gaussian random

        normal\_distribution<double> gauss\_dist(mean, variance);

        return gauss\_dist(gen);

    }

    double gaussian(double mu, double sigma, double x)

    {

        // Probability of x for 1-dim Gaussian with mean mu and var. sigma

        return exp(-(pow((mu - x), 2)) / (pow(sigma, 2)) / 2.0) / sqrt(2.0 \* M\_PI \* (pow(sigma, 2)));

    }

};

// Functions

double gen\_real\_random()

{

    // Generate real random between 0 and 1

    uniform\_real\_distribution<double> real\_dist(0.0, 1.0); //Real

    return real\_dist(gen);

}

double mod(double first\_term, double second\_term)

{

    // Compute the modulus

    return first\_term - (second\_term)\*floor(first\_term / (second\_term));

}

double evaluation(Robot r, Robot p[], int n)

{

    //Calculate the mean error of the system

    double sum = 0.0;

    for (int i = 0; i < n; i++) {

        //the second part is because of world's cyclicity

        double dx = mod((p[i].x - r.x + (world\_size / 2.0)), world\_size) - (world\_size / 2.0);

        double dy = mod((p[i].y - r.y + (world\_size / 2.0)), world\_size) - (world\_size / 2.0);

        double err = sqrt(pow(dx, 2) + pow(dy, 2));

        sum += err;

    }

    return sum / n;

}

double max(double arr[], int n)

{

    // Identify the max element in an array

    double max = 0;

    for (int i = 0; i < n; i++) {

        if (arr[i] > max)

            max = arr[i];

    }

    return max;

}

/\*

void visualization(int n, Robot robot, int step, Robot p[], Robot pr[])

{

    //Draw the robot, landmarks, particles and resampled particles on a graph

    //Graph Format

    plt::title("MCL, step " + to\_string(step));

    plt::xlim(0, 100);

    plt::ylim(0, 100);

    //Draw particles in green

    for (int i = 0; i < n; i++) {

        plt::plot({ p[i].x }, { p[i].y }, "go");

    }

    //Draw resampled particles in yellow

    for (int i = 0; i < n; i++) {

        plt::plot({ pr[i].x }, { pr[i].y }, "yo");

    }

    //Draw landmarks in red

    for (int i = 0; i < sizeof(landmarks) / sizeof(landmarks[0]); i++) {

        plt::plot({ landmarks[i][0] }, { landmarks[i][1] }, "ro");

    }

    //Draw robot position in blue

    plt::plot({ robot.x }, { robot.y }, "bo");

    //Save the image and close the plot

    plt::save("./Images/Step" + to\_string(step) + ".png");

    plt::clf();

}

\*/

int main()

{

    //Practice Interfacing with Robot Class

    Robot myrobot;

    myrobot.set\_noise(5.0, 0.1, 5.0);

    myrobot.set(30.0, 50.0, M\_PI / 2.0);

    myrobot.move(-M\_PI / 2.0, 15.0);

    //cout << myrobot.read\_sensors() << endl;

    myrobot.move(-M\_PI / 2.0, 10.0);

    //cout << myrobot.read\_sensors() << endl;

    // Create a set of particles

    int n = 1000;

    Robot p[n];

    for (int i = 0; i < n; i++) {

        p[i].set\_noise(0.05, 0.05, 5.0);

        //cout << p[i].show\_pose() << endl;

    }

    //Re-initialize myrobot object and Initialize a measurment vector

    myrobot = Robot();

    vector<double> z;

    //Move the robot and sense the environment afterwards

    myrobot = myrobot.move(0.1, 5.0);

    z = myrobot.sense();

    // Simulate a robot motion for each of these particles

    Robot p2[n];

    for (int i = 0; i < n; i++) {

        p2[i] = p[i].move(0.1, 5.0);

        p[i] = p2[i];

    }

    //####   DON'T MODIFY ANYTHING ABOVE HERE! ENTER CODE BELOW ####

    //TODO: Generate particle weights depending on robot's measurement

    //TODO: Print particle weights, each on a single line

    double w[n];

    return 0;

}

#### solution.cpp

//#include "src/matplotlibcpp.h"//Graph Library

#include <iostream>

#include <string>

#include <math.h>

#include <vector>

#include <stdexcept> // throw errors

#include <random> //C++ 11 Random Numbers

//namespace plt = matplotlibcpp;

using namespace std;

// Landmarks

double landmarks[8][2] = { { 20.0, 20.0 }, { 20.0, 80.0 }, { 20.0, 50.0 },

    { 50.0, 20.0 }, { 50.0, 80.0 }, { 80.0, 80.0 },

    { 80.0, 20.0 }, { 80.0, 50.0 } };

// Map size in meters

double world\_size = 100.0;

// Random Generators

random\_device rd;

mt19937 gen(rd());

// Global Functions

double mod(double first\_term, double second\_term);

double gen\_real\_random();

class Robot {

public:

    Robot()

    {

        // Constructor

        x = gen\_real\_random() \* world\_size; // robot's x coordinate

        y = gen\_real\_random() \* world\_size; // robot's y coordinate

        orient = gen\_real\_random() \* 2.0 \* M\_PI; // robot's orientation

        forward\_noise = 0.0; //noise of the forward movement

        turn\_noise = 0.0; //noise of the turn

        sense\_noise = 0.0; //noise of the sensing

    }

    void set(double new\_x, double new\_y, double new\_orient)

    {

        // Set robot new position and orientation

        if (new\_x < 0 || new\_x >= world\_size)

            throw std::invalid\_argument("X coordinate out of bound");

        if (new\_y < 0 || new\_y >= world\_size)

            throw std::invalid\_argument("Y coordinate out of bound");

        if (new\_orient < 0 || new\_orient >= 2 \* M\_PI)

            throw std::invalid\_argument("Orientation must be in [0..2pi]");

        x = new\_x;

        y = new\_y;

        orient = new\_orient;

    }

    void set\_noise(double new\_forward\_noise, double new\_turn\_noise, double new\_sense\_noise)

    {

        // Simulate noise, often useful in particle filters

        forward\_noise = new\_forward\_noise;

        turn\_noise = new\_turn\_noise;

        sense\_noise = new\_sense\_noise;

    }

    vector<double> sense()

    {

        // Measure the distances from the robot toward the landmarks

        vector<double> z(sizeof(landmarks) / sizeof(landmarks[0]));

        double dist;

        for (int i = 0; i < sizeof(landmarks) / sizeof(landmarks[0]); i++) {

            dist = sqrt(pow((x - landmarks[i][0]), 2) + pow((y - landmarks[i][1]), 2));

            dist += gen\_gauss\_random(0.0, sense\_noise);

            z[i] = dist;

        }

        return z;

    }

    Robot move(double turn, double forward)

    {

        if (forward < 0)

            throw std::invalid\_argument("Robot cannot move backward");

        // turn, and add randomness to the turning command

        orient = orient + turn + gen\_gauss\_random(0.0, turn\_noise);

        orient = mod(orient, 2 \* M\_PI);

        // move, and add randomness to the motion command

        double dist = forward + gen\_gauss\_random(0.0, forward\_noise);

        x = x + (cos(orient) \* dist);

        y = y + (sin(orient) \* dist);

        // cyclic truncate

        x = mod(x, world\_size);

        y = mod(y, world\_size);

        // set particle

        Robot res;

        res.set(x, y, orient);

        res.set\_noise(forward\_noise, turn\_noise, sense\_noise);

        return res;

    }

    string show\_pose()

    {

        // Returns the robot current position and orientation in a string format

        return "[x=" + to\_string(x) + " y=" + to\_string(y) + " orient=" + to\_string(orient) + "]";

    }

    string read\_sensors()

    {

        // Returns all the distances from the robot toward the landmarks

        vector<double> z = sense();

        string readings = "[";

        for (int i = 0; i < z.size(); i++) {

            readings += to\_string(z[i]) + " ";

        }

        readings[readings.size() - 1] = ']';

        return readings;

    }

    double measurement\_prob(vector<double> measurement)

    {

        // Calculates how likely a measurement should be

        double prob = 1.0;

        double dist;

        for (int i = 0; i < sizeof(landmarks) / sizeof(landmarks[0]); i++) {

            dist = sqrt(pow((x - landmarks[i][0]), 2) + pow((y - landmarks[i][1]), 2));

            prob \*= gaussian(dist, sense\_noise, measurement[i]);

        }

        return prob;

    }

    double x, y, orient; //robot poses

    double forward\_noise, turn\_noise, sense\_noise; //robot noises

private:

    double gen\_gauss\_random(double mean, double variance)

    {

        // Gaussian random

        normal\_distribution<double> gauss\_dist(mean, variance);

        return gauss\_dist(gen);

    }

    double gaussian(double mu, double sigma, double x)

    {

        // Probability of x for 1-dim Gaussian with mean mu and var. sigma

        return exp(-(pow((mu - x), 2)) / (pow(sigma, 2)) / 2.0) / sqrt(2.0 \* M\_PI \* (pow(sigma, 2)));

    }

};

// Functions

double gen\_real\_random()

{

    // Generate real random between 0 and 1

    uniform\_real\_distribution<double> real\_dist(0.0, 1.0); //Real

    return real\_dist(gen);

}

double mod(double first\_term, double second\_term)

{

    // Compute the modulus

    return first\_term - (second\_term)\*floor(first\_term / (second\_term));

}

double evaluation(Robot r, Robot p[], int n)

{

    //Calculate the mean error of the system

    double sum = 0.0;

    for (int i = 0; i < n; i++) {

        //the second part is because of world's cyclicity

        double dx = mod((p[i].x - r.x + (world\_size / 2.0)), world\_size) - (world\_size / 2.0);

        double dy = mod((p[i].y - r.y + (world\_size / 2.0)), world\_size) - (world\_size / 2.0);

        double err = sqrt(pow(dx, 2) + pow(dy, 2));

        sum += err;

    }

    return sum / n;

}

double max(double arr[], int n)

{

    // Identify the max element in an array

    double max = 0;

    for (int i = 0; i < n; i++) {

        if (arr[i] > max)

            max = arr[i];

    }

    return max;

}

/\*

void visualization(int n, Robot robot, int step, Robot p[], Robot pr[])

{

    //Draw the robot, landmarks, particles and resampled particles on a graph

    //Graph Format

    plt::title("MCL, step " + to\_string(step));

    plt::xlim(0, 100);

    plt::ylim(0, 100);

    //Draw particles in green

    for (int i = 0; i < n; i++) {

        plt::plot({ p[i].x }, { p[i].y }, "go");

    }

    //Draw resampled particles in yellow

    for (int i = 0; i < n; i++) {

        plt::plot({ pr[i].x }, { pr[i].y }, "yo");

    }

    //Draw landmarks in red

    for (int i = 0; i < sizeof(landmarks) / sizeof(landmarks[0]); i++) {

        plt::plot({ landmarks[i][0] }, { landmarks[i][1] }, "ro");

    }

    //Draw robot position in blue

    plt::plot({ robot.x }, { robot.y }, "bo");

    //Save the image and close the plot

    plt::save("./Images/Step" + to\_string(step) + ".png");

    plt::clf();

}

\*/

int main()

{

    //Practice Interfacing with Robot Class

    Robot myrobot;

    myrobot.set\_noise(5.0, 0.1, 5.0);

    myrobot.set(30.0, 50.0, M\_PI / 2.0);

    myrobot.move(-M\_PI / 2.0, 15.0);

    //cout << myrobot.read\_sensors() << endl;

    myrobot.move(-M\_PI / 2.0, 10.0);

    //cout << myrobot.read\_sensors() << endl;

    // Create a set of particles

    int n = 1000;

    Robot p[n];

    for (int i = 0; i < n; i++) {

        p[i].set\_noise(0.05, 0.05, 5.0);

        //cout << p[i].show\_pose() << endl;

    }

    //Re-initialize myrobot object and Initialize a measurment vector

    myrobot = Robot();

    vector<double> z;

    //Move the robot and sense the environment afterwards

    myrobot = myrobot.move(0.1, 5.0);

    z = myrobot.sense();

    // Simulate a robot motion for each of these particles

    Robot p2[n];

    for (int i = 0; i < n; i++) {

        p2[i] = p[i].move(0.1, 5.0);

        p[i] = p2[i];

    }

    //####   DON'T MODIFY ANYTHING ABOVE HERE! ENTER CODE BELOW ####

    //Generate particle weights depending on robot's measurement

    //Print particle weights, each on a single line

    double w[n];

    for (int i = 0; i < n; i++) {

        w[i] = p[i].measurement\_prob(z);

        cout << w[i] << endl;

    }

    return 0;

}

### Resampling

Video 5.8

A screenshot of a computer

AI-generated content may be incorrect.

**Resampling**

Suppose that you have 5 particles, each with an importance weight. **Compute** the probability of drawing each particle in the new set. Use the C++ coding section and follow the instructions to **print** the results.

// main.cpp

#include <iostream>

using namespace std;

double w[] = { 0.6, 1.2, 2.4, 0.6, 1.2 };//You can also change this to a vector

//TODO: Define a  ComputeProb function and compute the Probabilities

int main()

{

    //TODO: Print Probabilites each on a single line:

    //P1=Value

    //:

    //P5=Value

    return 0;

}

// solution.cpp

//One out of many possible solutions

#include <iostream>

using namespace std;

double w[] = { 0.6, 1.2, 2.4, 0.6, 1.2 };

double sum = 0;

void ComputeProb(double w[], int n)

{

    for (int i = 0; i < n; i++) {

        sum = sum + w[i];

    }

    for (int j = 0; j < n; j++) {

        w[j] = w[j] / sum;

        cout << "P" << j + 1 << "=" << w[j] << endl;

    }

}

int main()

{

    ComputeProb(w, sizeof(w) / sizeof(w[0]));

    return 0;

}

### Resampling Wheel

Video 5.9

Following the video, let's break down how particles are chosen in the **resampling wheel process**, using a set of 8 particles, each with an associated weight.

* Each particle represents a possible position for a robot based on prior movements and sensor readings.
* Each particle's weight reflects how likely that position is to be the robot’s actual location. These particles are visualized arranged in a **circular wheel**, where the size of each particle's segment corresponds to its weight: particles with smaller weights (like Particles 1 and 3 in the video) occupy smaller portions, while particles with higher weights (such as Particle 4) occupy larger portions of the wheel.
* This arrangement emphasizes that particles with higher weights have a greater chance of being selected, as they represent more probable locations.

To begin the resampling, a random starting index (from 1 to N) is chosen, let’s say Particle 6, and we initialize a "selection marker" (denoted by beta) to zero. In each iteration, we add a random offset to beta, ranging from 0 to twice the highest weight (2 *W\_4). This offset helps simulate* *spinning the wheel*\* and introduces a probabilistic element, favoring particles with larger weights.

We then compare beta to the current particle’s weight. If beta exceeds that weight, it means we haven’t yet reached a likely candidate, so we reduce beta by the particle's weight and move to the next particle. This process continues until we find a particle with a weight larger than the remaining beta, at which point we "select" that particle, effectively concentrating on the higher-weight (more probable) particles. Over multiple iterations, this method yields a new set of particles with a distribution that represents the robot’s most likely positions, based on the updated probability data.

This resampling method reduces computational complexity and shifts resources toward probable areas, improving localization performance. For more about the resampling processing, refer to this [**video**](https://youtu.be/MsYlueVDLI0?t=2472).

A screenshot of a computer program

AI-generated content may be incorrect.

**Resampling Wheel**

Now that you’ve learned the resampling wheel pseudocode, try implementing it in C++. In this quiz, **resample** the particles based on a probability proportional to their importance weights. Finally, **print the pose of each resampled particle on a single line** to match the expected output format and pass the quiz.

#### main.cpp

//#include "src/matplotlibcpp.h"//Graph Library

#include <iostream>

#include <string>

#include <math.h>

#include <vector>

#include <stdexcept> // throw errors

#include <random> //C++ 11 Random Numbers

//namespace plt = matplotlibcpp;

using namespace std;

// Landmarks

double landmarks[8][2] = { { 20.0, 20.0 }, { 20.0, 80.0 }, { 20.0, 50.0 },

    { 50.0, 20.0 }, { 50.0, 80.0 }, { 80.0, 80.0 },

    { 80.0, 20.0 }, { 80.0, 50.0 } };

// Map size in meters

double world\_size = 100.0;

// Random Generators

random\_device rd;

mt19937 gen(rd());

// Global Functions

double mod(double first\_term, double second\_term);

double gen\_real\_random();

class Robot {

public:

    Robot()

    {

        // Constructor

        x = gen\_real\_random() \* world\_size; // robot's x coordinate

        y = gen\_real\_random() \* world\_size; // robot's y coordinate

        orient = gen\_real\_random() \* 2.0 \* M\_PI; // robot's orientation

        forward\_noise = 0.0; //noise of the forward movement

        turn\_noise = 0.0; //noise of the turn

        sense\_noise = 0.0; //noise of the sensing

    }

    void set(double new\_x, double new\_y, double new\_orient)

    {

        // Set robot new position and orientation

        if (new\_x < 0 || new\_x >= world\_size)

            throw std::invalid\_argument("X coordinate out of bound");

        if (new\_y < 0 || new\_y >= world\_size)

            throw std::invalid\_argument("Y coordinate out of bound");

        if (new\_orient < 0 || new\_orient >= 2 \* M\_PI)

            throw std::invalid\_argument("Orientation must be in [0..2pi]");

        x = new\_x;

        y = new\_y;

        orient = new\_orient;

    }

    void set\_noise(double new\_forward\_noise, double new\_turn\_noise, double new\_sense\_noise)

    {

        // Simulate noise, often useful in particle filters

        forward\_noise = new\_forward\_noise;

        turn\_noise = new\_turn\_noise;

        sense\_noise = new\_sense\_noise;

    }

    vector<double> sense()

    {

        // Measure the distances from the robot toward the landmarks

        vector<double> z(sizeof(landmarks) / sizeof(landmarks[0]));

        double dist;

        for (int i = 0; i < sizeof(landmarks) / sizeof(landmarks[0]); i++) {

            dist = sqrt(pow((x - landmarks[i][0]), 2) + pow((y - landmarks[i][1]), 2));

            dist += gen\_gauss\_random(0.0, sense\_noise);

            z[i] = dist;

        }

        return z;

    }

    Robot move(double turn, double forward)

    {

        if (forward < 0)

            throw std::invalid\_argument("Robot cannot move backward");

        // turn, and add randomness to the turning command

        orient = orient + turn + gen\_gauss\_random(0.0, turn\_noise);

        orient = mod(orient, 2 \* M\_PI);

        // move, and add randomness to the motion command

        double dist = forward + gen\_gauss\_random(0.0, forward\_noise);

        x = x + (cos(orient) \* dist);

        y = y + (sin(orient) \* dist);

        // cyclic truncate

        x = mod(x, world\_size);

        y = mod(y, world\_size);

        // set particle

        Robot res;

        res.set(x, y, orient);

        res.set\_noise(forward\_noise, turn\_noise, sense\_noise);

        return res;

    }

    string show\_pose()

    {

        // Returns the robot current position and orientation in a string format

        return "[x=" + to\_string(x) + " y=" + to\_string(y) + " orient=" + to\_string(orient) + "]";

    }

    string read\_sensors()

    {

        // Returns all the distances from the robot toward the landmarks

        vector<double> z = sense();

        string readings = "[";

        for (int i = 0; i < z.size(); i++) {

            readings += to\_string(z[i]) + " ";

        }

        readings[readings.size() - 1] = ']';

        return readings;

    }

    double measurement\_prob(vector<double> measurement)

    {

        // Calculates how likely a measurement should be

        double prob = 1.0;

        double dist;

        for (int i = 0; i < sizeof(landmarks) / sizeof(landmarks[0]); i++) {

            dist = sqrt(pow((x - landmarks[i][0]), 2) + pow((y - landmarks[i][1]), 2));

            prob \*= gaussian(dist, sense\_noise, measurement[i]);

        }

        return prob;

    }

    double x, y, orient; //robot poses

    double forward\_noise, turn\_noise, sense\_noise; //robot noises

private:

    double gen\_gauss\_random(double mean, double variance)

    {

        // Gaussian random

        normal\_distribution<double> gauss\_dist(mean, variance);

        return gauss\_dist(gen);

    }

    double gaussian(double mu, double sigma, double x)

    {

        // Probability of x for 1-dim Gaussian with mean mu and var. sigma

        return exp(-(pow((mu - x), 2)) / (pow(sigma, 2)) / 2.0) / sqrt(2.0 \* M\_PI \* (pow(sigma, 2)));

    }

};

// Functions

double gen\_real\_random()

{

    // Generate real random between 0 and 1

    uniform\_real\_distribution<double> real\_dist(0.0, 1.0); //Real

    return real\_dist(gen);

}

double mod(double first\_term, double second\_term)

{

    // Compute the modulus

    return first\_term - (second\_term)\*floor(first\_term / (second\_term));

}

double evaluation(Robot r, Robot p[], int n)

{

    //Calculate the mean error of the system

    double sum = 0.0;

    for (int i = 0; i < n; i++) {

        //the second part is because of world's cyclicity

        double dx = mod((p[i].x - r.x + (world\_size / 2.0)), world\_size) - (world\_size / 2.0);

        double dy = mod((p[i].y - r.y + (world\_size / 2.0)), world\_size) - (world\_size / 2.0);

        double err = sqrt(pow(dx, 2) + pow(dy, 2));

        sum += err;

    }

    return sum / n;

}

double max(double arr[], int n)

{

    // Identify the max element in an array

    double max = 0;

    for (int i = 0; i < n; i++) {

        if (arr[i] > max)

            max = arr[i];

    }

    return max;

}

/\*

void visualization(int n, Robot robot, int step, Robot p[], Robot pr[])

{

    //Draw the robot, landmarks, particles and resampled particles on a graph

    //Graph Format

    plt::title("MCL, step " + to\_string(step));

    plt::xlim(0, 100);

    plt::ylim(0, 100);

    //Draw particles in green

    for (int i = 0; i < n; i++) {

        plt::plot({ p[i].x }, { p[i].y }, "go");

    }

    //Draw resampled particles in yellow

    for (int i = 0; i < n; i++) {

        plt::plot({ pr[i].x }, { pr[i].y }, "yo");

    }

    //Draw landmarks in red

    for (int i = 0; i < sizeof(landmarks) / sizeof(landmarks[0]); i++) {

        plt::plot({ landmarks[i][0] }, { landmarks[i][1] }, "ro");

    }

    //Draw robot position in blue

    plt::plot({ robot.x }, { robot.y }, "bo");

    //Save the image and close the plot

    plt::save("./Images/Step" + to\_string(step) + ".png");

    plt::clf();

}

\*/

int main()

{

    //Practice Interfacing with Robot Class

    Robot myrobot;

    myrobot.set\_noise(5.0, 0.1, 5.0);

    myrobot.set(30.0, 50.0, M\_PI / 2.0);

    myrobot.move(-M\_PI / 2.0, 15.0);

    //cout << myrobot.read\_sensors() << endl;

    myrobot.move(-M\_PI / 2.0, 10.0);

    //cout << myrobot.read\_sensors() << endl;

    // Create a set of particles

    int n = 1000;

    Robot p[n];

    for (int i = 0; i < n; i++) {

        p[i].set\_noise(0.05, 0.05, 5.0);

        //cout << p[i].show\_pose() << endl;

    }

    //Re-initialize myrobot object and Initialize a measurment vector

    myrobot = Robot();

    vector<double> z;

    //Move the robot and sense the environment afterwards

    myrobot = myrobot.move(0.1, 5.0);

    z = myrobot.sense();

    // Simulate a robot motion for each of these particles

    Robot p2[n];

    for (int i = 0; i < n; i++) {

        p2[i] = p[i].move(0.1, 5.0);

        p[i] = p2[i];

    }

    //Generate particle weights depending on robot's measurement

    double w[n];

    for (int i = 0; i < n; i++) {

        w[i] = p[i].measurement\_prob(z);

        //cout << w[i] << endl;

    }

    //####   DON'T MODIFY ANYTHING ABOVE HERE! ENTER CODE BELOW ####

    //TODO: Resample the particles with a sample probability proportional to the importance weight

    return 0;

}

#### solution.cpp

//#include "src/matplotlibcpp.h"//Graph Library

#include <iostream>

#include <string>

#include <math.h>

#include <vector>

#include <stdexcept> // throw errors

#include <random> //C++ 11 Random Numbers

//namespace plt = matplotlibcpp;

using namespace std;

// Landmarks

double landmarks[8][2] = { { 20.0, 20.0 }, { 20.0, 80.0 }, { 20.0, 50.0 },

    { 50.0, 20.0 }, { 50.0, 80.0 }, { 80.0, 80.0 },

    { 80.0, 20.0 }, { 80.0, 50.0 } };

// Map size in meters

double world\_size = 100.0;

// Random Generators

random\_device rd;

mt19937 gen(rd());

// Global Functions

double mod(double first\_term, double second\_term);

double gen\_real\_random();

class Robot {

public:

    Robot()

    {

        // Constructor

        x = gen\_real\_random() \* world\_size; // robot's x coordinate

        y = gen\_real\_random() \* world\_size; // robot's y coordinate

        orient = gen\_real\_random() \* 2.0 \* M\_PI; // robot's orientation

        forward\_noise = 0.0; //noise of the forward movement

        turn\_noise = 0.0; //noise of the turn

        sense\_noise = 0.0; //noise of the sensing

    }

    void set(double new\_x, double new\_y, double new\_orient)

    {

        // Set robot new position and orientation

        if (new\_x < 0 || new\_x >= world\_size)

            throw std::invalid\_argument("X coordinate out of bound");

        if (new\_y < 0 || new\_y >= world\_size)

            throw std::invalid\_argument("Y coordinate out of bound");

        if (new\_orient < 0 || new\_orient >= 2 \* M\_PI)

            throw std::invalid\_argument("Orientation must be in [0..2pi]");

        x = new\_x;

        y = new\_y;

        orient = new\_orient;

    }

    void set\_noise(double new\_forward\_noise, double new\_turn\_noise, double new\_sense\_noise)

    {

        // Simulate noise, often useful in particle filters

        forward\_noise = new\_forward\_noise;

        turn\_noise = new\_turn\_noise;

        sense\_noise = new\_sense\_noise;

    }

    vector<double> sense()

    {

        // Measure the distances from the robot toward the landmarks

        vector<double> z(sizeof(landmarks) / sizeof(landmarks[0]));

        double dist;

        for (int i = 0; i < sizeof(landmarks) / sizeof(landmarks[0]); i++) {

            dist = sqrt(pow((x - landmarks[i][0]), 2) + pow((y - landmarks[i][1]), 2));

            dist += gen\_gauss\_random(0.0, sense\_noise);

            z[i] = dist;

        }

        return z;

    }

    Robot move(double turn, double forward)

    {

        if (forward < 0)

            throw std::invalid\_argument("Robot cannot move backward");

        // turn, and add randomness to the turning command

        orient = orient + turn + gen\_gauss\_random(0.0, turn\_noise);

        orient = mod(orient, 2 \* M\_PI);

        // move, and add randomness to the motion command

        double dist = forward + gen\_gauss\_random(0.0, forward\_noise);

        x = x + (cos(orient) \* dist);

        y = y + (sin(orient) \* dist);

        // cyclic truncate

        x = mod(x, world\_size);

        y = mod(y, world\_size);

        // set particle

        Robot res;

        res.set(x, y, orient);

        res.set\_noise(forward\_noise, turn\_noise, sense\_noise);

        return res;

    }

    string show\_pose()

    {

        // Returns the robot current position and orientation in a string format

        return "[x=" + to\_string(x) + " y=" + to\_string(y) + " orient=" + to\_string(orient) + "]";

    }

    string read\_sensors()

    {

        // Returns all the distances from the robot toward the landmarks

        vector<double> z = sense();

        string readings = "[";

        for (int i = 0; i < z.size(); i++) {

            readings += to\_string(z[i]) + " ";

        }

        readings[readings.size() - 1] = ']';

        return readings;

    }

    double measurement\_prob(vector<double> measurement)

    {

        // Calculates how likely a measurement should be

        double prob = 1.0;

        double dist;

        for (int i = 0; i < sizeof(landmarks) / sizeof(landmarks[0]); i++) {

            dist = sqrt(pow((x - landmarks[i][0]), 2) + pow((y - landmarks[i][1]), 2));

            prob \*= gaussian(dist, sense\_noise, measurement[i]);

        }

        return prob;

    }

    double x, y, orient; //robot poses

    double forward\_noise, turn\_noise, sense\_noise; //robot noises

private:

    double gen\_gauss\_random(double mean, double variance)

    {

        // Gaussian random

        normal\_distribution<double> gauss\_dist(mean, variance);

        return gauss\_dist(gen);

    }

    double gaussian(double mu, double sigma, double x)

    {

        // Probability of x for 1-dim Gaussian with mean mu and var. sigma

        return exp(-(pow((mu - x), 2)) / (pow(sigma, 2)) / 2.0) / sqrt(2.0 \* M\_PI \* (pow(sigma, 2)));

    }

};

// Functions

double gen\_real\_random()

{

    // Generate real random between 0 and 1

    uniform\_real\_distribution<double> real\_dist(0.0, 1.0); //Real

    return real\_dist(gen);

}

double mod(double first\_term, double second\_term)

{

    // Compute the modulus

    return first\_term - (second\_term)\*floor(first\_term / (second\_term));

}

double evaluation(Robot r, Robot p[], int n)

{

    //Calculate the mean error of the system

    double sum = 0.0;

    for (int i = 0; i < n; i++) {

        //the second part is because of world's cyclicity

        double dx = mod((p[i].x - r.x + (world\_size / 2.0)), world\_size) - (world\_size / 2.0);

        double dy = mod((p[i].y - r.y + (world\_size / 2.0)), world\_size) - (world\_size / 2.0);

        double err = sqrt(pow(dx, 2) + pow(dy, 2));

        sum += err;

    }

    return sum / n;

}

double max(double arr[], int n)

{

    // Identify the max element in an array

    double max = 0;

    for (int i = 0; i < n; i++) {

        if (arr[i] > max)

            max = arr[i];

    }

    return max;

}

/\*

void visualization(int n, Robot robot, int step, Robot p[], Robot pr[])

{

    //Draw the robot, landmarks, particles and resampled particles on a graph

    //Graph Format

    plt::title("MCL, step " + to\_string(step));

    plt::xlim(0, 100);

    plt::ylim(0, 100);

    //Draw particles in green

    for (int i = 0; i < n; i++) {

        plt::plot({ p[i].x }, { p[i].y }, "go");

    }

    //Draw resampled particles in yellow

    for (int i = 0; i < n; i++) {

        plt::plot({ pr[i].x }, { pr[i].y }, "yo");

    }

    //Draw landmarks in red

    for (int i = 0; i < sizeof(landmarks) / sizeof(landmarks[0]); i++) {

        plt::plot({ landmarks[i][0] }, { landmarks[i][1] }, "ro");

    }

    //Draw robot position in blue

    plt::plot({ robot.x }, { robot.y }, "bo");

    //Save the image and close the plot

    plt::save("./Images/Step" + to\_string(step) + ".png");

    plt::clf();

}

\*/

int main()

{

    //Practice Interfacing with Robot Class

    Robot myrobot;

    myrobot.set\_noise(5.0, 0.1, 5.0);

    myrobot.set(30.0, 50.0, M\_PI / 2.0);

    myrobot.move(-M\_PI / 2.0, 15.0);

    //cout << myrobot.read\_sensors() << endl;

    myrobot.move(-M\_PI / 2.0, 10.0);

    //cout << myrobot.read\_sensors() << endl;

    // Create a set of particles

    int n = 1000;

    Robot p[n];

    for (int i = 0; i < n; i++) {

        p[i].set\_noise(0.05, 0.05, 5.0);

        //cout << p[i].show\_pose() << endl;

    }

    //Re-initialize myrobot object and Initialize a measurment vector

    myrobot = Robot();

    vector<double> z;

    //Move the robot and sense the environment afterwards

    myrobot = myrobot.move(0.1, 5.0);

    z = myrobot.sense();

    // Simulate a robot motion for each of these particles

    Robot p2[n];

    for (int i = 0; i < n; i++) {

        p2[i] = p[i].move(0.1, 5.0);

        p[i] = p2[i];

    }

    //Generate particle weights depending on robot's measurement

    double w[n];

    for (int i = 0; i < n; i++) {

        w[i] = p[i].measurement\_prob(z);

        //cout << w[i] << endl;

    }

    //####   DON'T MODIFY ANYTHING ABOVE HERE! ENTER CODE BELOW ####

    //Resample the particles with a sample probability proportional to the importance weight

    Robot p3[n];

    int index = gen\_real\_random() \* n;

    //cout << index << endl;

    double beta = 0.0;

    double mw = max(w, n);

    //cout << mw;

    for (int i = 0; i < n; i++) {

        beta += gen\_real\_random() \* 2.0 \* mw;

        while (beta > w[index]) {

            beta -= w[index];

            index = mod((index + 1), n);

        }

        p3[i] = p[index];

    }

    for (int k=0; k < n; k++) {

        p[k] = p3[k];

        cout << p[k].show\_pose() << endl;

    }

    return 0;

}

### Error

Video 5.10

**Error**

You’ve just coded the MCL algorithm, and now you should evaluate the overall quality of your solution. To do so, you’ll need to compute the average distance between the particles and the robot. A good solution will result in an average distance smaller than a meter. Now, use the evaluation function and compute the average distance, or error at each iteration.

Each number generated denotes the **average distance** between the particles and the robot in a world of 100mx100m. Notice how the number starts relatively high and converges to a smaller number after several iterations.

#### main.cpp

//#include "src/matplotlibcpp.h"//Graph Library

#include <iostream>

#include <string>

#include <math.h>

#include <vector>

#include <stdexcept> // throw errors

#include <random> //C++ 11 Random Numbers

//namespace plt = matplotlibcpp;

using namespace std;

// Landmarks

double landmarks[8][2] = { { 20.0, 20.0 }, { 20.0, 80.0 }, { 20.0, 50.0 },

    { 50.0, 20.0 }, { 50.0, 80.0 }, { 80.0, 80.0 },

    { 80.0, 20.0 }, { 80.0, 50.0 } };

// Map size in meters

double world\_size = 100.0;

// Random Generators

random\_device rd;

mt19937 gen(rd());

// Global Functions

double mod(double first\_term, double second\_term);

double gen\_real\_random();

class Robot {

public:

    Robot()

    {

        // Constructor

        x = gen\_real\_random() \* world\_size; // robot's x coordinate

        y = gen\_real\_random() \* world\_size; // robot's y coordinate

        orient = gen\_real\_random() \* 2.0 \* M\_PI; // robot's orientation

        forward\_noise = 0.0; //noise of the forward movement

        turn\_noise = 0.0; //noise of the turn

        sense\_noise = 0.0; //noise of the sensing

    }

    void set(double new\_x, double new\_y, double new\_orient)

    {

        // Set robot new position and orientation

        if (new\_x < 0 || new\_x >= world\_size)

            throw std::invalid\_argument("X coordinate out of bound");

        if (new\_y < 0 || new\_y >= world\_size)

            throw std::invalid\_argument("Y coordinate out of bound");

        if (new\_orient < 0 || new\_orient >= 2 \* M\_PI)

            throw std::invalid\_argument("Orientation must be in [0..2pi]");

        x = new\_x;

        y = new\_y;

        orient = new\_orient;

    }

    void set\_noise(double new\_forward\_noise, double new\_turn\_noise, double new\_sense\_noise)

    {

        // Simulate noise, often useful in particle filters

        forward\_noise = new\_forward\_noise;

        turn\_noise = new\_turn\_noise;

        sense\_noise = new\_sense\_noise;

    }

    vector<double> sense()

    {

        // Measure the distances from the robot toward the landmarks

        vector<double> z(sizeof(landmarks) / sizeof(landmarks[0]));

        double dist;

        for (int i = 0; i < sizeof(landmarks) / sizeof(landmarks[0]); i++) {

            dist = sqrt(pow((x - landmarks[i][0]), 2) + pow((y - landmarks[i][1]), 2));

            dist += gen\_gauss\_random(0.0, sense\_noise);

            z[i] = dist;

        }

        return z;

    }

    Robot move(double turn, double forward)

    {

        if (forward < 0)

            throw std::invalid\_argument("Robot cannot move backward");

        // turn, and add randomness to the turning command

        orient = orient + turn + gen\_gauss\_random(0.0, turn\_noise);

        orient = mod(orient, 2 \* M\_PI);

        // move, and add randomness to the motion command

        double dist = forward + gen\_gauss\_random(0.0, forward\_noise);

        x = x + (cos(orient) \* dist);

        y = y + (sin(orient) \* dist);

        // cyclic truncate

        x = mod(x, world\_size);

        y = mod(y, world\_size);

        // set particle

        Robot res;

        res.set(x, y, orient);

        res.set\_noise(forward\_noise, turn\_noise, sense\_noise);

        return res;

    }

    string show\_pose()

    {

        // Returns the robot current position and orientation in a string format

        return "[x=" + to\_string(x) + " y=" + to\_string(y) + " orient=" + to\_string(orient) + "]";

    }

    string read\_sensors()

    {

        // Returns all the distances from the robot toward the landmarks

        vector<double> z = sense();

        string readings = "[";

        for (int i = 0; i < z.size(); i++) {

            readings += to\_string(z[i]) + " ";

        }

        readings[readings.size() - 1] = ']';

        return readings;

    }

    double measurement\_prob(vector<double> measurement)

    {

        // Calculates how likely a measurement should be

        double prob = 1.0;

        double dist;

        for (int i = 0; i < sizeof(landmarks) / sizeof(landmarks[0]); i++) {

            dist = sqrt(pow((x - landmarks[i][0]), 2) + pow((y - landmarks[i][1]), 2));

            prob \*= gaussian(dist, sense\_noise, measurement[i]);

        }

        return prob;

    }

    double x, y, orient; //robot poses

    double forward\_noise, turn\_noise, sense\_noise; //robot noises

private:

    double gen\_gauss\_random(double mean, double variance)

    {

        // Gaussian random

        normal\_distribution<double> gauss\_dist(mean, variance);

        return gauss\_dist(gen);

    }

    double gaussian(double mu, double sigma, double x)

    {

        // Probability of x for 1-dim Gaussian with mean mu and var. sigma

        return exp(-(pow((mu - x), 2)) / (pow(sigma, 2)) / 2.0) / sqrt(2.0 \* M\_PI \* (pow(sigma, 2)));

    }

};

// Functions

double gen\_real\_random()

{

    // Generate real random between 0 and 1

    uniform\_real\_distribution<double> real\_dist(0.0, 1.0); //Real

    return real\_dist(gen);

}

double mod(double first\_term, double second\_term)

{

    // Compute the modulus

    return first\_term - (second\_term)\*floor(first\_term / (second\_term));

}

double evaluation(Robot r, Robot p[], int n)

{

    //Calculate the mean error of the system

    double sum = 0.0;

    for (int i = 0; i < n; i++) {

        //the second part is because of world's cyclicity

        double dx = mod((p[i].x - r.x + (world\_size / 2.0)), world\_size) - (world\_size / 2.0);

        double dy = mod((p[i].y - r.y + (world\_size / 2.0)), world\_size) - (world\_size / 2.0);

        double err = sqrt(pow(dx, 2) + pow(dy, 2));

        sum += err;

    }

    return sum / n;

}

double max(double arr[], int n)

{

    // Identify the max element in an array

    double max = 0;

    for (int i = 0; i < n; i++) {

        if (arr[i] > max)

            max = arr[i];

    }

    return max;

}

/\*

void visualization(int n, Robot robot, int step, Robot p[], Robot pr[])

{

    //Draw the robot, landmarks, particles and resampled particles on a graph

    //Graph Format

    plt::title("MCL, step " + to\_string(step));

    plt::xlim(0, 100);

    plt::ylim(0, 100);

    //Draw particles in green

    for (int i = 0; i < n; i++) {

        plt::plot({ p[i].x }, { p[i].y }, "go");

    }

    //Draw resampled particles in yellow

    for (int i = 0; i < n; i++) {

        plt::plot({ pr[i].x }, { pr[i].y }, "yo");

    }

    //Draw landmarks in red

    for (int i = 0; i < sizeof(landmarks) / sizeof(landmarks[0]); i++) {

        plt::plot({ landmarks[i][0] }, { landmarks[i][1] }, "ro");

    }

    //Draw robot position in blue

    plt::plot({ robot.x }, { robot.y }, "bo");

    //Save the image and close the plot

    plt::save("./Images/Step" + to\_string(step) + ".png");

    plt::clf();

}

\*/

int main()

{

    //Practice Interfacing with Robot Class

    Robot myrobot;

    myrobot.set\_noise(5.0, 0.1, 5.0);

    myrobot.set(30.0, 50.0, M\_PI / 2.0);

    myrobot.move(-M\_PI / 2.0, 15.0);

    //cout << myrobot.read\_sensors() << endl;

    myrobot.move(-M\_PI / 2.0, 10.0);

    //cout << myrobot.read\_sensors() << endl;

    // Create a set of particles

    int n = 1000;

    Robot p[n];

    for (int i = 0; i < n; i++) {

        p[i].set\_noise(0.05, 0.05, 5.0);

        //cout << p[i].show\_pose() << endl;

    }

    //Re-initialize myrobot object and Initialize a measurment vector

    myrobot = Robot();

    vector<double> z;

    //Iterating 50 times over the set of particles

    int steps = 50;

    for (int t = 0; t < steps; t++) {

        //Move the robot and sense the environment afterwards

        myrobot = myrobot.move(0.1, 5.0);

        z = myrobot.sense();

        // Simulate a robot motion for each of these particles

        Robot p2[n];

        for (int i = 0; i < n; i++) {

            p2[i] = p[i].move(0.1, 5.0);

            p[i] = p2[i];

        }

        //Generate particle weights depending on robot's measurement

        double w[n];

        for (int i = 0; i < n; i++) {

            w[i] = p[i].measurement\_prob(z);

            //cout << w[i] << endl;

        }

        //Resample the particles with a sample probability proportional to the importance weight

        Robot p3[n];

        int index = gen\_real\_random() \* n;

        //cout << index << endl;

        double beta = 0.0;

        double mw = max(w, n);

        //cout << mw;

        for (int i = 0; i < n; i++) {

            beta += gen\_real\_random() \* 2.0 \* mw;

            while (beta > w[index]) {

                beta -= w[index];

                index = mod((index + 1), n);

            }

            p3[i] = p[index];

        }

        for (int k=0; k < n; k++) {

            p[k] = p3[k];

            //cout << p[k].show\_pose() << endl;

        }

        //####   DON'T MODIFY ANYTHING ABOVE HERE! ENTER CODE BELOW ####

        // TODO: Evaluate the error by priting it in this form:

        // cout << "Step = " << t << ", Evaluation = " << ErrorValue << endl;

    } //End of Steps loop

    return 0;

}

#### solution.cpp

//#include "src/matplotlibcpp.h"//Graph Library

#include <iostream>

#include <string>

#include <math.h>

#include <vector>

#include <stdexcept> // throw errors

#include <random> //C++ 11 Random Numbers

//namespace plt = matplotlibcpp;

using namespace std;

// Landmarks

double landmarks[8][2] = { { 20.0, 20.0 }, { 20.0, 80.0 }, { 20.0, 50.0 },

    { 50.0, 20.0 }, { 50.0, 80.0 }, { 80.0, 80.0 },

    { 80.0, 20.0 }, { 80.0, 50.0 } };

// Map size in meters

double world\_size = 100.0;

// Random Generators

random\_device rd;

mt19937 gen(rd());

// Global Functions

double mod(double first\_term, double second\_term);

double gen\_real\_random();

class Robot {

public:

    Robot()

    {

        // Constructor

        x = gen\_real\_random() \* world\_size; // robot's x coordinate

        y = gen\_real\_random() \* world\_size; // robot's y coordinate

        orient = gen\_real\_random() \* 2.0 \* M\_PI; // robot's orientation

        forward\_noise = 0.0; //noise of the forward movement

        turn\_noise = 0.0; //noise of the turn

        sense\_noise = 0.0; //noise of the sensing

    }

    void set(double new\_x, double new\_y, double new\_orient)

    {

        // Set robot new position and orientation

        if (new\_x < 0 || new\_x >= world\_size)

            throw std::invalid\_argument("X coordinate out of bound");

        if (new\_y < 0 || new\_y >= world\_size)

            throw std::invalid\_argument("Y coordinate out of bound");

        if (new\_orient < 0 || new\_orient >= 2 \* M\_PI)

            throw std::invalid\_argument("Orientation must be in [0..2pi]");

        x = new\_x;

        y = new\_y;

        orient = new\_orient;

    }

    void set\_noise(double new\_forward\_noise, double new\_turn\_noise, double new\_sense\_noise)

    {

        // Simulate noise, often useful in particle filters

        forward\_noise = new\_forward\_noise;

        turn\_noise = new\_turn\_noise;

        sense\_noise = new\_sense\_noise;

    }

    vector<double> sense()

    {

        // Measure the distances from the robot toward the landmarks

        vector<double> z(sizeof(landmarks) / sizeof(landmarks[0]));

        double dist;

        for (int i = 0; i < sizeof(landmarks) / sizeof(landmarks[0]); i++) {

            dist = sqrt(pow((x - landmarks[i][0]), 2) + pow((y - landmarks[i][1]), 2));

            dist += gen\_gauss\_random(0.0, sense\_noise);

            z[i] = dist;

        }

        return z;

    }

    Robot move(double turn, double forward)

    {

        if (forward < 0)

            throw std::invalid\_argument("Robot cannot move backward");

        // turn, and add randomness to the turning command

        orient = orient + turn + gen\_gauss\_random(0.0, turn\_noise);

        orient = mod(orient, 2 \* M\_PI);

        // move, and add randomness to the motion command

        double dist = forward + gen\_gauss\_random(0.0, forward\_noise);

        x = x + (cos(orient) \* dist);

        y = y + (sin(orient) \* dist);

        // cyclic truncate

        x = mod(x, world\_size);

        y = mod(y, world\_size);

        // set particle

        Robot res;

        res.set(x, y, orient);

        res.set\_noise(forward\_noise, turn\_noise, sense\_noise);

        return res;

    }

    string show\_pose()

    {

        // Returns the robot current position and orientation in a string format

        return "[x=" + to\_string(x) + " y=" + to\_string(y) + " orient=" + to\_string(orient) + "]";

    }

    string read\_sensors()

    {

        // Returns all the distances from the robot toward the landmarks

        vector<double> z = sense();

        string readings = "[";

        for (int i = 0; i < z.size(); i++) {

            readings += to\_string(z[i]) + " ";

        }

        readings[readings.size() - 1] = ']';

        return readings;

    }

    double measurement\_prob(vector<double> measurement)

    {

        // Calculates how likely a measurement should be

        double prob = 1.0;

        double dist;

        for (int i = 0; i < sizeof(landmarks) / sizeof(landmarks[0]); i++) {

            dist = sqrt(pow((x - landmarks[i][0]), 2) + pow((y - landmarks[i][1]), 2));

            prob \*= gaussian(dist, sense\_noise, measurement[i]);

        }

        return prob;

    }

    double x, y, orient; //robot poses

    double forward\_noise, turn\_noise, sense\_noise; //robot noises

private:

    double gen\_gauss\_random(double mean, double variance)

    {

        // Gaussian random

        normal\_distribution<double> gauss\_dist(mean, variance);

        return gauss\_dist(gen);

    }

    double gaussian(double mu, double sigma, double x)

    {

        // Probability of x for 1-dim Gaussian with mean mu and var. sigma

        return exp(-(pow((mu - x), 2)) / (pow(sigma, 2)) / 2.0) / sqrt(2.0 \* M\_PI \* (pow(sigma, 2)));

    }

};

// Functions

double gen\_real\_random()

{

    // Generate real random between 0 and 1

    uniform\_real\_distribution<double> real\_dist(0.0, 1.0); //Real

    return real\_dist(gen);

}

double mod(double first\_term, double second\_term)

{

    // Compute the modulus

    return first\_term - (second\_term)\*floor(first\_term / (second\_term));

}

double evaluation(Robot r, Robot p[], int n)

{

    //Calculate the mean error of the system

    double sum = 0.0;

    for (int i = 0; i < n; i++) {

        //the second part is because of world's cyclicity

        double dx = mod((p[i].x - r.x + (world\_size / 2.0)), world\_size) - (world\_size / 2.0);

        double dy = mod((p[i].y - r.y + (world\_size / 2.0)), world\_size) - (world\_size / 2.0);

        double err = sqrt(pow(dx, 2) + pow(dy, 2));

        sum += err;

    }

    return sum / n;

}

double max(double arr[], int n)

{

    // Identify the max element in an array

    double max = 0;

    for (int i = 0; i < n; i++) {

        if (arr[i] > max)

            max = arr[i];

    }

    return max;

}

/\*

void visualization(int n, Robot robot, int step, Robot p[], Robot pr[])

{

    //Draw the robot, landmarks, particles and resampled particles on a graph

    //Graph Format

    plt::title("MCL, step " + to\_string(step));

    plt::xlim(0, 100);

    plt::ylim(0, 100);

    //Draw particles in green

    for (int i = 0; i < n; i++) {

        plt::plot({ p[i].x }, { p[i].y }, "go");

    }

    //Draw resampled particles in yellow

    for (int i = 0; i < n; i++) {

        plt::plot({ pr[i].x }, { pr[i].y }, "yo");

    }

    //Draw landmarks in red

    for (int i = 0; i < sizeof(landmarks) / sizeof(landmarks[0]); i++) {

        plt::plot({ landmarks[i][0] }, { landmarks[i][1] }, "ro");

    }

    //Draw robot position in blue

    plt::plot({ robot.x }, { robot.y }, "bo");

    //Save the image and close the plot

    plt::save("./Images/Step" + to\_string(step) + ".png");

    plt::clf();

}

\*/

int main()

{

    //Practice Interfacing with Robot Class

    Robot myrobot;

    myrobot.set\_noise(5.0, 0.1, 5.0);

    myrobot.set(30.0, 50.0, M\_PI / 2.0);

    myrobot.move(-M\_PI / 2.0, 15.0);

    //cout << myrobot.read\_sensors() << endl;

    myrobot.move(-M\_PI / 2.0, 10.0);

    //cout << myrobot.read\_sensors() << endl;

    // Create a set of particles

    int n = 1000;

    Robot p[n];

    for (int i = 0; i < n; i++) {

        p[i].set\_noise(0.05, 0.05, 5.0);

        //cout << p[i].show\_pose() << endl;

    }

    //Re-initialize myrobot object and Initialize a measurment vector

    myrobot = Robot();

     vector<double> z;

    //Iterating 50 times over the set of particles

    int steps = 50;

    for (int t = 0; t < steps; t++) {

        //Move the robot and sense the environment afterwards

        myrobot = myrobot.move(0.1, 5.0);

        z = myrobot.sense();

        // Simulate a robot motion for each of these particles

        Robot p2[n];

        for (int i = 0; i < n; i++) {

            p2[i] = p[i].move(0.1, 5.0);

            p[i] = p2[i];

        }

        //Generate particle weights depending on robot's measurement

        double w[n];

        for (int i = 0; i < n; i++) {

            w[i] = p[i].measurement\_prob(z);

            //cout << w[i] << endl;

        }

        //Resample the particles with a sample probability proportional to the importance weight

        Robot p3[n];

        int index = gen\_real\_random() \* n;

        //cout << index << endl;

        double beta = 0.0;

        double mw = max(w, n);

        //cout << mw;

        for (int i = 0; i < n; i++) {

            beta += gen\_real\_random() \* 2.0 \* mw;

            while (beta > w[index]) {

                beta -= w[index];

                index = mod((index + 1), n);

            }

            p3[i] = p[index];

        }

        for (int k=0; k < n; k++) {

            p[k] = p3[k];

            //cout << p[k].show\_pose() << endl;

        }

        //####   DON'T MODIFY ANYTHING ABOVE HERE! ENTER CODE BELOW ####

        //Evaluate the error by priting it in this form:

        // cout << "Step = " << t << ", Evaluation = " << ErrorValue << endl;

        cout << "Step = " << t << ", Evaluation = " << evaluation(myrobot, p, n) << endl;

    } //End of Steps loop

    return 0;

}

### Graphing

Video 5.11

**Graphing**

So far, you’ve coded MCL and evaluated the overall quality of your solution. Now, you’ll be able to visualize what you’ve coded, or more precisely MCL in action.

**Udacity Workspace**

You will be using the Udacity Workspace/VM for this quiz. Thus, move to the next concept, enable the GPU, Go to Desktop and follow these instructions. Remeber to disable your GPU once you are done generating and visualizing the images.

A screenshot of a computer program

AI-generated content may be incorrect.

A diagram of a diagram

AI-generated content may be incorrect.

A diagram of a map

AI-generated content may be incorrect.

### Udacity Workspace

### Images

Video 5.13

### Outro

Video 5.14

## Where Am I?

### Overview

A blue and green circular object with question marks above it

AI-generated content may be incorrect.

**Project Overview**

Welcome to the Where Am I? localization project! In this project, you will learn to utilize ROS AMCL package to accurately localize a mobile robot inside a map in the Gazebo simulation environments.

Over the course of this lesson, you will learn several aspects of robotic software engineering with a focus on ROS:

* Create a ROS package that launches a custom robot model in a custom Gazebo world
* Utilize the ROS AMCL package and the Tele-Operation / Navigation Stack to localize the robot
* Explore, add, and tune specific parameters corresponding to each package to achieve the best possible localization results

A white background with black text

AI-generated content may be incorrect.

**Udacity Classroom Workspaces**

Udacity will provide you with a ROS Workspace to work on this project. It is in the [**'Project Workspace'**](https://classroom.udacity.com/nanodegrees/nd209/parts/a431d446-05df-4641-9e3d-79e1d55a7a2f/modules/b66739be-878e-4cea-8569-881b7eb2d34c/lessons/7dfe4265-e484-4efc-89e7-088540ff6720/concepts/a99f915f-2c34-4e0b-a96b-6fc381be08db) concept at the end of this lesson. By now you should be quite familiar with it. Here is the lesson for a recap: [**Udacity Workspace Instructions**](https://classroom.udacity.com/nanodegrees/nd209/parts/0778207d-f34a-4178-8ccf-9e06b5bd2203/modules/48156d08-abb1-4c03-a18d-9db738a0b92b/lessons/e0c61e8d-7eac-4807-8737-d2bd321ae7a2/concepts/47784838-aea6-4834-9ebb-79fbb3e135af)

Launch your Workspace's desktop GUI. Now, you are all set to start the Where Am I?Project!

**Native Installation & Virtual Machine**

If you are working with a native ROS installation or using a VM, some of the following package might need to be installed. You could install them as shown below:

A screenshot of a computer code

AI-generated content may be incorrect.

### Simulation Setup

In the previous projects, you have built your simulation environment and a robot. Let us set them up in the Project 3 Workspace for our next tasks.

Navigate to Workspace by clicking the Project Workspace on the side bar.

Setup your catkin\_ws folder as well as the src folder, then we need to grab the code from last project. Namely, the package you created to launch the simulation of your Gazebo world and your robot. You could do that in two ways.

**Method 1: git**

If you have pushed your submission for the ROS Introduction project to GitHub, go ahead and create a new repository then duplicate your code from last project to it. Substitute the url in the following commands with your project GitHub url.

A screen shot of a computer code

AI-generated content may be incorrect.

*Source:*[***https://help.github.com/articles/duplicating-a-repository/***](https://help.github.com/articles/duplicating-a-repository/)

**Method 2: Folder Upload**

If you want to upload the package folder to the Project 3 Workspace, go to the project Workspace and click the + button, then select Upload Folder to upload your package folder!

A screenshot of a computer

AI-generated content may be incorrect.

A screenshot of a computer program

AI-generated content may be incorrect.

### Project Workspace

* To follow along with the project instructions, use local VM image (Ubuntu 16.04 LTS) running on your VMWare/VirtualBox.
* Once you log into the VM image, open a Terminal window.
* You're now ready to follow along in your development environment with this project!

### Map Setup

Great! We have our simulation environment ready now. However, we cannot localize the robot just yet. The poor robot has zero information on its surroundings! Let us generate a map for it so that it knows what to expect in this environment.

Generally speaking, in the development of a robotic project, engineers utilize Mapping tools to measure and map the area robot will be operating in. Since we are developing in simulation environment the problem is simplified. We could generate the map from Gazebo world directly using a ROS package: **[pgm\_map\_creator](https://github.com/udacity/pgm_map_creator" \t "_blank)**.

**Note that currently, the map creator could not handle objects in the environment well. Please use the it with vertical surfaces only!**

A screenshot of a computer

AI-generated content may be incorrect.

**PGM Map File**

The map ROS AMCL Package uses is a pgm file. A pgm file is a grayscale image file. For more information about pgm file or more generally, pnm file, please refer to [**Netpbm format Wiki Page**](https://en.wikipedia.org/wiki/Netpbm_format).

By default, AMCL package will treat 'darker' pixels as obstacle in the pgm map file, and 'lighter' pixels as free space. The threshold could be set as a parameter which we will cover when we are building the launch file.

Navigate to your ROS package folder and create a maps folder. That's where your map file will reside.

A white rectangle with blue text

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.

Link: [**metadata about the map**](http://wiki.ros.org/map_server#YAML_format).

### AMCL Package

You learned about Monte Carlo Localization (MCL) in great detail in the previous lessons. Adaptive Monte Carlo Localization (AMCL) dynamically adjusts the number of particles over a period of time, as the robot navigates around in a map. This adaptive process offers a significant computational advantage over MCL.

The [**ROS AMCL package (http://wiki.ros.org/amcl)**](http://wiki.ros.org/amcl) implements this variant and you will integrate this package with your robot to localize it inside the provided map.

A screenshot of a computer

AI-generated content may be incorrect.

As you have learned in the previous projects, ROS utilizes launch files to run ROS nodes. In the following concepts, you will start to build the launch file for the AMCL package!

### AMCL Launch File

A screenshot of a computer program

AI-generated content may be incorrect.

### AMCL Launch File: Map Server Node

**Map Server Node**

The first node is the map\_server node ([**http://wiki.ros.org/map\_server**](http://wiki.ros.org/map_server)). The map\_server node provides map data as a ROS service to other nodes such as the amcl node. Here, map\_server node will locate the map you created in the [**Map Setup**](https://classroom.udacity.com/nanodegrees/nd209-beta/parts/930229bf-7a57-45c8-9111-83c1d2954661/modules/b66739be-878e-4cea-8569-881b7eb2d34c/lessons/7dfe4265-e484-4efc-89e7-088540ff6720/concepts/f97e67d6-2baa-4022-853d-b5b6d39be176) step and send it out as the map data.

First, add an argument of the path to your map file so that you could easily change the map loaded and avoid typing long paths again:

A screenshot of a computer

AI-generated content may be incorrect.

### AMCL Launch File: AMCL Node

A screenshot of a computer

AI-generated content may be incorrect.

For more information on remap, check out the ROS Wiki here: [**http://wiki.ros.org/roslaunch/XML/remap**](http://wiki.ros.org/roslaunch/XML/remap)

A screenshot of a computer program

AI-generated content may be incorrect.

From the ROS Wiki ([**http://wiki.ros.org/amcl**](http://wiki.ros.org/amcl)), we could find the purpose of the parameters added above:

* odom\_frame\_id (string, default: "odom"): Which frame to use for odometry
* odom\_model\_type (string, default: "diff"): Which model to use, either "diff", "omni", "diff-corrected" or "omni-corrected"
* base\_frame\_id (string, default: "base\_link"): Which frame to use for the robot base
* global\_frame\_id (string, default: "map"): The name of the coordinate frame published by the localization system

Remember, AMCL package 'links' the robot (odom frame) with the world (map frame). These parameters are required for amcl package to localize the robot in the world.

A screenshot of a computer

AI-generated content may be incorrect.

A screenshot of a computer program

AI-generated content may be incorrect.

### AMCL Launch File: Move Base Node

**Move Base Node**

Two nodes down, one node to go!

You will be working with the **[move\_base](http://wiki.ros.org/move_base" \t "_blank)** package using which you can define a navigation goal position for your robot in the map, and the robot will navigate to that goal position. Note that this step is optional if you choose to use teleop node to control and localize your robot.

The move\_base package is a very powerful tool. It utilizes a costmap - where each part of the map is divided into which area is occupied, like walls or obstacles, and which area is unoccupied. As the robot moves around, a local costmap, in relation to the global costmap, keeps getting updated allowing the package to define a continuous path for the robot to move along.

What makes this package more remarkable is that it has some built-in corrective behaviors or maneuvers. Based on specific conditions, like detecting a particular obstacle or if the robot is stuck, it will navigate the robot around the obstacle or rotate the robot till it finds a clear path ahead.

A screenshot of a computer program

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.

wget https://s3-us-west-1.amazonaws.com/udacity-robotics/Resource/where\_am\_i/config.zip

unzip config.zip

rm config.zip

  <rosparam file="$(find udacity\_bot)/config/costmap\_common\_params.yaml" command="load" ns="global\_costmap" />

  <rosparam file="$(find udacity\_bot)/config/costmap\_common\_params.yaml" command="load" ns="local\_costmap" />

  <rosparam file="$(find udacity\_bot)/config/local\_costmap\_params.yaml" command="load" />

  <rosparam file="$(find udacity\_bot)/config/global\_costmap\_params.yaml" command="load" />

  <rosparam file="$(find udacity\_bot)/config/base\_local\_planner\_params.yaml" command="load" />

### Optional: Teleop Package

A screenshot of a computer

AI-generated content may be incorrect.

### Localization: Launching

A screenshot of a computer

AI-generated content may be incorrect.

**Rviz Configuration**

As you did in a previous section, setup your RViz by adding the necessary displays and selecting the required topics to visualize the robot and also the map.

**Add by display type**

In Rviz,

* Select odom for fixed frame
* Click the “Add” button and
  + add RobotModel: this would add the robot itself to RViz
  + add Map and select first topic/map: the second and third topics in the list will show the global costmap, and the local costmap. Both can be helpful to tune your parameters
  + add PoseArray and select topic/particlecloud: this will display a set of arrows around the robot

Each arrow is essentially a particle defining the pose of the robot that your localization package created. Your goal is to add/tune the parameters that will help localize your robot better and thereby improve the pose array.

*Note:* You can save the above RViz setup in a configuration file and launch RViz with the same configuration every time. This will make the process more efficient for you! Click file -> save config to save the current configuration.

A screenshot of a computer

AI-generated content may be incorrect.

**Add by topic**

When adding elements to RViz, you could also select By topic tab. Here, all valid topics would be displayed and you could locate what you need faster!

A screenshot of a computer

AI-generated content may be incorrect.

***Transform Timeout* and *Map Update Loop***

If you received warning on *Transform Timeout* and *Map Update Loop*, you might want to configure the corresponding parameters. Namely larger *transform\_tolerance* value for the AMCL node and lower *update\_frequency* & *publish\_frequency* values in the configuration files.

### Localization: Parameters

For the next part of the project, you will identify and tune parameters for your amcl node in the amcl.launch file, to achieve better results.

* AMCL Parameters

The **[amcl package](http://wiki.ros.org/amcl" \l "Parameters" \t "_blank)** has a lot of parameters to select from. Different sets of parameters contribute to different aspects of the algorithm. Broadly speaking, they can be categorized into three categories - overall filter, laser, and odometry. Let’s cover some of the parameters that we recommend you start with or details to focus on.

**Overall Filter**

* min\_particles and max\_particles - As amcl dynamically adjusts its particles for every iteration, it expects a range of the number of particles as an input. Often, this range is tuned based on your system specifications. A larger range, with a high maximum might be too computationally extensive for a low-end system.
* initial\_pose - For the project, you should set the position to [0, 0]. Feel free to play around with the mean yaw value.
* update\_min\* - amcl relies on incoming laser scans. Upon receiving a scan, it checks the values for update\_min\_a and update\_min\_d and compares to how far the robot has moved. Based on this comparison it decides whether or not to perform a filter update or to discard the scan data. Discarding data could result in poorer localization results, and too many frequent filter updates for a fast moving robot could also cause computational problems.

**Laser**

There are two different types of models to consider under this - the likelihood\_field and the beam. Each of these models defines how the laser rangefinder sensor estimates the obstacles in relation to the robot.

The likelihood\_field model is usually more computationally efficient and reliable for an environment such as the one you are working with. So you can focus on parameters for that particular model such as the -

* laser\_\*\_range
* laser\_max\_beams
* laser\_z\_hit and laser\_z\_rand

Tuning of these parameters will have to be experimental. While tuning them, observe the laser scan information in RViz and try to make sure that the laser scan matches or is aligned with the actual map, and how it gets updated as the robot moves. The better the estimation of where the obstacles are, the better the localization results.

**Odometry**

odom\_model\_type - Since you are working with a differential drive mobile robot, it’s best to use the diff-corrected type. There are additional parameters that are specific to this type - the odom\_alphas (1 through 4). These parameters define how much noise is expected from the robot's movements/motions as it navigates inside the map.

**Note:** The odometry information for this project is received directly from Gazebo, and is equivalent to the ground truth value (no noise expected). So, you need not have to tune these parameters and can leave them at their default values. But feel free to experiment with some values and see if you notice any changes.

**Important:** The above set of parameters should help you get started, however they aren't the only ones that can improve your results. You are encouraged and required to go through the documentation, identify which parameters might help you improve your localization results, and experiment with them. All the remaining parameters and corresponding documentation can be found on the [**ROS wiki's amcl page**](http://wiki.ros.org/amcl#Parameters).

Identifying and tuning all these parameters can take time and effort. But don't worry. Based on the information and resources provided, you are well-equipped to tackle the problem head-on! Make sure to discuss your approaches with your fellow students in the ND Slack, and to reach out to your mentor for any further help.

### Localization: Testing

**Testing**

Now, let us test the performance of your AMCL package! You have two options to control your robot while it localize itself here:

* Send navigation goal via RViz
* Send move command via teleop package.

Navigate your robot, observe its performance and tune your parameters for AMCL! Capture screenshots and include them in your project submission.

**Option 1: Send 2D Navigation Goal**

Your first option would be sending a 2D Nav Goal from RViz. The move\_base will try to navigate your robot based on the localization. Based on the new observation and the odometry, the robot to further perform the localization.

Click the 2D Nav Goal button in the toolbar, then click and drag on the map to send the goal to the robot. It will start moving and localize itself in the process. If you would like to give amcl node a nudge, you could give the robot an initial position estimate on the map using 2D Pose Estimate.

A screenshot of a computer

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.

### Project Rubric

**Basic Requirements**

| **Criteria** | **Submission Requirements** |
| --- | --- |
| Did the student submit all required files? | Student submitted all required files:   * A fully functional ROS package containing:   + AMCL launch and configuration files.   + Teleoperation package setup files.   + Robot description (URDF) and associated files.   + World and map files for simulation. * Screenshots clearly showing:The localized robot in RViz with a visible particle cloud indicating pose certainty. |

**Simulation Setup**

| **Criteria** | **Submission Requirements** |
| --- | --- |
| Did the student set up the simulation environment properly? | The simulation environment must load without errors or crashes in Gazebo, and the robot should spawn in a stable state without tipping over or moving unexpectedly. |
| Is the student's simulation setup suitable for the localization task? | The environment should include enough distinct geometric features (e.g., walls, corners, furniture) for effective localization. |

**Localization Setup**

| **Criteria** | **Submission Requirements** |
| --- | --- |
| Did the student correctly build the launch files for localization? | * The launch files must include all required nodes and connections:   + A map\_server node to load the map.   + An amcl node with all necessary parameters set correctly.   + A move\_base node configured for navigation. * The launch files must execute and load all components without runtime errors. |
| Did the student properly set the parameters for localization? | Student filled required parameters for AMCL and move\_base in the launch file and the config file |

**Localization Performance**

| **Criteria** | **Submission Requirements** |
| --- | --- |
| Is the student's robot able to localize itself? | * Student's robot could quickly localize itself after being tele-operated in the student's world or given nav\_goal goal target. * Particle cloud visualization in RViz should reflect accurate pose estimation and adapt dynamically as the robot moves. |

**Suggestions to Make Your Project Stand Out**

Standing out submissions should have robots capable of continuously monitoring their surroundings and stoping whenever there is an obstacle blocking them. Also, start documenting your work on localization, which will contribute towards the final Home Service Robot project!

### Submit Project

Recap

You have successfully build and implemented the AMCL localization package on your robot! Let us review what are needed for the submission:

* Your Localization ROS Workspace folder
* Screenshot(s) of your robot localizing in the simulation

Submit Your Project

Once you have completed your project, use the [**Project Rubric**](https://review.udacity.com/#!/rubrics/2351/view) to review the project. If you have covered all of the points in the rubric, then you are ready to submit! If you see room for improvement in **any** category in which you do not meet specifications, keep working!