Udacity CS373: Programming a Robotic Car
Unit 6: Putting It All Together

Localization

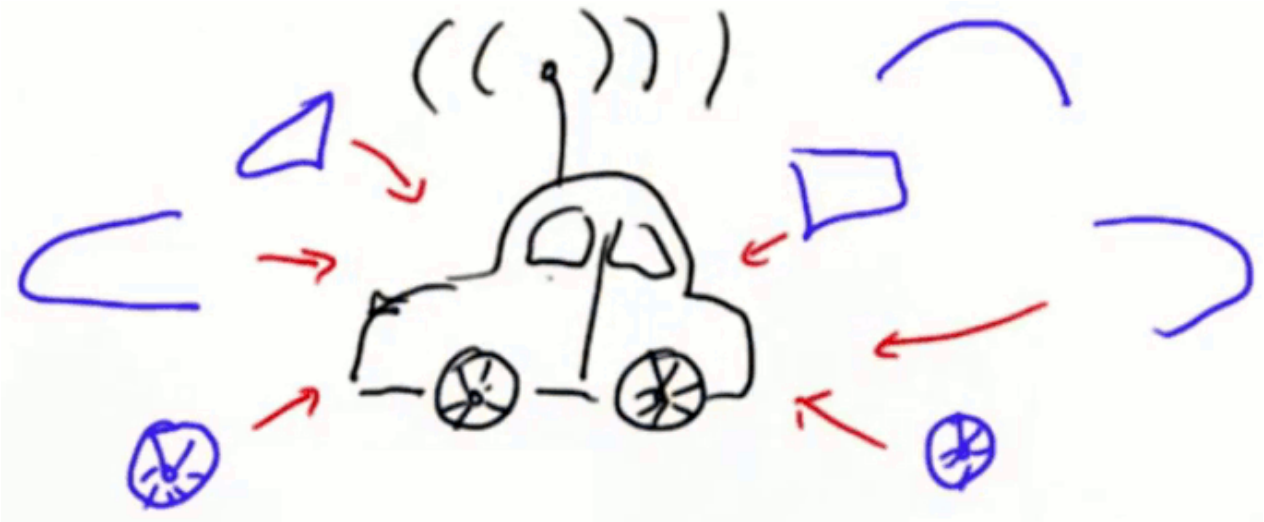
Welcome to Unit 6!

In this final unit, you will focus on putting everything you have learned up until now together. In this course you have learned:

Localization, Tracking, Planning, Control, Probabilities, Filters, A-star, DP, PID.

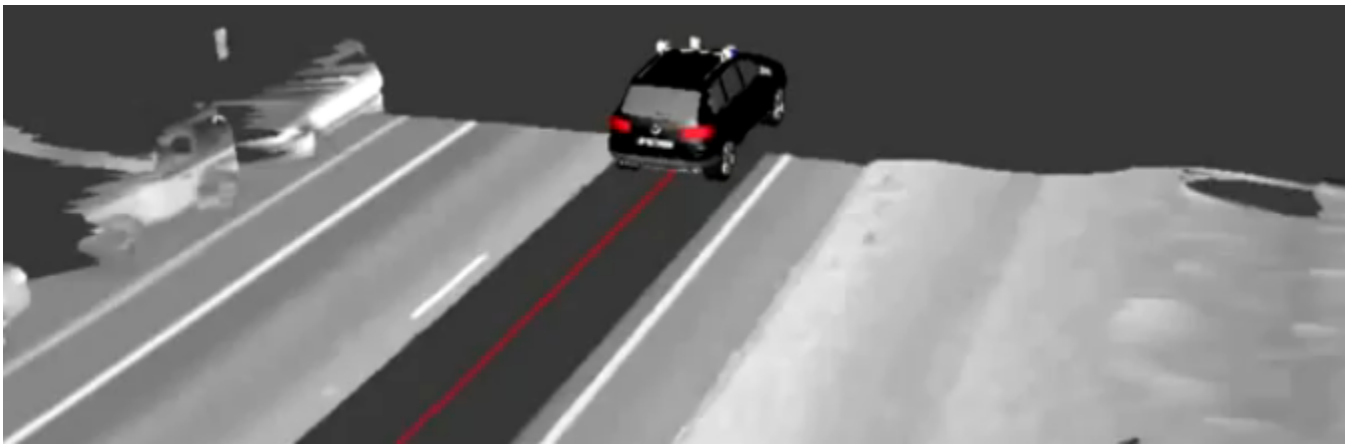
That is a lot of stuff! In addition to learning these concepts you also wrote programs to implement them. Now, you are ready to take all these pieces and put them together to build a true robotic self

driving car.

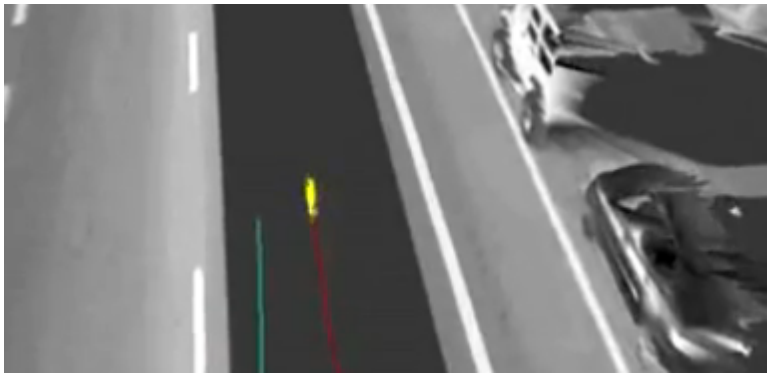


Q6-2: Localization

Previously, you learned how maps are used to localize a moving robot. In this image you can see a laser scan map being produced, in this case by a Stanford car.



Particles localize this robot by matching momentary measurements with the previously acquired map.



With this information Junior could be localized in the DARPA Urban Challenge with a centimeter precision. A similar method is used by the Google StreetView car – it is one of its secret sauces. The robotic car relies heavily on a previously built map and the type of localisation method that you implemented to localize the robot and make it follow known lanes.

Previously, you learned about Kalman filters, histogram filters (in the first unit) and particle filters. For this localization quiz, ask yourself the following questions about each of the filter types and then check the corresponding box if the attribute applies, there may be none or more than one.

1. Are any of these filters multimodal in their distribution?
2. Are any of these filters exponential in computational complexity relative to the number of dimensions?
3. Are any of these filters useful in the context of robotics?

QUIZ	Multi-modal?	Exponential?	Useful?
Kalman			
Histogram			
Particle			

Answer to Q6-2

Planning

Recall your planning concepts: breadth-first planning, A* planning and Dynamic Programming planning. Each planning concept addresses slightly different use cases, which are being used to plan the trajectory of the Google StreetView car and the DARPA Grand Challenge cars. In fact, you implemented good number of those, and that is quite amazing!

Q6-3: Planning

Here is a quiz about planning!

Check any or all of those boxes, if you think that the corresponding planner:

1. acts in a continuous space,
2. finds an optimal solution,
3. is a universal plan (that means the solution once found can be applied to arbitrary states)
4. has a solution that is local, which means given an approximate initial solution, it can not really do anything more than locally refine the solution.

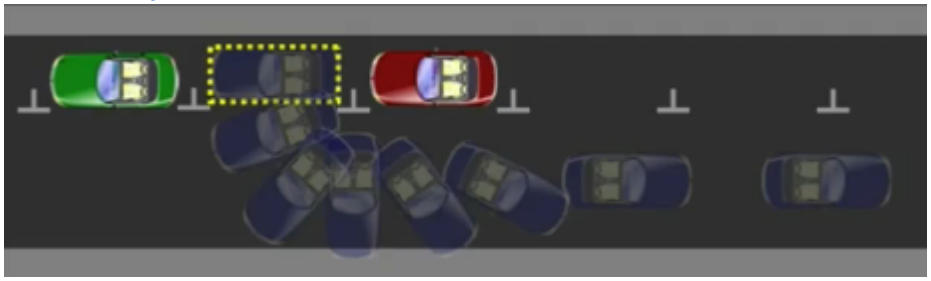
QUIZ	Continuous	Optimal	Universal	Local
Breadth First				
A*				
Dynamic Programming				
Smoothing				

Answer to Q6-3

PID

You also learned about control and implemented your first controller. Controllers are what make things like the autonomous sliding parking experiment possible.

Optional material: see Autonomous Sliding Parking on YouTube http://www.youtube.com/watch?v=gzI54rm9m1Q.



Q6-4: PID

Your knowledge of the PID control includes the P term, the I term, and the D term.

Please check which of the following terms are associated the most with:

1. avoiding overshoot,
2. minimizing error relative to the reference trajectory
3. compensating drift.

QUIZ	Avoiding overshoot	Minimizing error	Compensating drift
P			
I			
D			

Answer to Q6-4

Your Robot Car

Put it all together now in a single piece of software! It took Sebastian a whole day to do this, but you will not have to do it all yourself. Although you will have some help (because Sebastian knows you are a busy student) you should still be able to take all the lessons that you learned and put it in a single system. Here is a little bit of help with setting it all up.

The environment that is set up for you is very similar to what you have seen in the past lessons. There is a class **`robot`** that has certain **`noise`** characteristics associated with it. There is the familiar **`__init__`** function, the position setting function **`set`**, and the **`set_noise`** function.

There are two new checking functions: **`check_collision`** that checks if you have a collision with the world called **`grid`**, and a **`check_goal`** function that checks if you have reached a **`goal`** according to a certain distance, **`threshold`**.

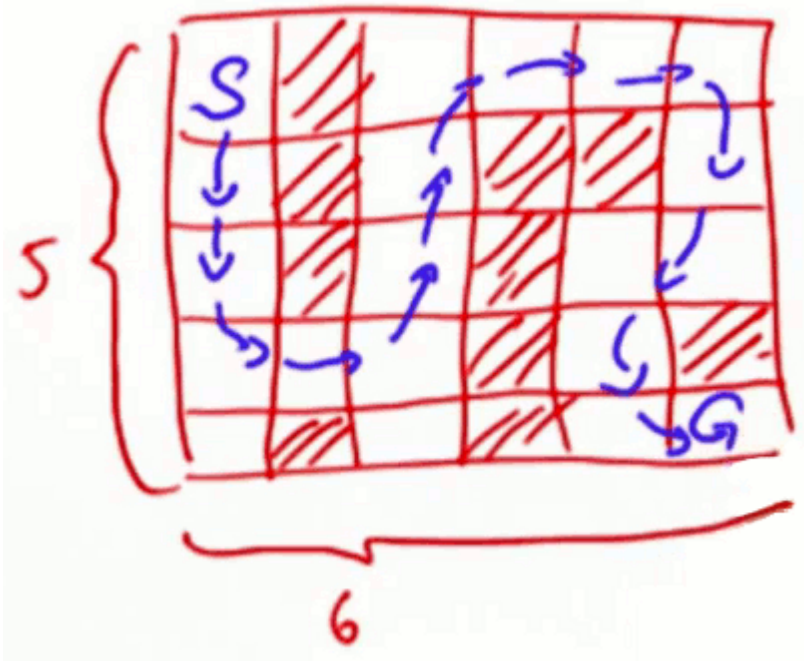
The **`move`** function should be very familiar to you at this point; it applies **`noise`** to the motion command. It is the same code that you originally wrote. The **`sense`** function measures the (x, y) location similar to a GPS on a car, but with substantial measurement noise. Corresponding to the **`sense`** function, is a **`measurement_probability`** function that you might want to use in your filter, which evaluates the probability of a measurement relative to the ground truth coordinates of the robot, using gaussians.

Armed with all this, here is the problem!

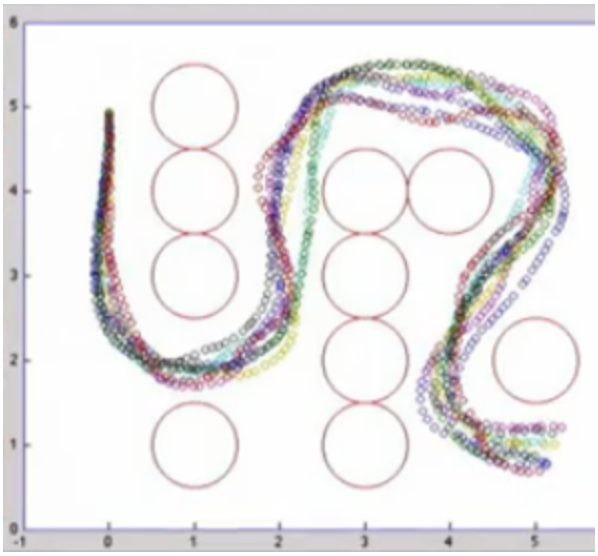
Given a grid:

```
# grid format:
#   0 = navigable space
#   1 = occupied space
grid = [[0, 1, 0, 0, 0, 0],
        [0, 1, 0, 1, 1, 0],
        [0, 1, 0, 1, 0, 0],
        [0, 0, 0, 1, 0, 1],
        [0, 1, 0, 1, 0, 0]]
```

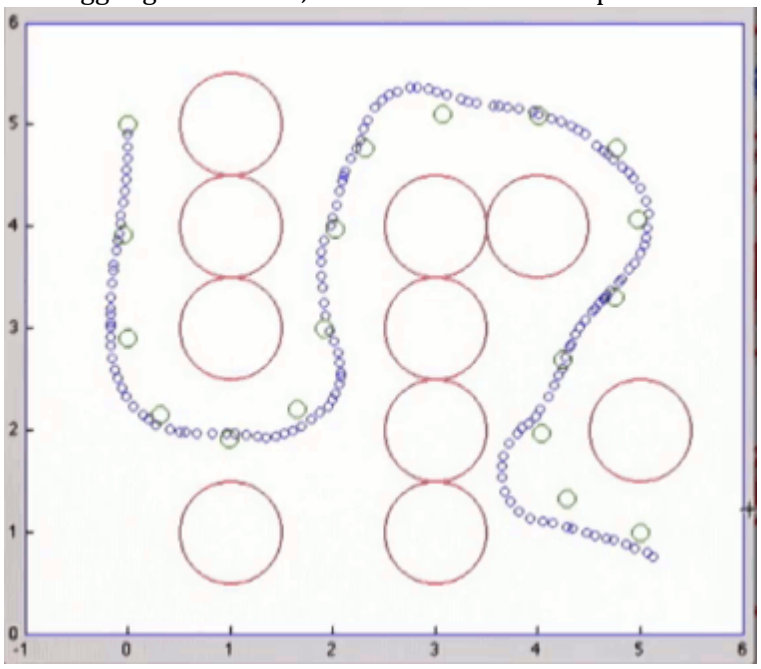
Put everything together and build a robotic car. Use the bicycle model to drive through the continuous free space on something close to the shortest path, all the way into the goal.



Below you can see a solution that Sebastian implemented. The obstacles are implemented as circles through the center of the grid cells, not exactly correct, but good enough for this implementation. You can see that the paths, displayed here from multiple runs, are far from optimal because there is a control noise and there is a measurement noise. In the end they all make it safely through the free space and into the corner where the goal objective is.



Looking at the obstacles in detail, like in the image below, you can see that the spacing of the circles is somewhat variable. Also, observe that there are little corners in some places, either as an effect of a control noise or a measurement noise, or of somewhat deficient implementation. You can also see the control setpoints as bigger green circles, that are the smoothed points of the A* planner.



In the version that Sebastian has implemented for you, the controller does something very, very different. It actually chooses as a control objective going straight to the goal, where using the **`atan2`** function, executes the action with a **`speed`** of **`0.1`** and then reports a collision every time the robot hits something. If you look at the output – the coordinates and the orientation – you can see that there are still frequent collisions the robot undergoes in its attempt to reach the goal. The robot does eventually does reach its goal, but you can see two big regions of collision before the goal is finally reached.

Segmented CTE

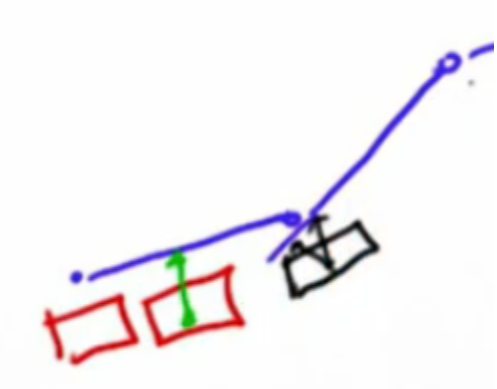
Sebastian has prepared code for you that is still a bit incomplete.

This code has a function called **`main`**, which runs a path planner **`plan`**, A*, a smoother, and then a controller, as in **`run`**. In the controller, **`run`**, even the particle filter is implemented, but there is nothing new; it is the same code that you have implemented. The code goes through a loop, where cross-track error, **`cte`** is computed, then applies the **`PD`** controller (no **`I`** term there), and finally runs the particle filter, as before, to estimate where the robot is.

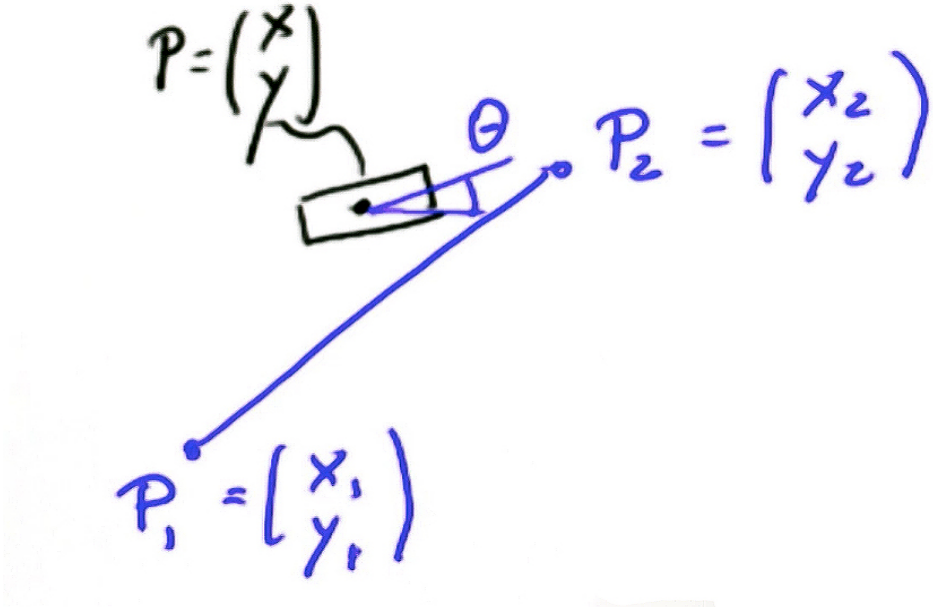
Q6-6: Segmented CTE

For this quiz, implement the cross-track error function, **`cte`**. Use as an input the **`estimate`** of the robot's position that you can get by running **`filter.get_position()`**.

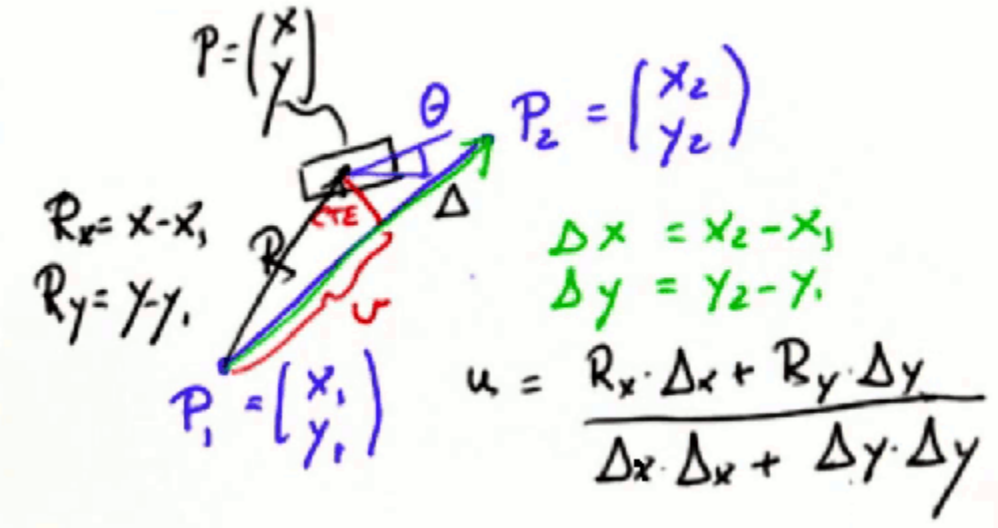
Here is the difficult part. Your path is now a sequence of linear pieces. When your robot drives along (red boxes), it has a certain crosstrack error. However, as the robot state projects beyond the end of a line segment, which is represented by the position of the black box, you have to change the corresponding line segment to be the next one. In addition to calculating the assigned error relative to an arbitrary line segment, not just the y-axis, you also have to detect when the robot steps beyond the end of a line segment and switch over to the next one.



Suppose this is your line segment, and the path is given by the coordinates of the start point P_1 and the end point P_2. Both points are (x,y) coordinates. Suppose your robot's position is something like the black box at P(x,y), where it has its own (x,y) estimate that comes from the particle filter in your case, and has its own orientation, theta θ.



At this point both the cross-track error and how far the robot has progressed along the line segment **u**, can be calculated using a dot product. Specifically, you can call the green vector Δx and Δy, as defined by $\Delta x = x_2 - x_1$ and $\Delta y = y_2 - y_1$. Refer to the black vector from P_1 to P(x,y) - R_1, $R_x = x - x_1$ and $R_y = y - y_1$. Then **u**, which measures the robot's progress along this segment, is given by the dot product:



The denominator normalizes the vector length to 1. This is the dot product of the black vector R and the green vector Δ.

$$u = \frac{R_x \Delta x + R_y \Delta y}{\Delta x^2 + \Delta y^2}$$

If **u** is larger than 1, it means the robot has left the segment and has to move to the next one. Finally, the cross-track error, the red line, is given by a similar (but not identical) dot product of:

$$CTE = \frac{R_y \Delta_x - R_x \Delta_y}{\Delta x^2 + \Delta y^2}$$

In this equation you have to multiply R_y with Δx and R_x with Δy. The normalizer is the same. Incorporate this equation into your code.

When you run your controller, you will see that the variable called **index** has been set up for you:

```
index = 0 # index into the path
```

When **u > 1**, increment **index** to make sure it never goes beyond what is legal in the path length. The cross-track error should be computed relative to the current **index**, and is, of course, a signed error, using the exact dot product described earlier.

The last thing that you will be given is the **path**. You should use the path called **spath**, which is given to the run function as one of the parameters. **spath[index]** is the path segment at segment number **index**, where **[0]** stands for **x** coordinate, and **[1]** for **y**:

```
spath[index][0] # x coordinate at index
spath[index][1] # y coordinate at index
```

Please fill in the missing code.

When you run your controller, with the missing code included, you will get a valid, nice path that mostly does not collide. Occasionally it does, because of randomness in the system, but it should mostly be collision free. For this example, it will require around 130 robot steps. A typical answer to this assignment will look like this random run:

[True, 0, 137]

True means that the robot actually found the goal, *0* means zero collisions, and *137* means that it took 137 steps. It is possible to sometimes get collisions because the obstacles are relatively large and the noise in the system makes it hard to navigate. Most of the time, though, the robot should be able to get to the goal without difficulties if you implement this piece of code correctly. A friendly warning from Sebastian: this takes a while.

Answer to Q6-6

Fun with Parameters

Take a moment to play with these model parameters:

```
weight_data    = 0.1
weight_smooth  = 0.2
p_gain         = 2.0
d_gain         = 6.0
```

Change the parameters as you wish and try to find values that will produce fewer collisions on average and that may allow the robot to reach the goal even faster. In the example the best values are obtained by an approximate investigation without applying *twiddle*. You may find applying twiddle particularly hard because it might never return and you will have to build in a **`timeout`** somehow. But it is fun to play with these parameters and try to find a better solution than the proposed one. You can use twiddle or whatever other method you want. But don't expect the correct answer for this question. This is a good opportunity to play with these parameters and see what the effects are on the solution.

To get this question right you have to change the parameters (or leave unchanged if you want) so that robot can reach the goal in less than 150 steps. Even if you got it, try to change them further to decrease the number of steps to 125. Be careful running twiddle: it might not converge in which case your program will be terminated by a timeout and your result considered wrong.

Wrap-Up

This concludes all of the new material for this course. Even though you haven't done the final exam yet, congratulations for getting this far! The fact that you got this far means that you are a very amazing student because you put in a lot of work for this class. You have learned a lot and you should feel empowered to program robots better than before.

In this class, you studied the very basic set of methods that every roboticist should know for programming any robot intelligently. All of those robots have tracking problems, state estimation

problems, planning problems and control problems – be it a surgical robot, be it a robot maid or intelligent future household robot or even a flying robot such as autonomous plane.

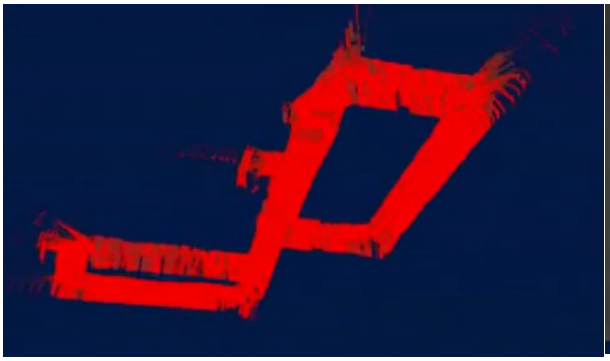
Thank you for being here!

SLAM

Learning more about SLAM was highly requested in emails and on the discussion forum, so here it is!

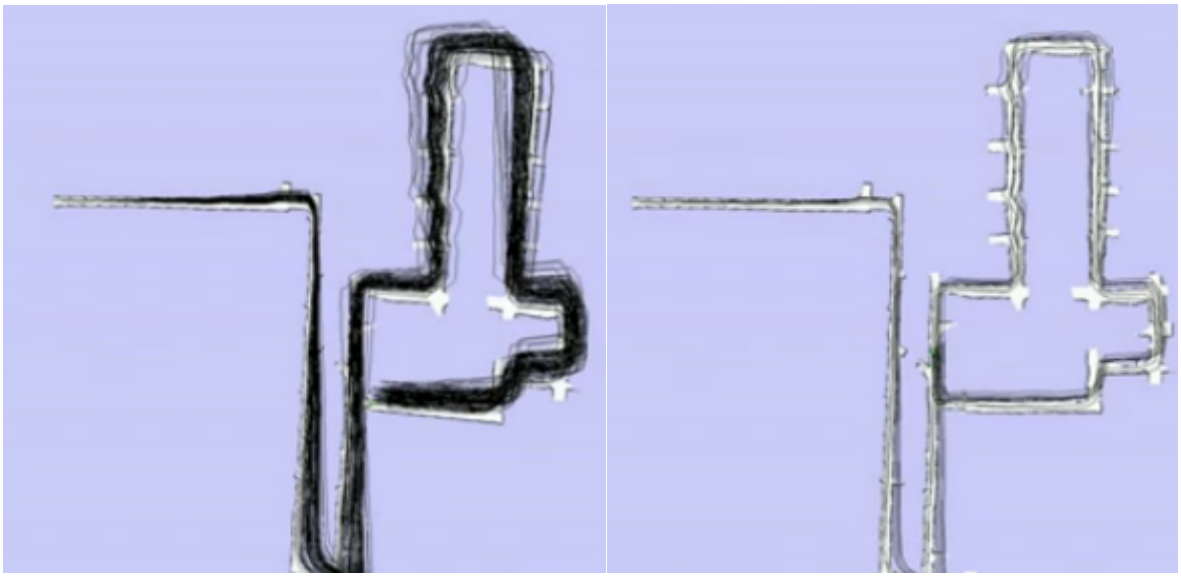
SLAM is a method for mapping. It stands for Simultaneous Localization And Mapping, which is all about building maps of the environment. Remember in the localization classes that the map was assumed to be given. That's not always the case and it's fascinating to understand how a robot can make these maps.

In the pictures below you can see a 3D map of an abandoned underground coal mine in Pittsburgh, Pennsylvania, near Carnegie Mellon university.



A number of different methods for building maps have been developed over the past ten years. What all of these methods have in common is that their process is to build a model of the environment while also addressing the fact that the robot itself introduces uncertainty when it moves.

In this example, when the loop is being closed, you can see how the mapping technology is able to accommodate this and find the consistent map despite the fact that the robot drifted along the way.



The key insight in building maps is that the robot itself might lose track of where it is by virtue of its motion uncertainty. In the localization you can address this issue by using an existing map, but now you don't have a map – you're building it!

This is when SLAM comes into play. SLAM doesn't stand for slamming a robot, it means *simultaneous localization and mapping.*

This is a vast research field. In this unit you will learn the method **Graph SLAM**, which is by far the easiest method to understand. With this method you can reduce the mapping problem to a couple of intuitive additions to a big matrix and a vector.

Q6-10: Is Localization Necessary

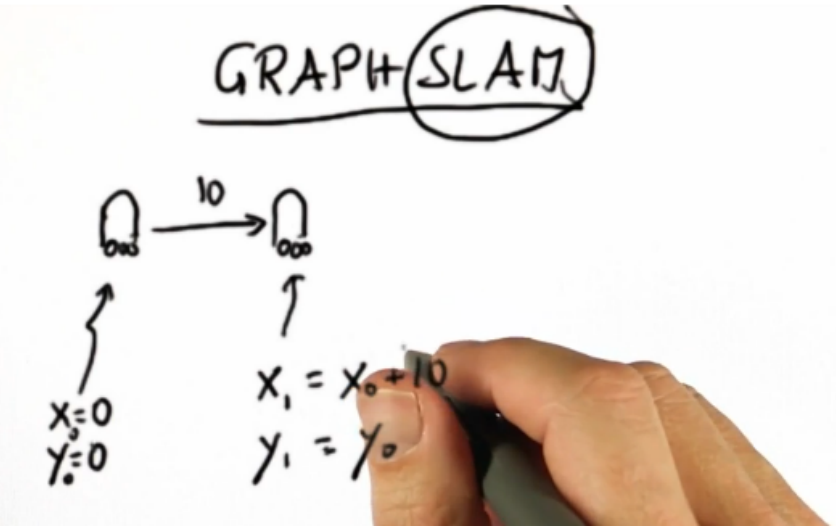
Here is a quick quiz.
When mapping the environment with a mobile robot, uncertainty in robot motion forces us to also perform localization?

a. yes
b. no

Answer to Q6-10

Q6-11: Graph SLAM

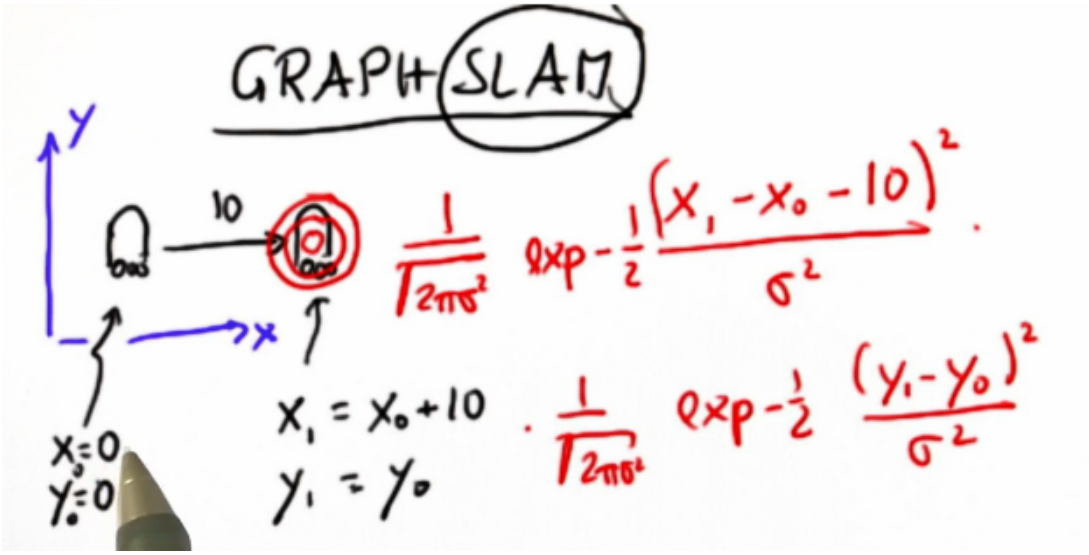
Graph SLAM is one of many methods for SLAM and it is by far the easiest to explain. Assume a robot with an initial location ($x_0 = 0$, $y_0 = 0$). Keep things simple by assuming the robot has a perfect compass, so you don't have to consider heading direction. Now, assume the robot moves to the right, namely in x direction, by 10. In a perfect world you would know that its new location is ($x_1 = x_0 + 10$, $y_0 = y_1$). But as you learned from various Kalman filter lessons and others, the location is actually uncertain. Rather than assuming the robot moved 10 units to the right exactly, you know that the actual location is a Gaussian centered around (10, 0). You also know it is possible for the robot to be somewhere else.



Recall the math for a Gaussian. Here is how it looks for the x variable: rather than writing $x_1 = x_0 + 10$, this equation expresses a Gaussian that peaks when x_1 and $x_0 + 10$ are the same:

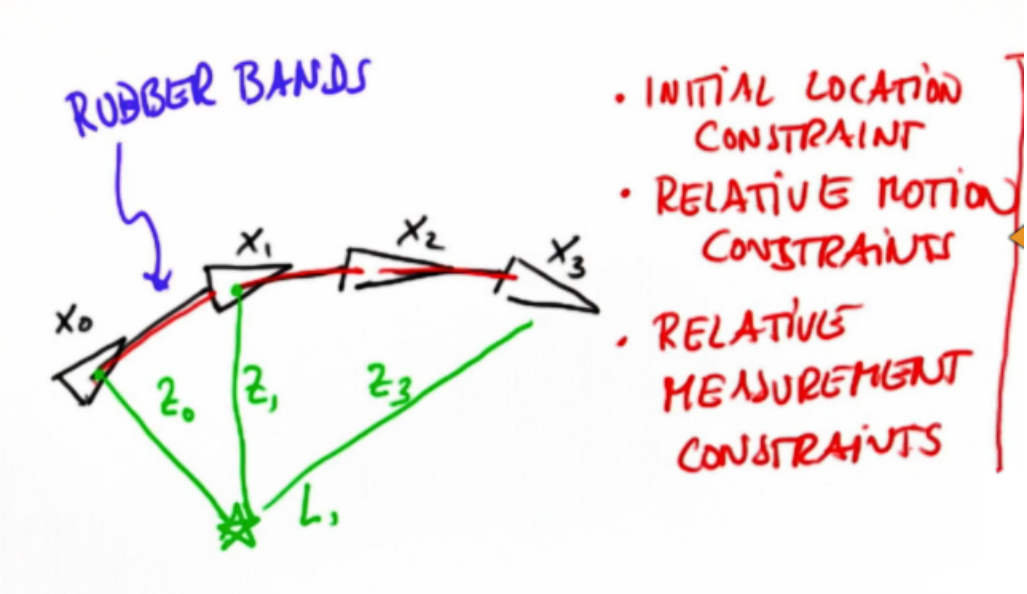
$$\frac{1}{\sqrt{2\pi\sigma^2}} exp - \frac{1}{2} \frac{(x_1 - x_0 - 10)^2}{\sigma^2}$$

Subtracting one from another and placing the result as a mean into a Gaussian, you get a probability distribution that relates x_1 and x_0. You can do the same for y. Since there is no motion, it is just making y_1 and y_0 as close as possible. The product of these two Gaussians is now a constraint. The likelihood of the position (x_1, y_1) has to be maximized given that the initial position is ($x_0 = 0$, $y_0 = 0$).



Graph SLAM defines probabilities by using a sequence of such constraints. Consider a robot that moves in some space where each location is characterized by a vector $x_0, x_1, x_2, \ldots$. Graph SLAM collects the initial location and lots of relative constraints that relate each robot pose to the previous robot pose – that are called relative robot motion constraints. These constraints are like rubber bands. In expectation the rubber band will be exactly the motion the robot sensed or that was commanded, but in reality it might have to bend a little bit to make the map more consistent.

On the map you can find landmarks. A landmark is seen from the robot pose with some relative measurement. All these measurements are also relative constraints, very much like the ones before. Again they are captured by Gaussians and impose relative measurement constraints. One such constraint is introduced every time the robot sees a landmark. So, Graph SLAM collects those constraints (that are insanely easy to collect) and relaxes the set of rubber bands to find the most likely configuration of the robot path along with the location of landmarks. That is the mapping process.



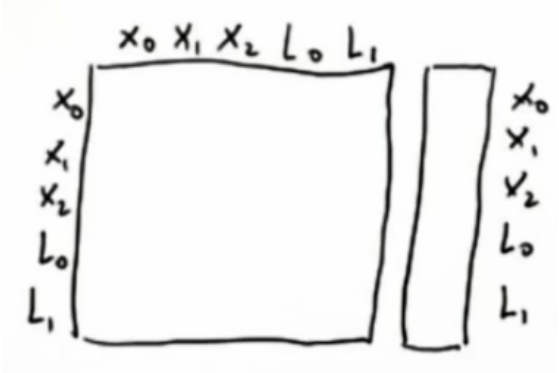
Suppose you have 6 robot poses (so you have 5 motions) and 8 landmark measurements. How many constraints do you have?

Answer to Q6-11

Optional material:
For deeper insight into Graph SLAM you can read this article -
The GraphSLAM Algorithm with Applications to Large-Scale Mapping of Urban Structure by Sebastian Thrun and Michael Montemerlo, published in THE INTERNATIONAL JOURNAL OF ROBOTICS RESEARCH / May–June 2006.

Q6-12: Implementing Constraints

To implement Graph SLAM, a matrix and a vector are introduced. The matrix is quadratic and labelled with all the poses and all the landmarks, assuming that the landmarks are distinguishable.

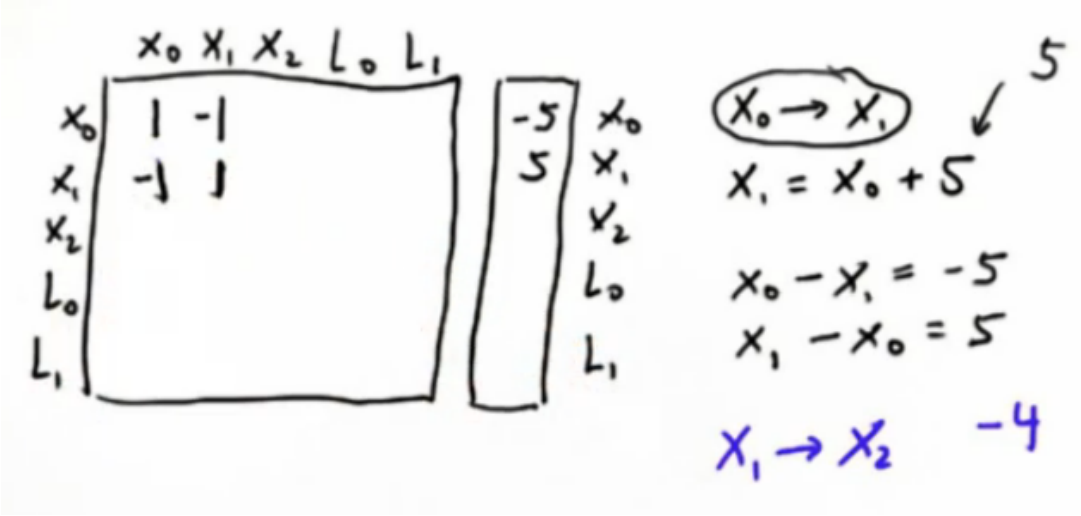


Every time you make an observation, for example between two poses, they become little additions locally in the four elements of the matrix defined over those poses. For example, the robots moves from x_0 to x_1 and it therefore believes that x_1 should be $x_1 = x_0 + 5$. This is entered into matrix in two ways:

1. The equation is changed so that both x terms are grouped together and the term, corresponding to the source pose, has positive sign: $x_0 - x_1 = -5$. Then, +1 and -1 coefficients for x_0 and x_1 respectively are added to the matrix which initially contains all zeros. The -5 constraint is added to the corresponding element in the vector.

2. You do the same for the equation with positive sign for the destination pose:
   $x_1 - x_0 = 5$.

Put differently, the motion constraint that relates x_0 and x_1 by the motion of 5 has modified the matrix incrementally, adding values for all elements that correspond to x_0 and x_1. The constraint is written twice, and in both cases you should verify that the diagonal elements are positive and the corresponding off-diagonal elements in the same row are negative.



Suppose the robot goes from x_1 to x_2 and the motion is -4; it moves in the opposite direction. What would be the new values in the matrix and vector?

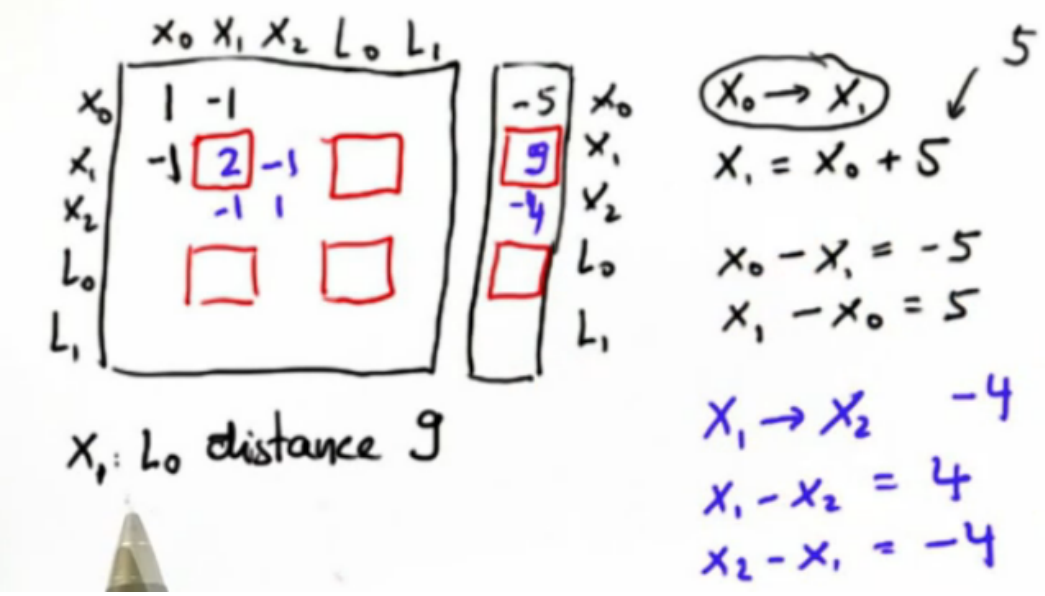
Hint: this motion only affects values that occur in the submatrix formed by x_1 and x_2 in Omega and the corresponding entries in Xi. Also, remember that you are to add up the values.

A6-12: Implementing Constraints

Q6-13: Adding Landmarks

Get ready for another quiz!

Suppose at x_1 the robot sees landmark L_0 at a distance of 9. This means there is a relative constraint between x_1 and L_0. The matrix elements for such constraints do not form a submatrix; they are spread apart. Modify these values in the matrix and vector to indicate that landmark L_0's position is 9 greater than x_1.



Answer to Q6-13

Q6-14: SLAM Quiz

This quiz is all about showing what you learned about Graph SLAM so far.

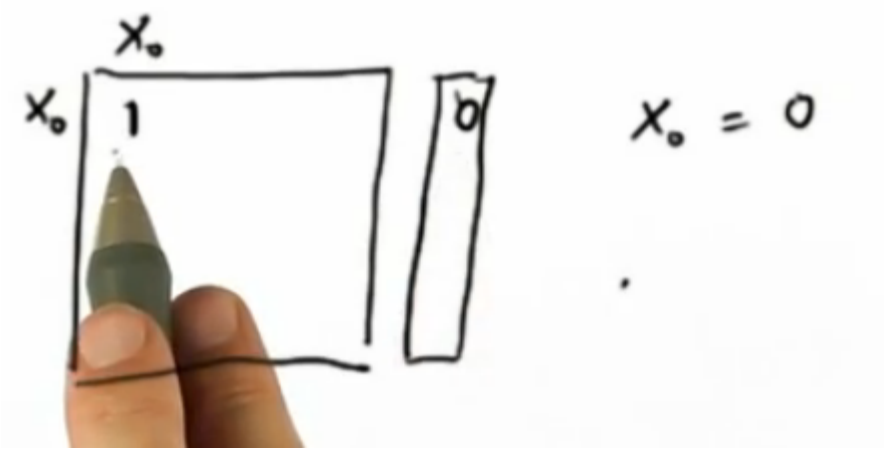
Please check all the answers that apply to the Graph SLAM:
a. All about local constraints
b. It requires multiplications
c. It requires additions
d. None of the above

Answer to Q6-14

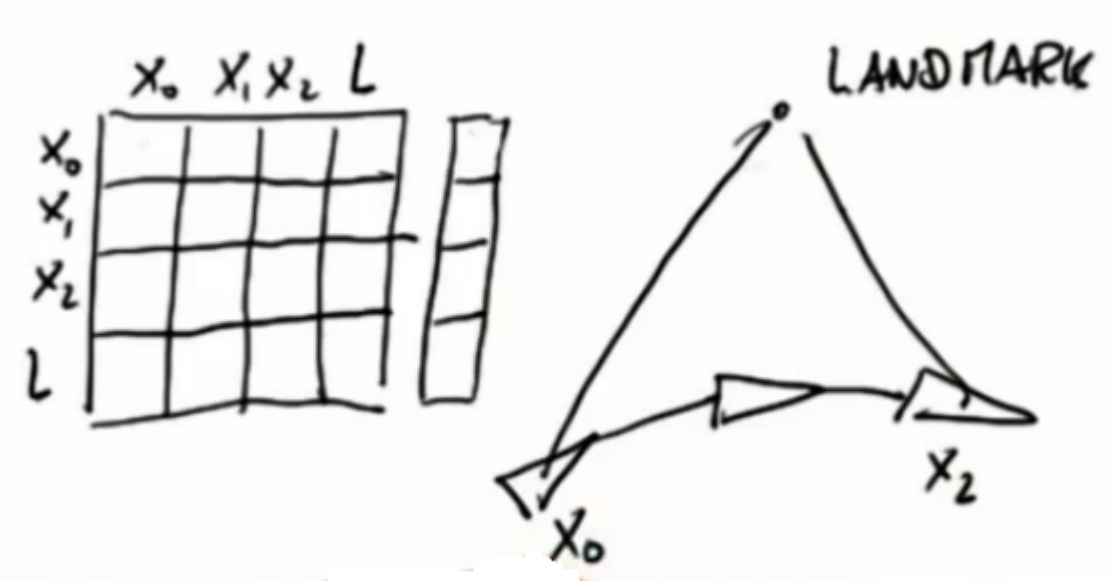
Q6-15: Matrix Modification

So far you have learned how to incorporate motion and measurement constraints to the Graph SLAM matrix and vector. Now there is one more thing to consider: the initial pose. How can you account for that constraint in the Graph SLAM?

Suppose your initial pose in a one-dimensional world is $x_0 = 0$. The figure below shows the procedure for including that in the Graph SLAM matrix and vector. You simply add 1 to the matrix element corresponding to x_0, and add the initial pose value to the corresponding vector element.



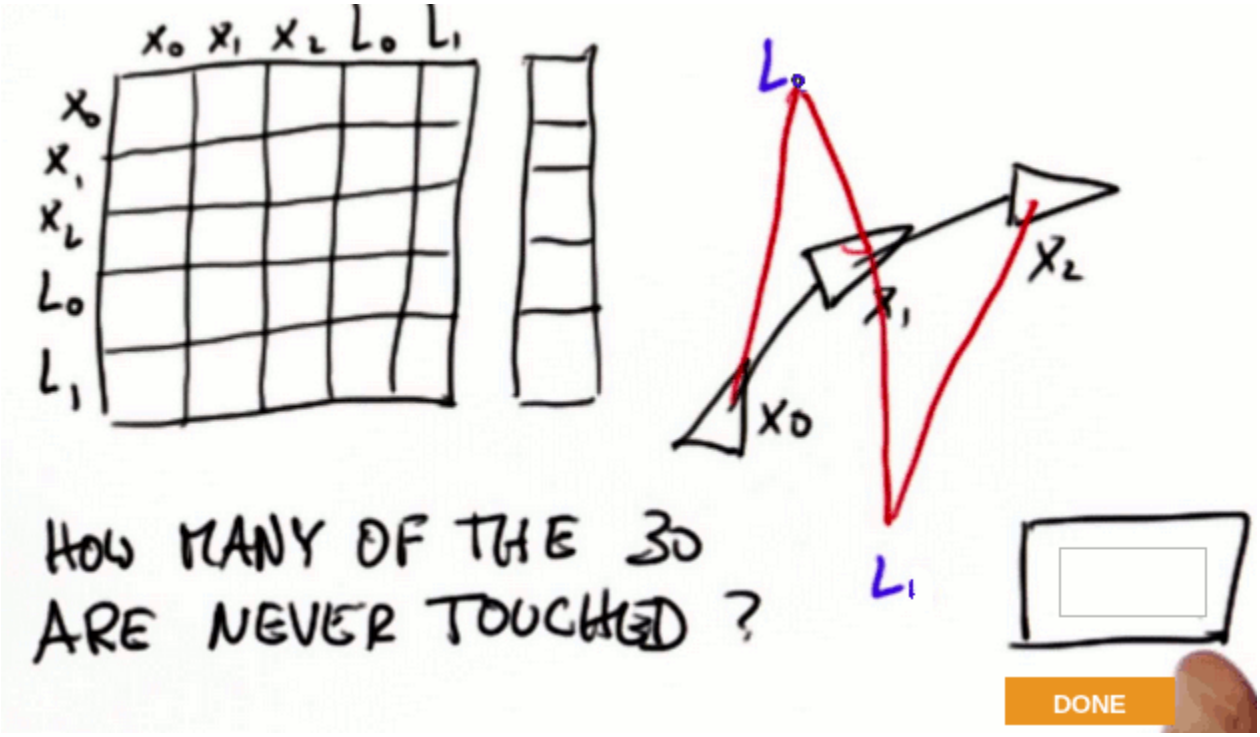
For this quiz, consider a robot which moves around through poses x_0 to x_2. The robot sees the same landmark *L* when he is at x_0 and x_2, but he is not able to see it from x_1. This is shown in the figure below. Go ahead and checkmark the fields that are modified in the matrix and vector *without putting actual numbers in*. Remember: the remaining fields will not be modified, so they will be zero.



Answer to Q6-15

Q6-16: Untouched Fields

For this quiz you are presented the same situation as before, except that this time the robot sees two landmarks. Landmark L_0 is seen from positions x_0 and x_1, while landmark L_1 is seen from positions x_1 and x_2. Can you answer how many of the 30 fields are not modified this time?



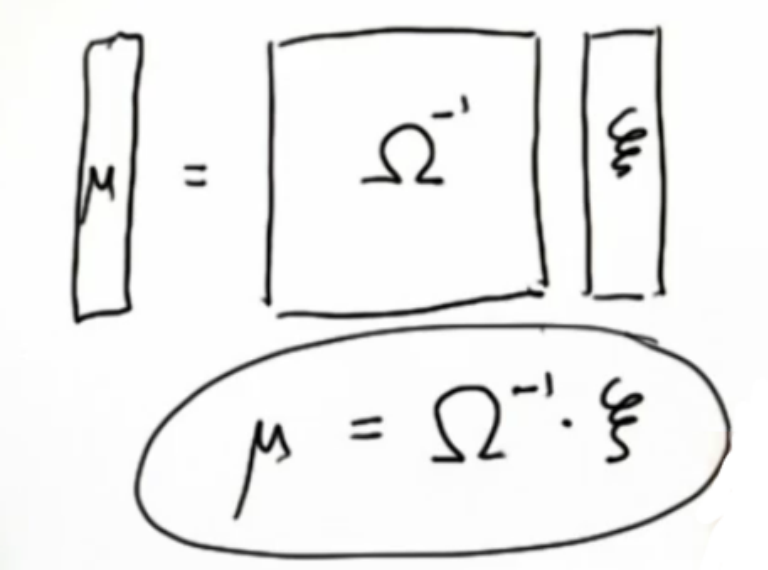
[Answer to Q6-16]

Omega and Xi

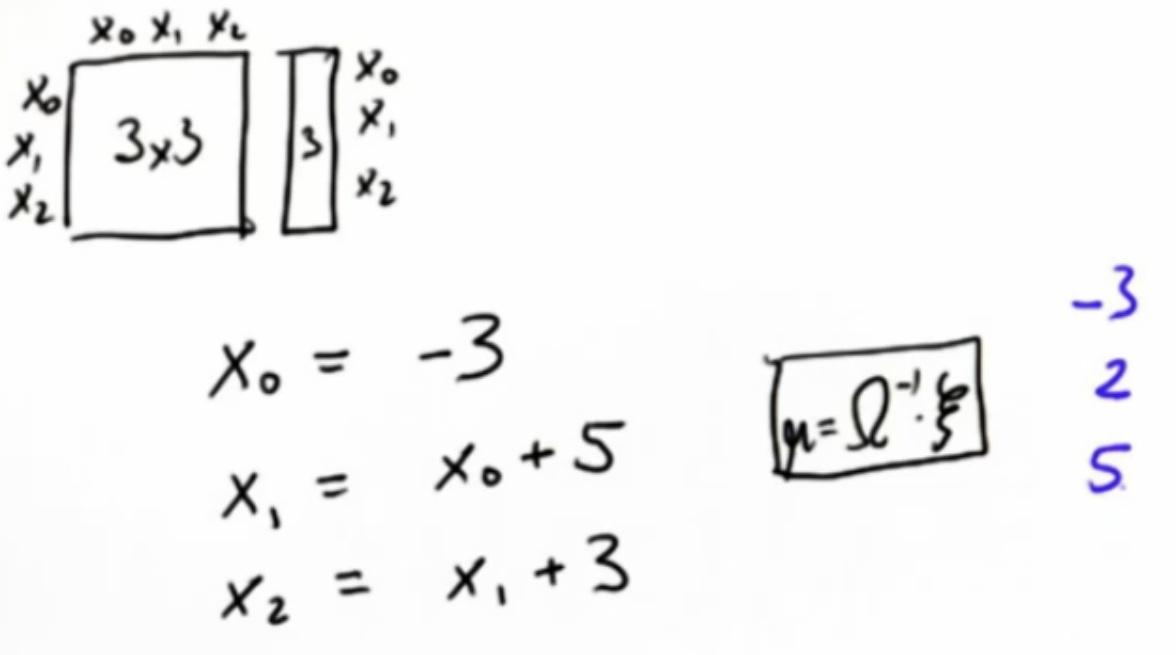
Q6-17: Omega and Xi

From now on, define Omega as the Graph SLAM matrix and Xi as the Graph SLAM vector. You can use the Omega and Xi matrices that you have been building to easily calculate the resulting mu vector, which will contain the estimated robot poses (this is the **localization** part of SLAM) and the estimated location of the landmarks (which is the **mapping**).

To do this calculation, you only need to invert the Omega matrix, which you can do in the matrix class with the inverse() function, and multiply this with the Xi vector. The output will be the mu vector! This calculation is shown graphically in the figure below.



For this assignment, you have to build the Omega matrix, the Xi vector using the **`initial_position`**, **`move1`** and **`move2`** values provided. You should then use the formula above to calculate and return mu. The figure below shows an example where **`initial_position=-3`**, **`move1=5`** and **`move2=3`**. The corresponding mu vector is shown in blue.

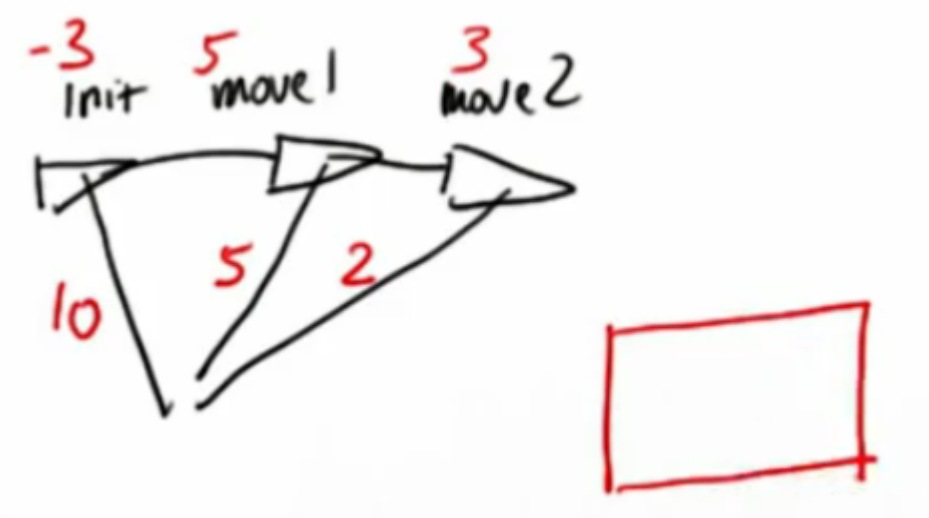


[Answer to Q6-17]

Landmark Position

Q6-18: Landmark Position

In this quiz, the robot starts at an initial position of -3, where it senses a landmark a distance 10 away. It then moves by 5 and measures the landmark to be 5 away. Finally, it moves by 3 and measures a landmark to be 2 away. What is the robot's best guess as to the position of the landmark? Assume a one-dimensional world in this example.



[Answer to Q6-18]

Expand

Q6-19: Expand

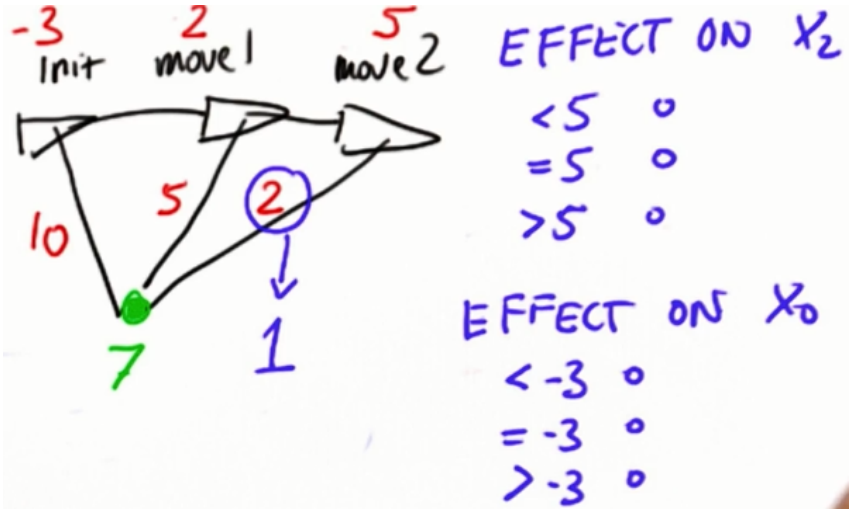
So now you will expand your routine to accommodate the landmark. Specifically, you will use a function that has been provided for you called **`expand.`** For example, you can call **`Omega.expand`** to convert a 3x3 matrix to a 4x4 matrix which includes the landmark itself.

[Answer to Q6-19]

Q6-20: Introducing Noise

Consider the robot motion again. Say you change the last measurement from 2 to 1.

What is the effect on x_2? What is the effect on x_0? Hint: Try it out in your code!



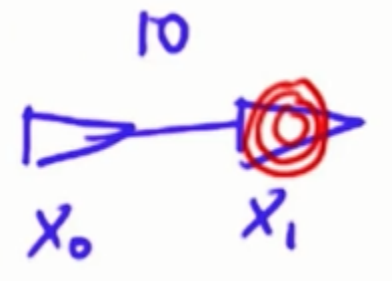
[Answer to Q6-20]

Confident Measurements

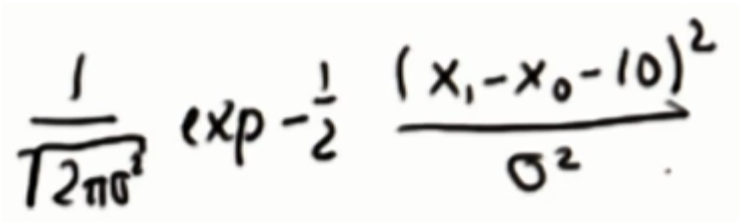
Q6-21: Confident Measurements

Here is a glimpse into why this works:

Suppose you have two robot positions $x_0$and x_1. The second position is 10 to the right, with some amount of Gaussian noise.

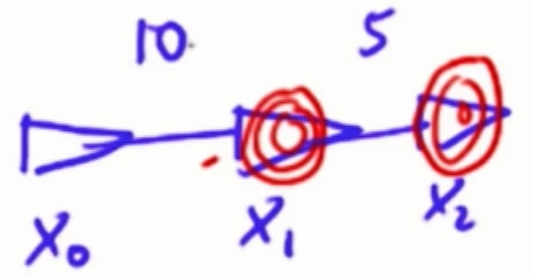


The uncertainty looks like:



There is a constant, an exponential, and an expression that will be minimized when $(x_1 - x_0)$ is 10, though it may deviate from exactly 10.. The Gaussian is maximum when this equation is fulfilled.

Now model a second motion, with an even bigger gaussian relative to the first one.



The constraint of course looks very similar

$$\frac{1}{\sqrt{2\pi\sigma^2}} \exp -\frac{1}{2} \frac{(x_2 - x_1 - 5)^2}{\sigma^2}$$

The total probability is the product of the two. If you want to maximize the product, here are a few tricks. The constant has no bearing on the maximum, so you can drop it. You can also drop the exponential if you turn the product into an addition. And now you can also drop the -1/2 inside what was the exponential.

You end up with an equation like $x_1/\sigma - x_0/\sigma = 10/\sigma$. The $1/\sigma$ term represents your confidence. For a small σ the $1/\sigma$ term becomes large, which means you are more confident.

For this quiz, modify the code so the last measurement has a really high confidence with a factor of 5. You should get **`[-3, 2.179, 5.714, 6.821]`** as the answer. You see in this result, the difference between the last point and the landmark are very close to the measurement difference of 1.0 because you have a relatively high confidence compared to the other measurements and motions.

Answer to Q6-21

Implementing SLAM

Q6-22: Implementing SLAM

Every time there's a constraint (initial position, motion or measurement) take it and add it to **`Omega`** and **`Xi`**. Also, multiply by a strength factor $1/\sigma$ representing your confidence.

Simply calculate $\mu = \Omega^{-1}\xi$ and the result is the path and the map. Amazing!

In this quiz, you have a generalized environment with the following parameters:

```
num_landmarks      = 5        # number of landmarks
N                  = 20       # time steps
world_size         = 100.0    # size of world
measurement_range  = 50.0     # range at which you can sense landmarks
motion_noise       = 2.0      # noise in robot motion
measurement_noise  = 2.0      # noise in the measurements
distance             = 20.0      # distance by which robot (intends to) move
each iteratation
```

make_data is a function which, given your environmental parameters, returns a sequence of motions and a sequence of measurements.

You will write the function **slam** which takes in this data and some environment parameters and returns a sequence of the robot path and estimated landmark positions.
Note that there is an initial constraint which places the robot at **[50., 50.]** the center of the robot world.

Answer to Q6-22

Congratulations

Congratulations you've made it very far! You've learned a lot about SLAM. Put the positions and the map into a matrix Ω and vector ξ. Between poses and measurements, make some simple additions to your matrix and vector and then you can solve for the path and the map at the same time using $\mu = \Omega^{-1}\xi$.

Answer Key

A6-2: Localization

You learned that the Kalman filter is unimodal, whereas histogram and particle filters are multimodal.
The Kalman filter was more efficient than the histogram and the particle filter, that both require exponentially more units in the number of the dimensions in the worst case.
And of course, they all are very, very useful, in case you did not get this from the class!

QUIZ	Multi-modal?	Exponential?	Useful?
Kalman			**YES**
Histogram	YES	YES	**YES**
Particle	YES	YES	**YES**

A6-3: Planning

Smoothing was the only one that really worked in a continuous domain, everything else was very discrete.
Breadth-first, A-star and DP all find the optimal solution.
DP is the only one that is universal.
And smoothing was the only one that was local

QUIZ	Continuous	Optimal	Universal	Local
Breadth First		YES		
A-star		YES		
Dynamic Programming		YES	YES	
Smoothing	YES			YES

A6-4: PID

The primary function of P is to minimize error, the D term avoids the overshoot, and systematic drift and biases are best addressed by the I term.

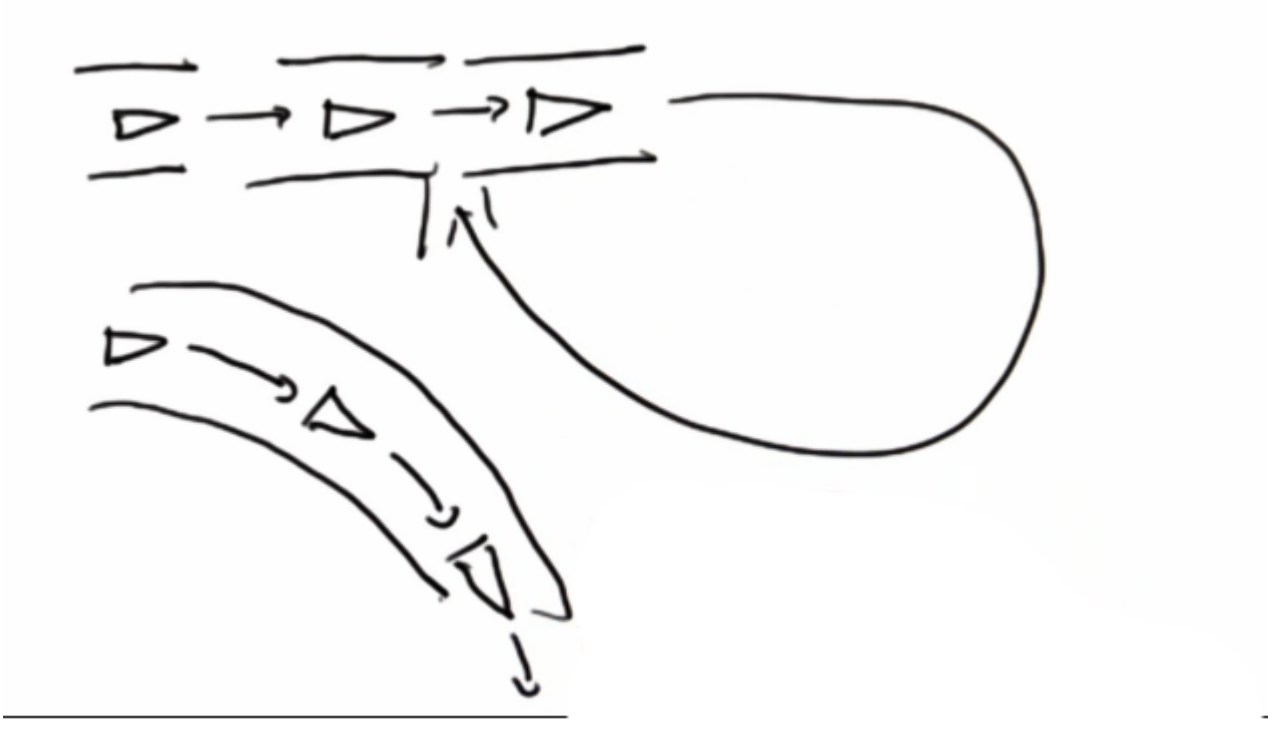
QUIZ	Avoiding overshoot	Minimizing error	Compensating drift
P		YES	
I			YES
D	YES		

A6-6: Segmented CTE
And here is Sebastians code for this exercise:

```
# some basic vector calculations
dx = spath[index+1][0] - spath[index][0]
dy = spath[index+1][1] - spath[index][1]
drx = estimate[0] -  spath[index][0]
dry = estimate[1] -  spath[index][1]
# u is the robot estimate projected into the path segment
u = (drx * dx + dry * dy)/(dx * dx + dy * dy)
#the cte is the estimate projected onto the normal of the path segment
cte = (dry * dx - drx * dy)/(dx * dx + dy * dy)
if u > 1:
    index += 1
```

A6-10: Is Localization Necessary
And the answer is **yes**. In nearly all cases of mapping the robot uncertainty is in motion, and that uncertainty might grow over time. It's necessary to address that, otherwise the map looks very bad.

Here is an example. Suppose a robot drives in the corridor and in senses surrounding walls. If this robot has a drift problem and because of uncertainty in its motion it believes the environment to be different from what it actually is. On the image below the top trajectory is the real one and the bottom is belief trajectory of a drifty robot.



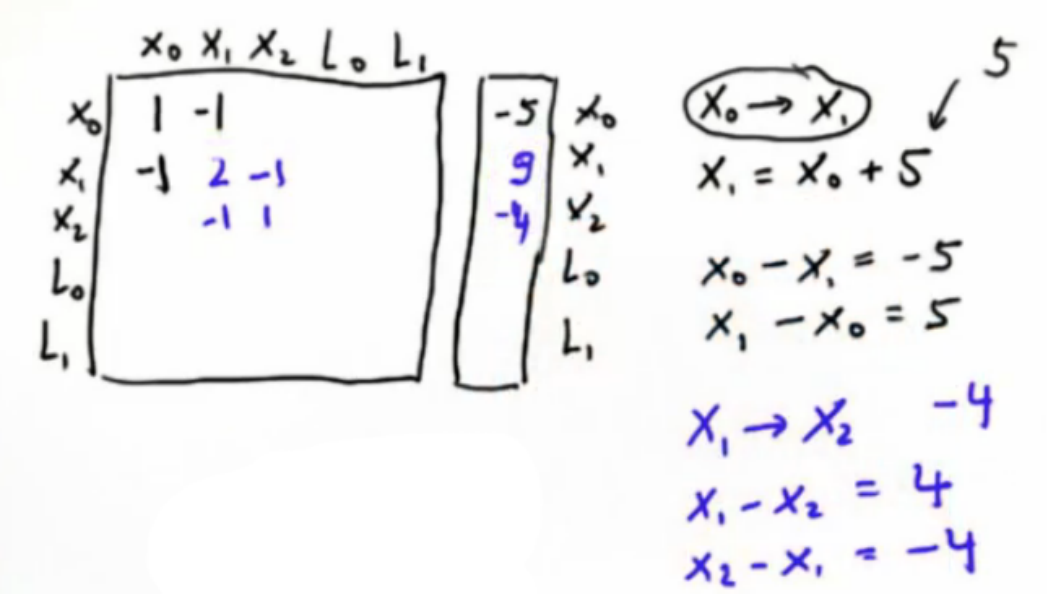
This might be indistinguishable at a first glance, but if the robot ever comes back to the same place, there is no opportunity to correct the map. So a good SLAM technique is able to understand not just the fact that environment is uncertain but also the robot itself runs on uncertain trajectory. That makes SLAM hard.

A6-11: Graph SLAM

And the answer is **14**. There is one initial location constraint, 5 motions (between 6 points) and 8 landmark constraints.

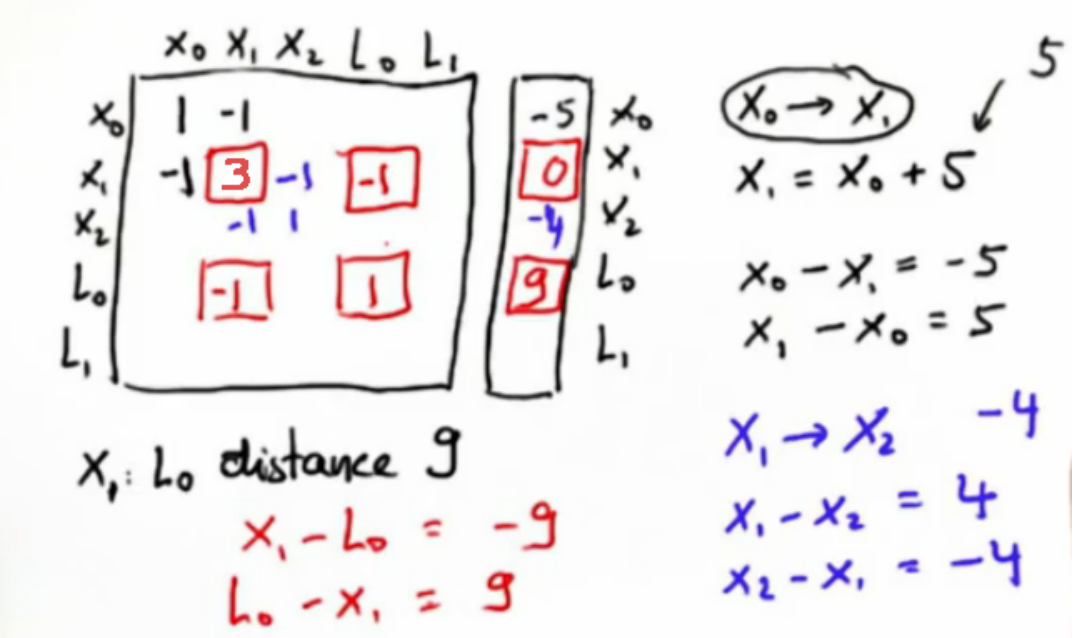
A6-12: Implementing Constraints

First, retransform the equations exactly the way it was done in the example: $x_1[?][?][?]\,x_2 = 4$ and $x_2[?][?][?]\,x_1 = [?][?][?]4$. Add 1 to the (^{x_1}, ^{x_1}) matrix element and -1 to the (^{x_1}, ^{x_2}), -1 to the (^{x_2}, ^{x_1}) and +1 again to the (^{x_2}, ^{x_2}).



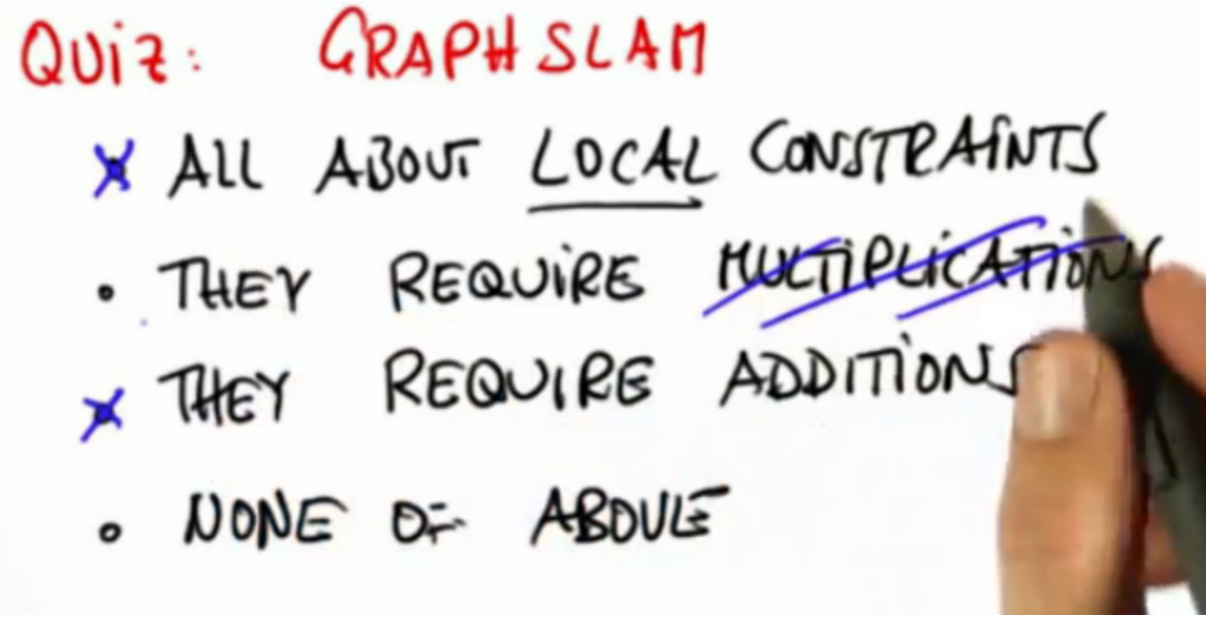
A6-13: Adding Landmarks

Transforming the equations exactly as before, you can obtain the following equations: $x_1[?][?][?]\,L_0 = [?][?][?]9$ and $L_0[?][?][?]\,x_1 = 9$. Adding +1 and -1 to the diagonal and off-diagonal matrix elements respectively and adding -9 and +9 to ^{x_1} and ^{L_0}vector elements gives you the correct result.



A6-14: SLAM Quiz

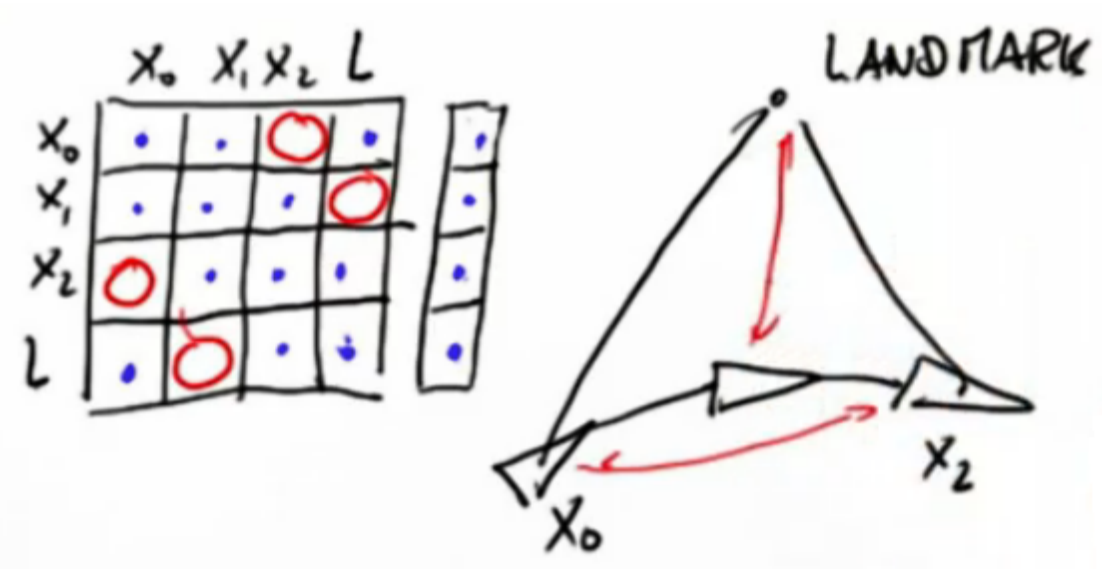
The right answers are the *first* and *third* boxes. Graph SLAM is indeed all about local constraints: you can see that because every motion ties together two locations, and every measurement ties together a location with a landmark. These constraints are represented as additions to the appropriate elements of the matrix and vector that are part of the algorithm. Multiplication is the wrong option here, the algorithm is all about addition! It's that simple!



A6-15: Matrix Modification

And the answer is given by the blue dots in the figure below! Note that almost every element of the matrix and vector are modified for this run. The only fields which are zero (marked in red)

correspond to the movement constraint between x_0 and x_2 and the measurement constraint between x_1 and *L*, as indicated in the red arrows below.



Here is a list of the fields each constraint modifies:

Initial pose: **matrix[0][0], vector[0]**

Movement $x_0 \rightarrow x_1$*:* **matrix[0][0], matrix[0][1], matrix[1][0], matrix[1][1], vector[0], vector[1]**

Movement $x_1 \rightarrow x_2$*:* **matrix[1][1], matrix[1][2], matrix[2][1], matrix[2][2], vector[1], vector[2]**

Measurement $x_0 \rightarrow L$*:* **matrix[0][0], matrix[0][3], matrix[3][0], matrix[3][3], vector[0], vector[3]**

Measurement $x_2 \rightarrow L$*:* **matrix[2][2], matrix[2][3], matrix[3][2], matrix[3][3], vector[2], vector[3]**

Note that **matrix[0][2]**, **matrix[2][0]**, **matrix[1][3]** and **matrix[3][1]** are *not* in this list. What this means is that there is no direct constraint between x_0 and x_2, since there is no direct motion information between them, as well as no direct constraint between x_1 and L, since there is no measurement information between x_1 and *L*.

A6-16: Untouched Fields

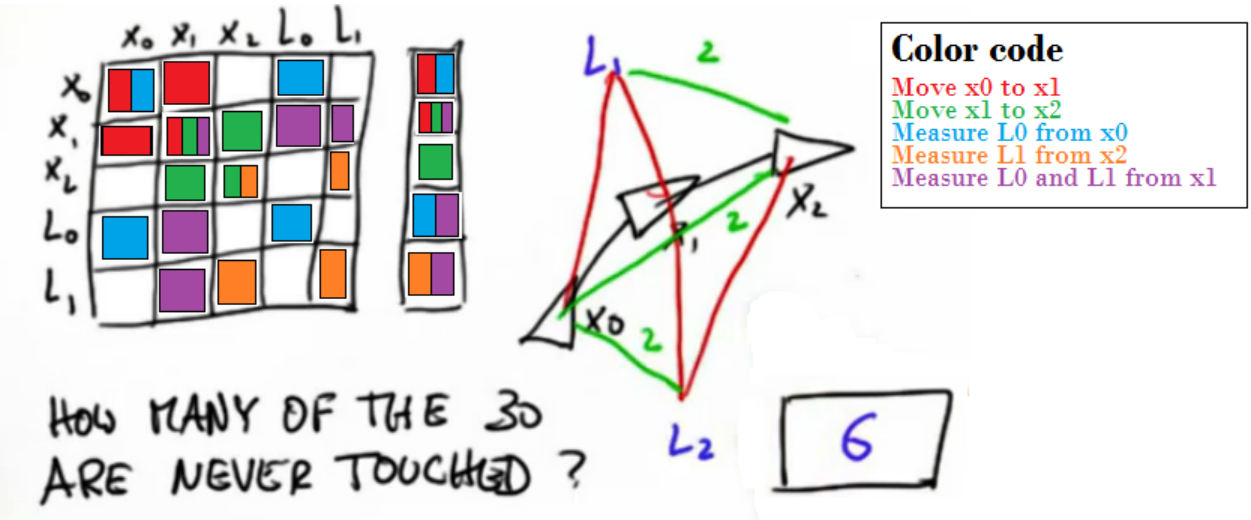
And the answer is... 6? No, the answer is 8, but Sebastian initially made a mistake. So if you got this answer right, congratulations! It can be somewhat confusing, as you will see below.

To answer this question, you need to take note of the constraints that are not present in the sequence. In his initial answer, Sebastian accounted for the direct motion constraint between x_0 and x_2, which is not observed. He also considered the measurement constraints corresponding to

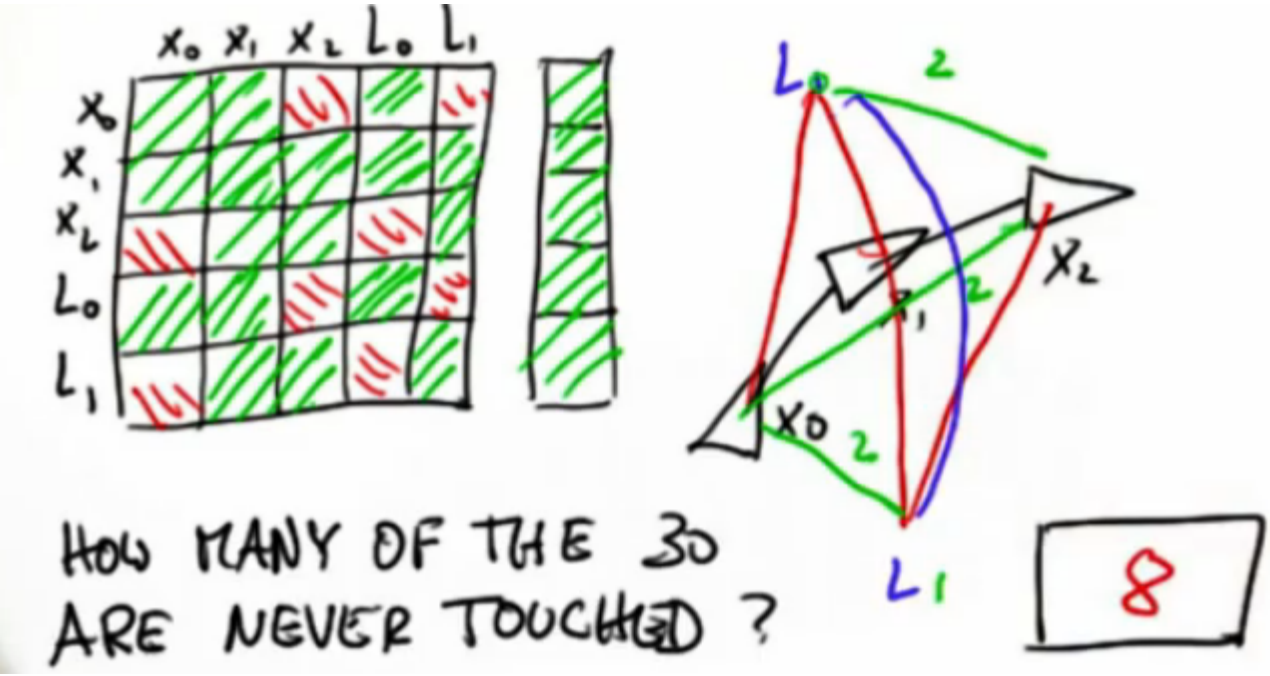
the landmark L_1, which is not seen at x_0, and the landmark L_0, which is not seen at x_2.

These three constraints are marked in green in the figure above. Each constraint is counted twice: the matrix element for (x_0, L_1) is always modified at the same time as the matrix element for (L_1, x_0). Due to that, the Graph SLAM matrix is always symmetric.

Sebastian then moves on to check which fields are modified by each constraint presented, to verify his answer. The figure below summarizes this verification, marking with different colors the elements which are modified by each constraint. In proceeding this way, he noticed his mistake: there are in fact 8 fields which are not modified in this sequence!



What happened is that he disregarded the fact that there is no relative measurement between the two landmarks, as marked in blue in the figure below. You could say that, since landmarks can't see, there is no direct link between them. Put differently, the small submatrix formed by the matrix elements corresponding to L_0 and L_1 will always be diagonal using this approach. The answer is, therefore, that there are actually 8 fields that are never touched.



A6-17: Omega and Xi

The matrices you have to build are the following:

```
Omega = [[2,-1,0],[-1,2,-1],[0,-1,1]]
Xi = [[initial_pos-move1],[move1-move2],[move2]]
```

You should have considered these constraints in order to get the right values: **initial_pos** at x_0, **move1** from x_0 to x_1, and **move2** from x_1 to x_2. You can also do it incrementally:

```
Omega = [[1,0,0],[0,0,0],[0,0,0]]
Xi = [[initial_pos],[0],[0]]

Omega += [[1,-1,0],[-1,1,0],[0,0,0]]
Xi += [[-move1],[move1],[0]]

Omega += [[0,0,0],[0,1,-1],[0,-1,1]]
Xi += [[0],[-move2],[move2]]
```

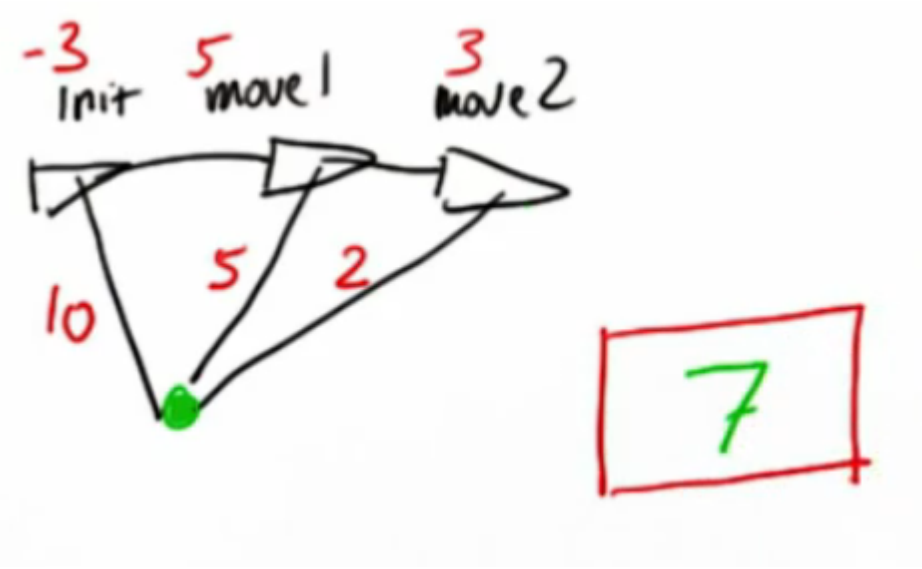
To calculate mu, just apply the formula directly to **Omega** and **Xi**:

```
mu = Omega.inverse()*Xi
return mu
```

A6-18: Landmark Position

And the answer is 7. You can check that result directly because there is no motion or measurement

uncertainty. In the first position you might be tempted to say that there are actually two possibilities for a landmark that is 10 units distant from x_0= -3, namely -13 and 7. However, as you move to x_1 and x_2, the only option that is consistent with subsequent measurements is 7.



The equations used to reach this conclusion are shown in the table below. They are the same as the equations encoded in **Omega** and **Xi**. If you assume a fixed coordinate x_l for the landmark, the only value that works for all equations below is $x_l = 7$.

$x_0 = -3$	$x_1 = x_0 + move1 = -3 + 5 = 2$	$x_2 = x_1 + move2 = 2 + 3 = 5$
$d_0 = \lvert x_l - x_0 \rvert = 10$	$d_1 = \lvert x_l - x_1 \rvert = 5$	$d_2 = \lvert x_l - x_2 \rvert = 2$

A6-19: Expand

First use the expand command to increase the dimensions of **Omega** and **Xi**

```
Omega = Omega.expand(4, 4, [0,1,2], [0,1,2])
Xi = Xi.expand(4, 1, [0, 1, 2], [0])
```

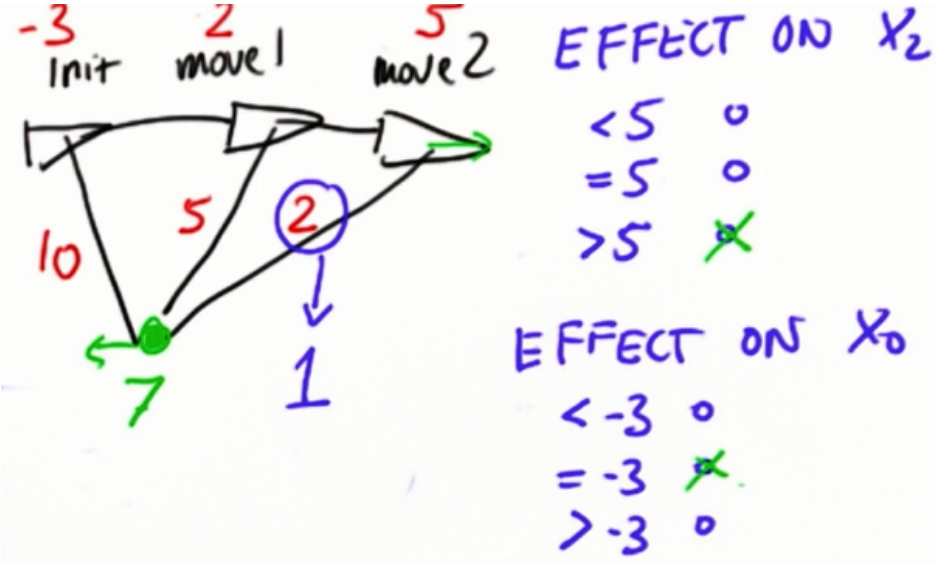
Now add in the measurement constraints for each of the 3 posts and subtract –Z0, -Z1 and –Z2 from the corresponding Xi

```
 Omega += matrix([[1., 0., 0., -1.],[0., 0., 0., 0.],[0., 0., 0., 0.],[-
1., 0., 0., 1.]])
Xi += matrix([[-Z0], [0.], [0.], [Z0]])

Omega += matrix([[0., 0., 0., 0.],[0., 1., 0., -1.],[0., 0., 0., 0.],[0.,
```

```
-1., 0., 1.]])
    Xi += matrix([[0.], [-Z1], [0.], [Z1]])

    Omega += matrix([[0., 0., 0., 0.],[0., 0., 0., 0.],[0., 0., 1., -1.],[0.,
0., -1., 1.]])
    Xi += matrix([[0.], [0.], [-Z2], [Z2]])
```

A6-20: Introducing Noise

Prior to the change, the result was **`[-3, 2, 5, 7]`**. After the change you get **`[-3, 2.125, 5.5, 6.875]`**



What does this mean?

The initial position doesn't change because the only information you have about the absolute coordinate's location is the initial position anchor of -3. None of the relative rubber bands change the fact that this coordinate is -3. The relative rubber bands affect the other two, but not the initial position.

You can understand the effect on x_2 by thinking about it in terms of the rubber band. By changing the strength of the rubber band to 1, you make it tighter and expect the landmark and the position to move closer together.

So this is a case where the rubber bands don't add up and Graph SLAM really shines. Your method computes $\Omega^{-1}\xi$, which finds the best solution to the relaxation of the rubber bands.

A6-21: Confident Measurements

In the **`Omega`** and **`Xi`** for the second position you simply replace all the 1s with 5.

```
    Omega += matrix([[0., 0., 0., 0.],[0., 0., 0., 0.],[0., 0., 5., -5.],[0.,
0., -5., 5.]])
    Xi += matrix([[0.], [0.], [-Z2*5], [Z2*5]])
```

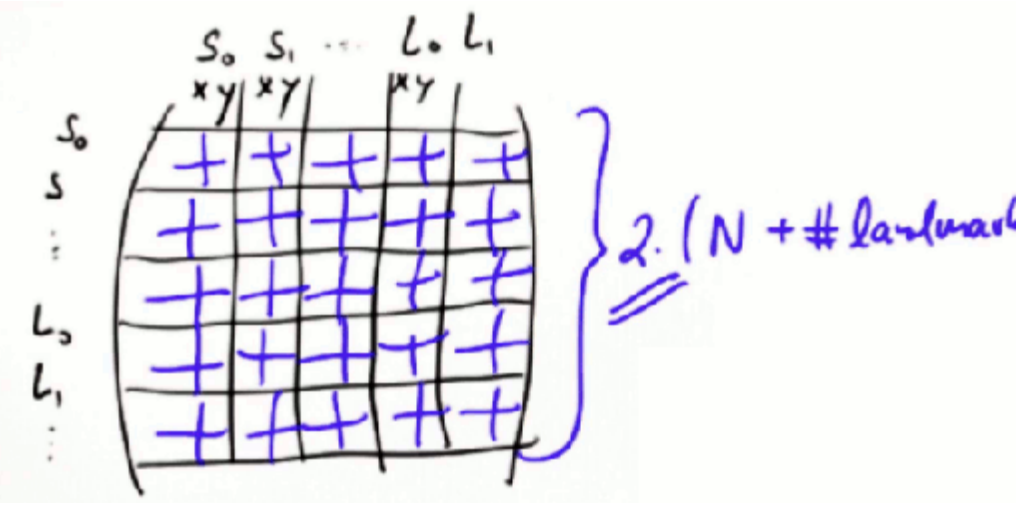
A6-22: Implementing SLAM

Here is Sebastians solution!

You take all the input parameters, and you set the dimension of the matrix **Omega** and the vector **Xi**. The dimension is the length of the path plus the number of landmarks times two, because you are modelling x and y for each of those, in the same data structure. You then create a matrix for **Omega** and a vector for **Xi**, give it appropriate dimensions, and subsequently you introduce a constraint that the initial positions have to be **world_size / 2.0** with a strength value of 1.0. That tells that this has no bearing to the final solution, because it's the only absolute constraint. But you see that it's 1 added in the main diagonal, 1 for x and 1 for y and then the same for Xi.

```
#set the dimension of the filter
dim = 2* (N + num_landmarks)
#make the constraint information matrix and vector
Omega = matrix()
Omega.zero(dim,dim)
Omega.value[0][0] = 1.0
Omega.value[1][1] = 1.0

Xi = matrix()
Xi.zero(dim, 1)
Xi.value[0][0] = world_size / 2
Xi.value[1][0] = world_size / 2
```

It is important to understand how the data structures are set up.



Your positions are marked with S, and landmarks L. each of these have a **x** component and a **y** component. In doing this you are taking this matrix and setting it up not by a matrix of path length

plus the number of landmarks, but each of those becomes a **2 x 2** matrix, where you explicitly retain the **x** and **y** value. So the dimension there is 2 * N (path length) + number of landmarks, and the 2 is the result of modelling xy, xy etc. That is really important for this solution.
You might have chosen a different one, you might have said - I will build one matrix for x and one matrix for y, and then each of those just become a single value, which is closer to what you learned in the class. It's fine in this case. In general it has disadvantages, in that it can not correlate x and y, so for real robot that has real rotations does not work, and Sebastians example is better. But for this specific example these separate matrices would work just fine.
Back to the example. You now process all the items and go through all the measurements of which there are multiple and then through all the combinations of x and y. Motion is handled very much the same way. You then solve as before, and return the solution.
Full solution code:

```
def slam(data, N, num_landmarks, motion_noise, measurement_noise):
    #
    #
    # Add your code here!
    #
    #
    #set the dimension of the filter
    dim = 2 * (N + num_landmarks)
    #make the constraint information matrix and vector
    Omega = matrix()
    Omega.zero(dim,dim)
    Omega.value[0][0] = 1.0
    Omega.value[1][1] = 1.0

    Xi = matrix()
    Xi.zero(dim, 1)
    Xi.value[0][0] = world_size / 2
    Xi.value[1][0] = world_size / 2
    for k in range(len(data)):
        #n is the index of the robots pose in the matrix/vector
        n = k * 2
        measurement = data[k][0]
        motion = data[k][1]
        # integrate measurements
        for i in range(len(measurement)):
            #m is the index of the landmark coordinate in the matrix/vector
            m = 2 * (N + measurement[i][0])
            # update the information matrix according to measurement
            for b in range(2):
                Omega.value[n+b][n+b] +=  1.0 / measurement_noise
                Omega.value[m+b][m+b] +=  1.0 / measurement_noise
                Omega.value[n+b][m+b] += -1.0 / measurement_noise
                Omega.value[m+b][n+b] += -1.0 / measurement_noise
Xi.value[n+b][0]      += -measurement[i][1+b] / measurement_noise
Xi.value[m+b][0]      +=  measurement[i][1+b] / measurement_noise
```

```
    # update the information matrix according to motion
    for b in range(4):
        Omega.value[n+b][n+b]     +=  1.0 / motion_noise
    for b in range(2):
        Omega.value[n+b  ][n+b+2] += -1.0 / motion_noise
        Omega.value[n+b+2][n+b  ] += -1.0 / motion_noise
        Xi.value[n+b  ][0]        += -motion[b] / motion_noise
        Xi.value[n+b+2][0]        +=  motion[b] / motion_noise

mu = Omega.inverse() * Xi
return mu # Make sure you return mu for grading!
```