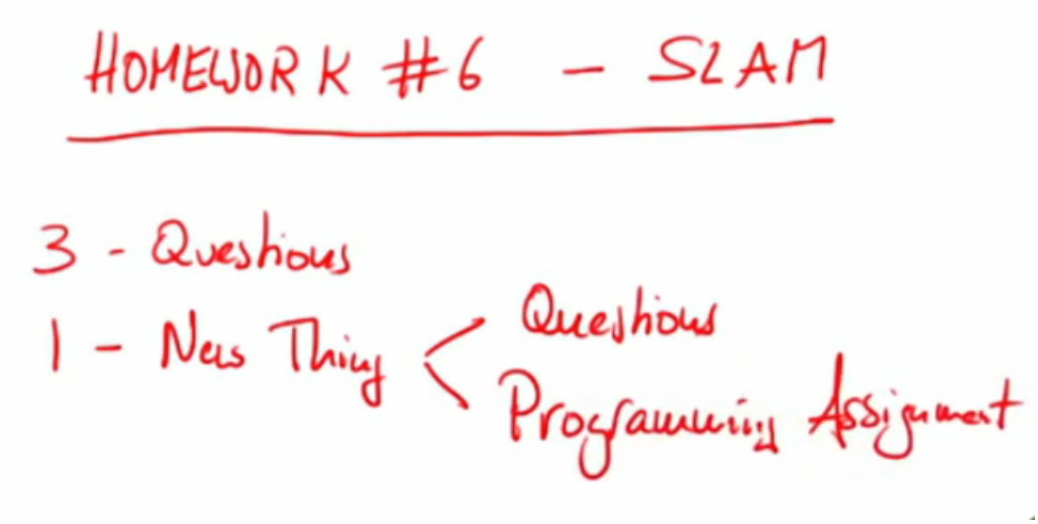
Udacity CS373: Programming a Robotic Car
Unit 6: Homework

Welcome to the homework 6 about SLAM!

In this homework you have to answer some more questions about Graph SLAM algorithm. In this homework a new thing about making it more efficient is introduced towards the end of the assignment.

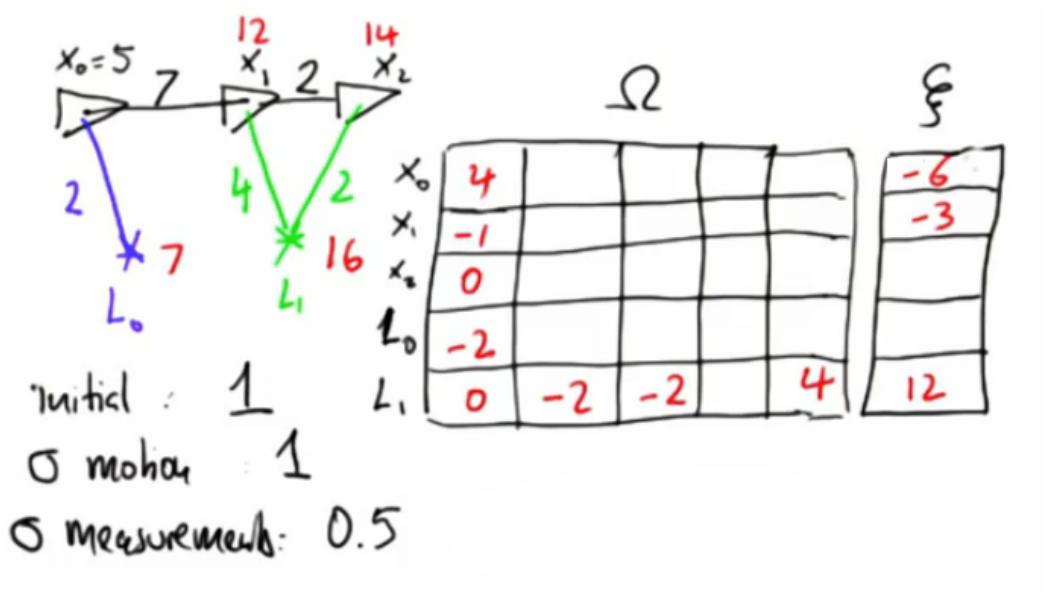


So in this homework you will get 3 questions and 1 new thing which comes with additional questions and a programming assignment.

Homework-1: (Matrix fill-in)

In this question there is a robot that starts at an initial position x_0 = 5. The robot lives in 1-D world even though it is drawn in 2-D. It measures a landmark L_0 with the relative distance of 2. Then it moves to x_1 by taking the step of 7. Now it sees a different landmark L_1 with a measurement value of 4. Finally, it moves again with the motion of 2 and now it sees the same second landmark L_1 with a measurement of 2. Obviously, when you work it out and solve, you will get the best coordinates $x_0 = 12$, $x_1 = 14$ and $L_0 = 7$, $L_1 = 16$, but none of these numbers are inserted in the following question.

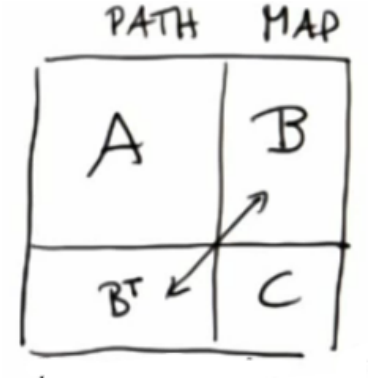
Here is the matrix Ω and the vector ξ. You have to add all these constraints into Ω with a caveat that for initial constraint the strength is 1, motion update has a $\sigma_{motion} = 1$, but the measurement update has $\sigma_{measurement} = 0.5$. Remember, the updates are weighted with $1/\sigma$. You are given some of Ω and ξ values and you have to fill in the missing ones.



You can verify them directly or you can solve the $\mu = \Omega^{[?][?][?][1]}\xi$ equation that should give you expected coordinates.

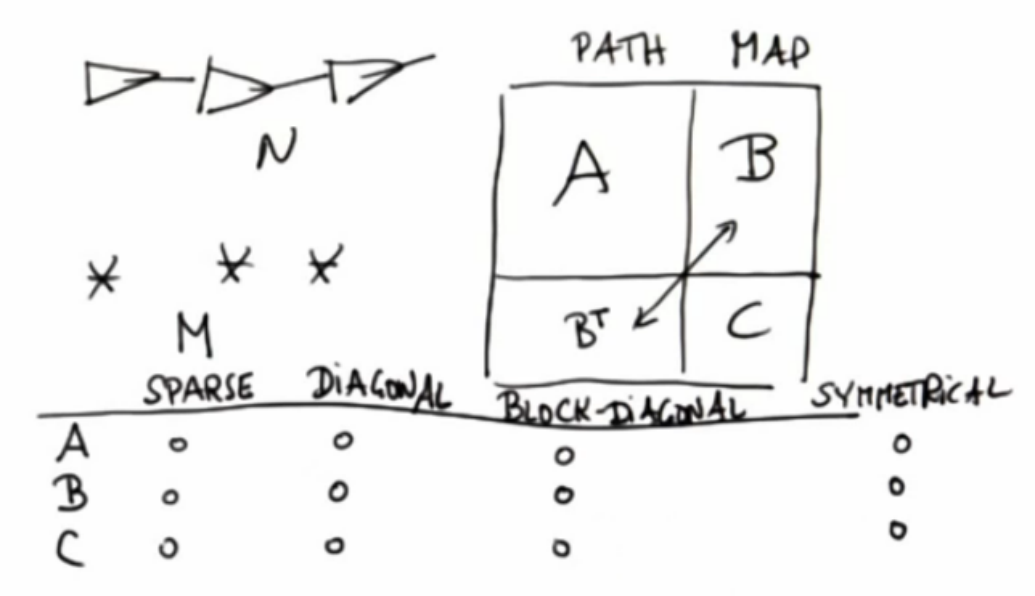
Homework-2: (Matrix properties)

Here is another question. Suppose you have a very long robot path of *N* steps and *M* landmarks. Then you can divide the Ω matrix into four parts.



Here *A* corresponds to path, *C* corresponds to map, *B* and B^T parts (that are symmetrical relative to the main diagonal) link path and map together.

Assume that at each point in time the robot doesn't see more than 3 landmarks. What you can say about the submatrices A, B and C? In particular, check the statement that are true.

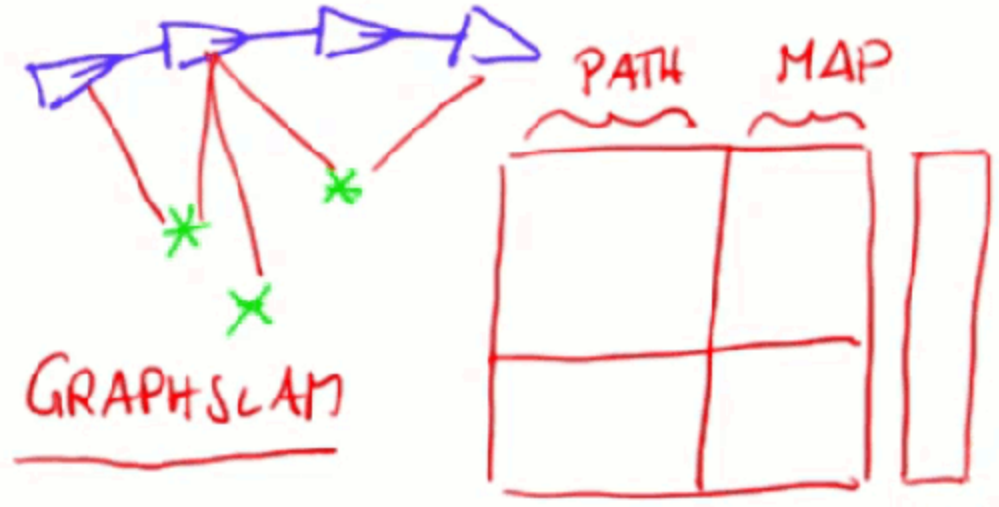


- *Sparse* means the majority of values are zero.
- *Diagonal* means that all the elements that don't lie on the main diagonal are zero.
- *Block diagonal* is a weaker version of diagonal: for some small number k even off-diagonal elements might be non-zero but k is independent of the path length.
- *Symmetrical* means that it can be flipped along the main diagonal so that values remain exactly the same.

They are not mutually exclusive so every combination works.

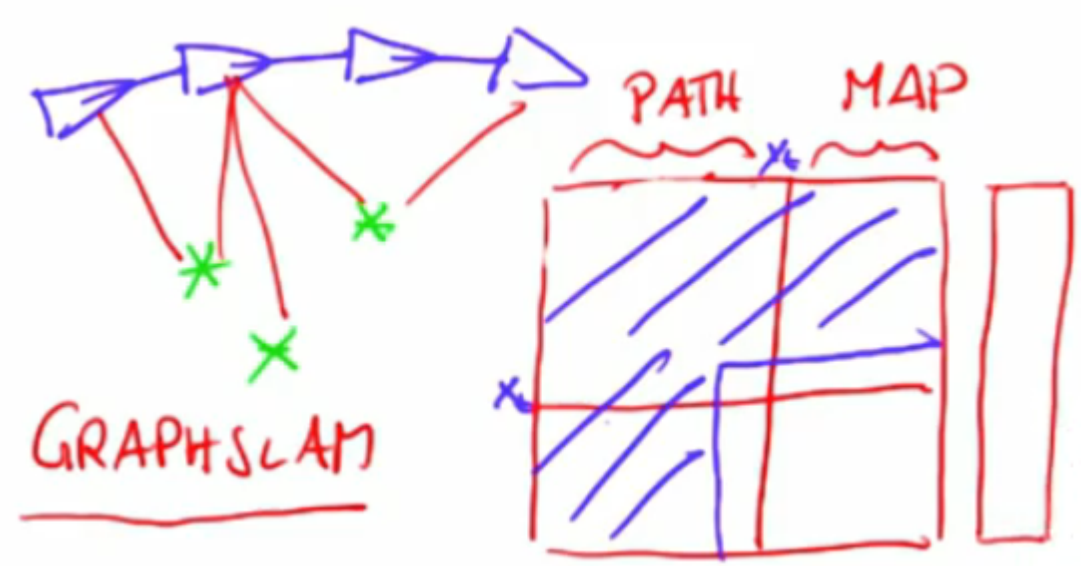
Homework-3: (Online Slam)

This is a really challenging question!
One of the weaknesses of SLAM is as we move along and map the world by seeing these landmarks, the matrix Omega, which is the big thing here, and of course the vector Xi, grow linearly with the length of the path.

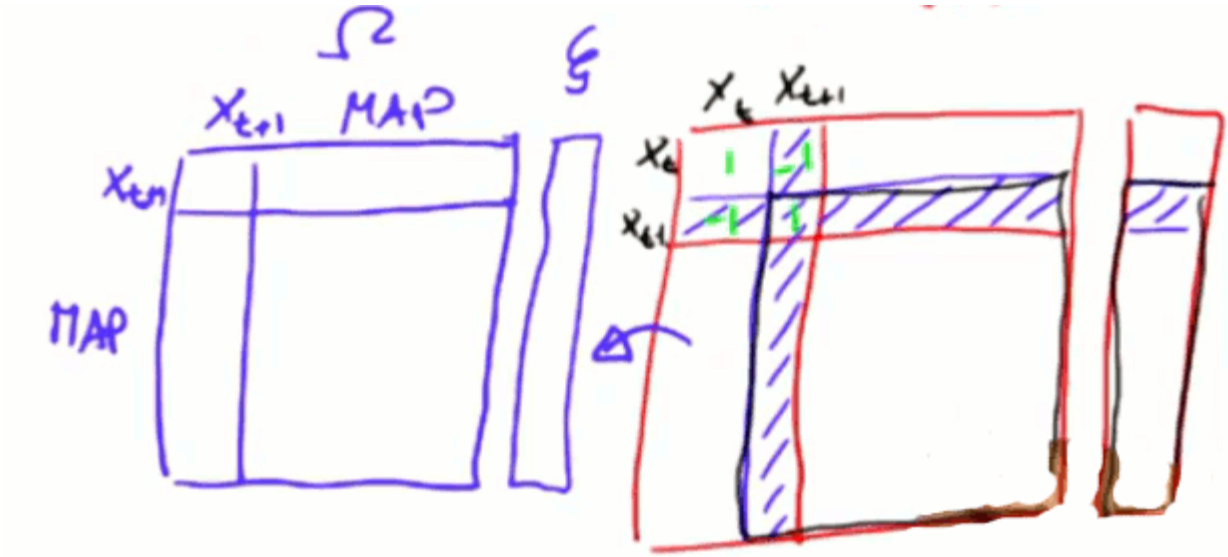


This means that if you go to your local supermarket and buy a robot, and that robot lives for a long

time, say a year or a decade, then the path will be really large, even tho the environment in which it might operate, the map, might be of a fixed size. And that means that robot programmed by Graph SLAM will eventually stop working, because it gets slower and slower. We all know computer operating systems that have that property - the older they are, the slower they are. You don't have to worry how to fix operating systems, you just have to know how to fix SLAM.
The crucial idea that you will learn now and then have to implement from scratch in your software, is following.You can actually reduce the map that you maintain to one that contains only the most recent position in the path. And all the stuff that is outside, streaked with blue lines, that can be safely erased, when you build the map.



Suppose you have a robot position and you have matrix Omega that only contains information pertaining about one position, if the information is one-dimensional, it's just one row and one column, but you can have many different entries for map. Suppose the robot moves to a new position **`x[t+1]`**. Then you do exactly the following - you grow the matrix and vector, by using the **`expand()`** function that you are familiar with from earlier, so that you now have space for your new position. The new area added are the rows streaked with blue, one row if you had one dimensional position, or 2 rows if you had 2-dimensional position. The same applies to the column. And you initialize them all by 0, and then you can apply the regular motion update, that as you know adds 1 to the main diagonal and -1 to the off-diagonal. Same applies on the right side to the vector Xi.
That is a traditional motion update, but that runs the risk that your path increases. Therefore you go back to the left form and make the **`x[t+1]`** survive.



Simplified speaking, you might think about doing this by just cutting out the new submatrix and the subvector, however if you do this, as you can easily verify, that submatrix does not give you the correct answer. And here is where the meat is. You now cut out 3 submatrices or sub values from the full matrix on the right side. One from the top, let's call it A, one in the corner, it's a single element for a 1-dimensional robot, let's call that B, and one called C from the vector. And it is easy to see that the one on the side is A^{-1}. These variables carry a lot of importance, you can not just erase them, but you can fold them back into the surviving matrix by the following simple operation. Let's call the surviving matrix be called **Omega_prime** and the surviving vector called **Xi_prime**. And you can get the matrix **Omega_prime** and **Xi_prime** by using a function **take()** from your matrix library. You have to take a look and learn how to make **take()** get exactly the elements that you need. And then you modify **Omega_prime** and **Xi_prime** with a following piece of math.

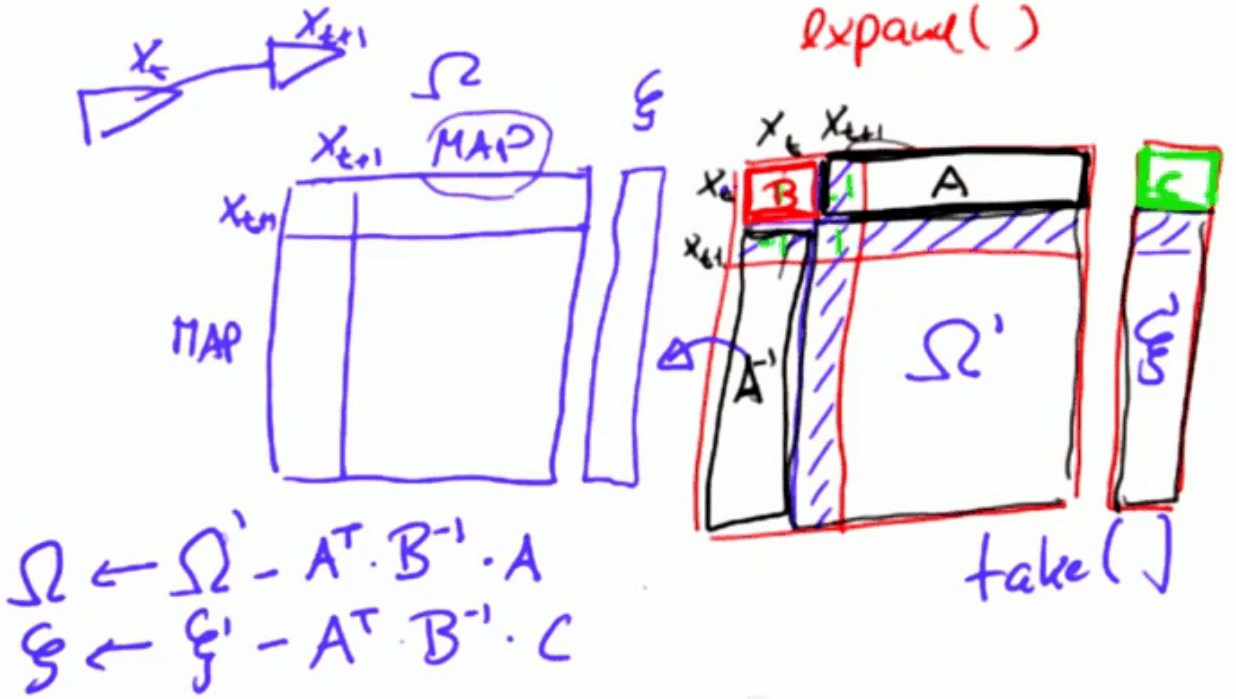
$$\mathbf{Omega <- Omega\_prime - A^T * B^{-1} * A}$$

If the implementation works correctly, this gives you matrix the same size as **Omega_prime**, and that is what you subtract to arrive at the reduced Omega. The same applies to **Xi_prime**.

$$\mathbf{Xi <- Xi\_prime - A^T * B^{-1} * C}$$

This turns out to be the same as the Gaussian where we technically do away, or we call it - integrate away, the variable $\mathbf{x_1}$. For detailed mathematical proof on this you can see the book "Probabilistic Robotics" by Sebastian Thrun, Wolfram Burgard and Dieter Fox.

The intuition here is, that the values A, B, C do carry an importance and to get rid of those you have to redistribute them into the remaining variables, using the math equations given above. When you do this, you are left with a matrix of the same original dimension, because you first added a pose, and then you subtracted one. And as you see, when you do this many times, the final dimensionality of the multidimensional matrix is determined only by the size of map, plus one row and one column for one-dimensional pose, or 2 and 2 for 2-dimensional. That means SLAM scales to really large environments, because we can do the trick every single time when the robot moves.



Many of you have asked for a challenging programming assignment. Sebastian thinks that you will be busy with it for a while!

You have a piece of code where Sebastian implemented SLAM. And then you run the code with 3 landmarks, 3 time steps, a world size of 100, and measurement range of 100. You make data at random, as you are familiar with, and you compute the result. And the result in this case is a vector of **`Omega`**, **`Xi`** and **`mu`** concatenated, then what you might get out looks as follows. There are 3 landmarks, there is a sequence of estimated positions, leading up to the actual robot position. They're both correct, and then there's estimates for where landmarks are. Every time you run it, you get a different answer because your landmarks and your world is different every time.

```
Landmarks:  [[76.0, 64.0], [46.0, 4.0], [9.0, 28.0]]
Robot: [x=88.26149 y=33.99187]

Estimated Pose(s):
    [50.000, 50.000]
    [68.030, 42.428]
    [86.057, 33.766]

Estimated Landmarks:
    [77.235, 64.401]
    [45.894, 4.397]
```

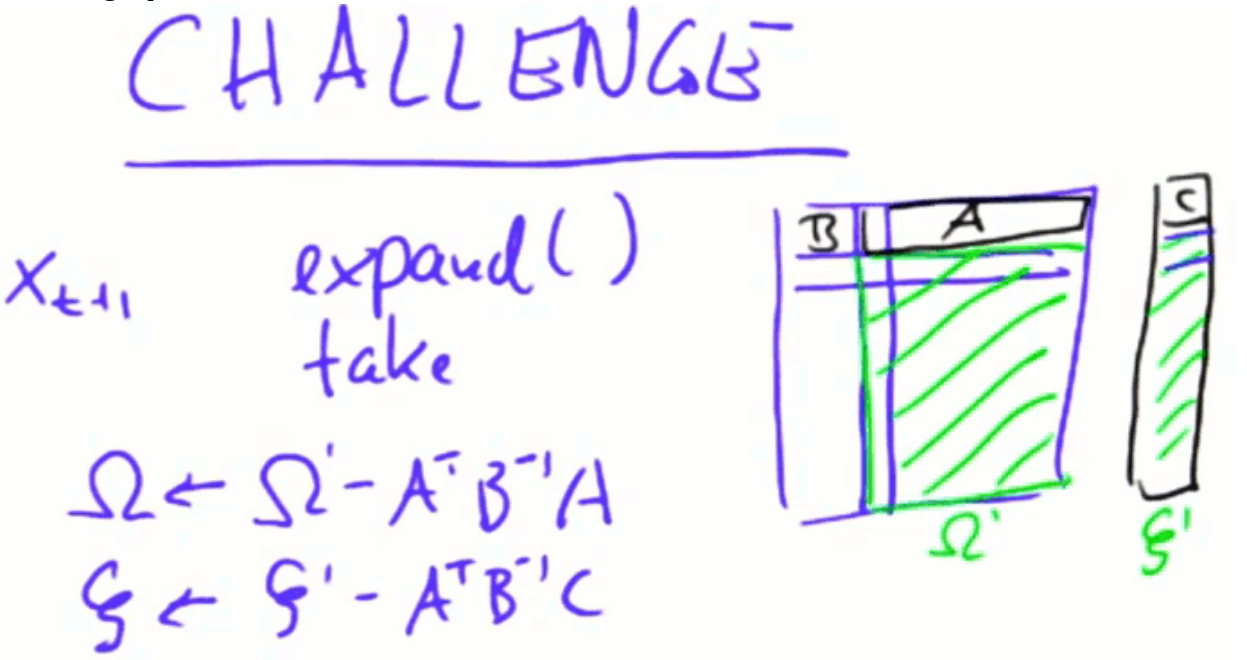
Your task now is to implement a function called online SLAM. It does exactly what you just learned. It resizes the matrix every time a new motion occurs and then goes back to the original size.

In the final result, you get exactly the same estimated pose as for the full SLAM algorithm which is **86.0** and **33.7**.

These are exactly the same number for the estimated pose; that's how you can verify that your solution is correct. The same is true for the estimated landmark. Those coordinates are identical despite the fact that you reduce the size of **`Omega`** and **`Xi`**, but when you print **`Omega`** and **`Xi`**, you

find that the dimensionality is reduced. It's an **`8 x 8`** matrix **`Omega`**, and the number 8 comes because there's 6 coordinates for the landmarks and 1 final robot pose. That is substantially smaller than the matrix you would obtain for the full SLAM case. The same is true for the information vector of size 8.

What you are asked to do is to fill the entire online SLAM routine and to do this, every time you get a new pose, you want to run **`expand()`** to grow the matrix by inserting something right behind the existing pose. You then run **`take()`** to take out the sub-matrix. You also calculate A, B, and for the information vector C, and then as before your reduced Omega your reduced Xi is obtained with the following equations:



If you make no mistake, then you get exactly the same area as you had before, and you can test your routine on arbitrary maps and arbitrary data sets, and it'll just be fine. So, good luck; this is a wonderful programming assignment because it gives you the first really scalable SLAM algorithm and when you implement it, it's actually a major achievement.

It took the scientific feat of SLAM easily 15 years to really discover this form, and ever since then what was really complex and involved lots of Kalman filters, and lots of headaches, became amazingly easy. **So, implement it and you can call yourself a robotic mapper.**