[16-833] Homework 3 : Written Report

Bharath Somayajula

March 31, 2023

Contents

1 Linear SLAM

1.1 Odometry

1.1.1 Measurement Function

Since the odometry results in a change in position, the measurement function is simply

$$h_o(\mathbf{r}^t, \mathbf{r}^{t+1}) = \mathbf{r}^{t+1} - \mathbf{r}^t$$

1.1.2 Jacobian

The Jacobian of measurement function is

$$H_o(\mathbf{r}^t, \mathbf{r}^{t+1}) = \begin{bmatrix} -1 & 0 & 1 & 0 \\ 0 & -1 & 0 & 1 \end{bmatrix}$$

1.2 Landmark

Since landmarks are measured using the relative position of the robot and landmark, the measurement function is simply

$$h_l(\mathbf{r}^t, \mathbf{l}^k) = \mathbf{l}^k - \mathbf{r}^t$$

1.2.1 Jacobian

The Jacobian of measurement function is

$$H_l(\mathbf{r}^t, \mathbf{l}^k) = \begin{bmatrix} -1 & 0 & 1 & 0 \\ 0 & -1 & 0 & 1 \end{bmatrix}$$

1.3 Results for *2d_linear.npz*

1.3.1 *default*

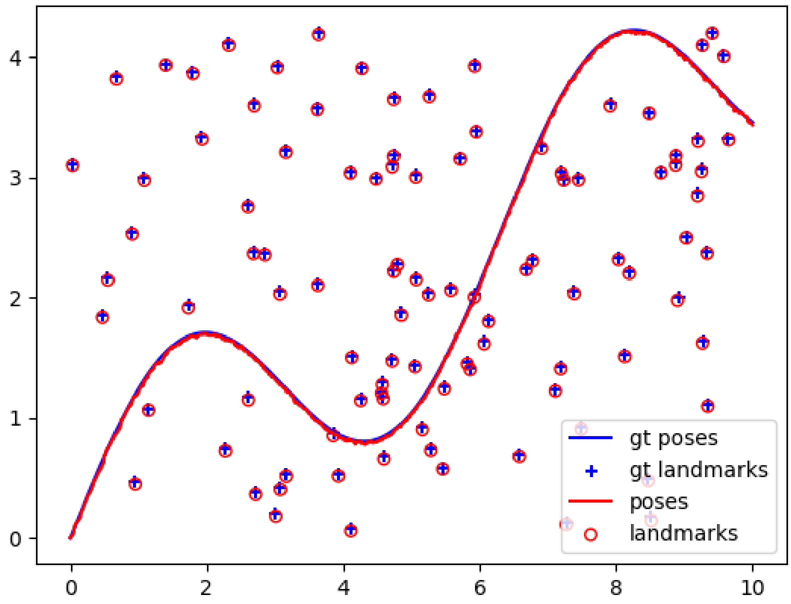


Figure 1: Results with *default*

1.3.2 *pinv*

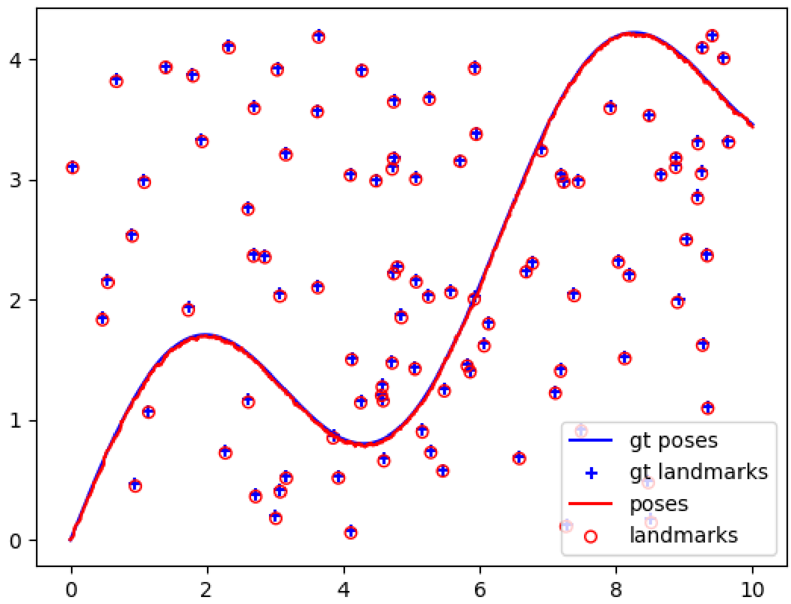
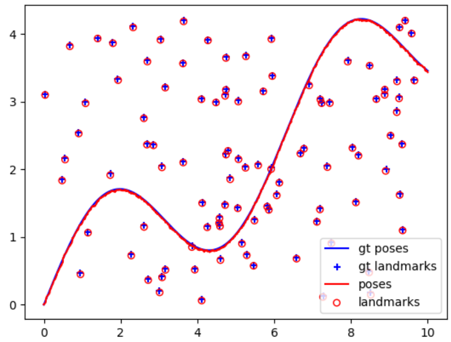
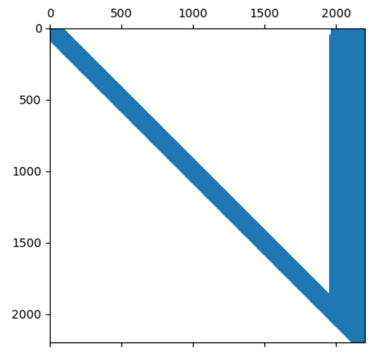


Figure 2: Results with *pinv*

1.3.3 *lu*



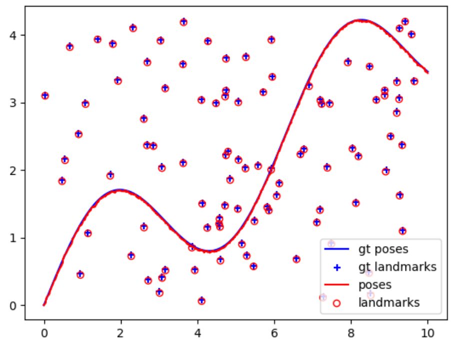
(a) Map



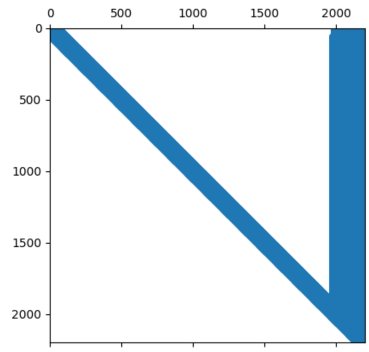
(b) Sparsity

Figure 3: Results with *lu*

1.3.4 *qr*



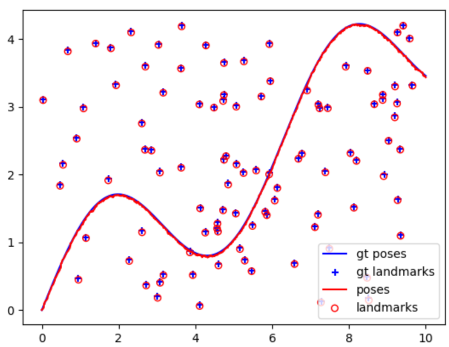
(a) Map



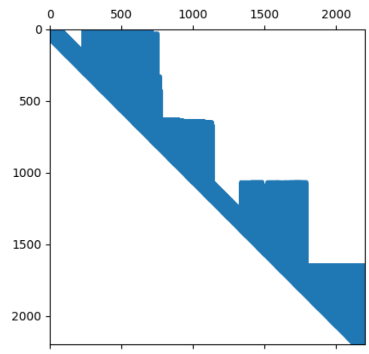
(b) Sparsity

Figure 4: Results with *qr*

1.3.5 *lu_ colamd*



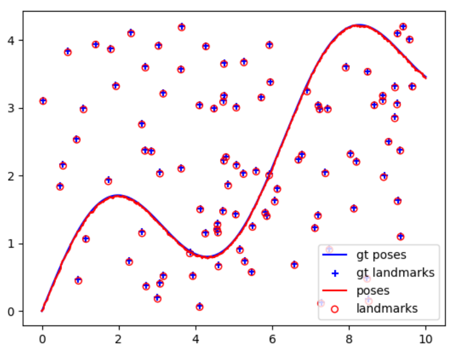
(a) Map



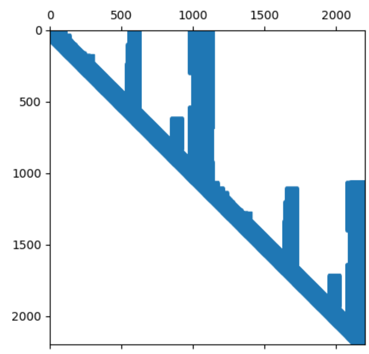
(b) Sparsity

Figure 5: Results with *lu_ colamd*

1.3.6 *qr_ colamd*



(a) Map



(b) Sparsity

Figure 6: Results with *qr_ colamd*

1.3.7 Time

Method	Time(ms)
default	32
pinv	1268
lu	288
qr	15
lu_ colamd	255
qr_ colamd	35

Table 1: Optimization time for each method in milliseconds

1.3.8 Conclusions

1. The *pinv* method is the slowest since it involves computation of matrix inverse

2. The *qr* method is faster than *lu* since QR factorization takes advantage of sparsity of input matrix during factorization process

1.4 Results for *2d_linear_loop.npz*

1.4.1 *default*

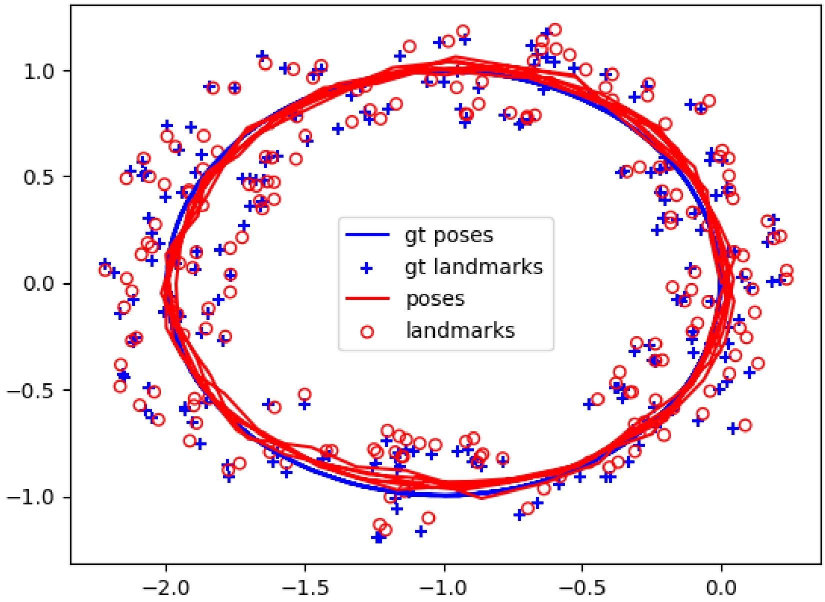


Figure 7: Results with *default*

1.4.2 *pinv*

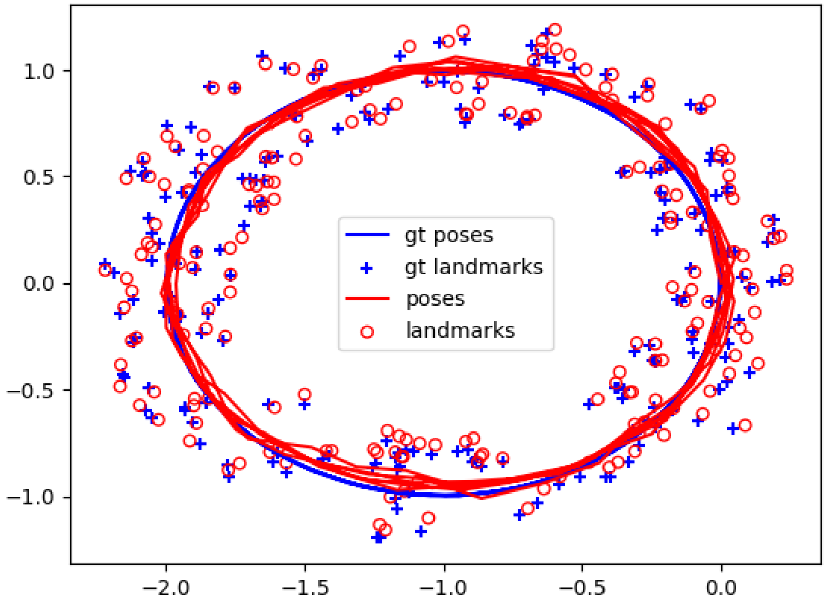
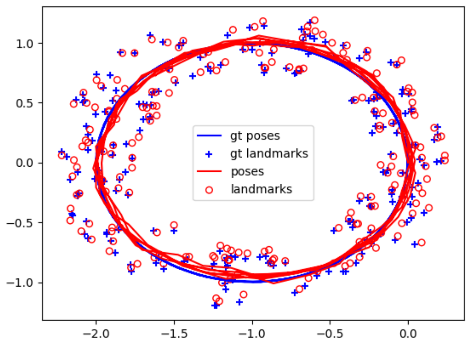
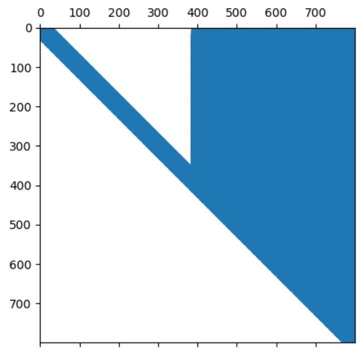


Figure 8: Results with *pinv*

1.4.3 *lu*



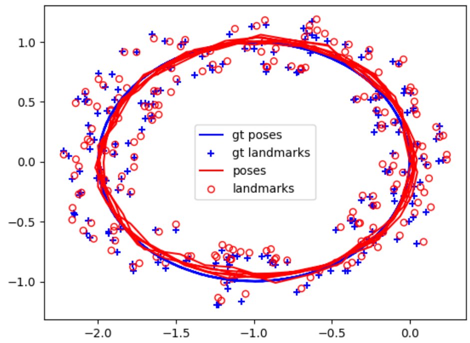
(a) Map



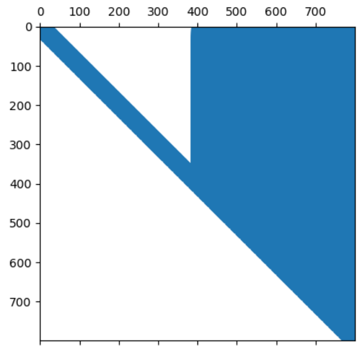
(b) Sparsity

Figure 9: Results with *lu*

1.4.4 *qr*



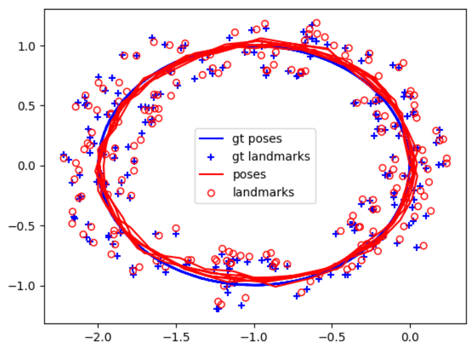
(a) Map



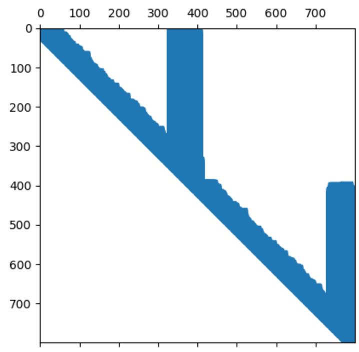
(b) Sparsity

Figure 10: Results with *qr*

1.4.5 *lu_colamd*



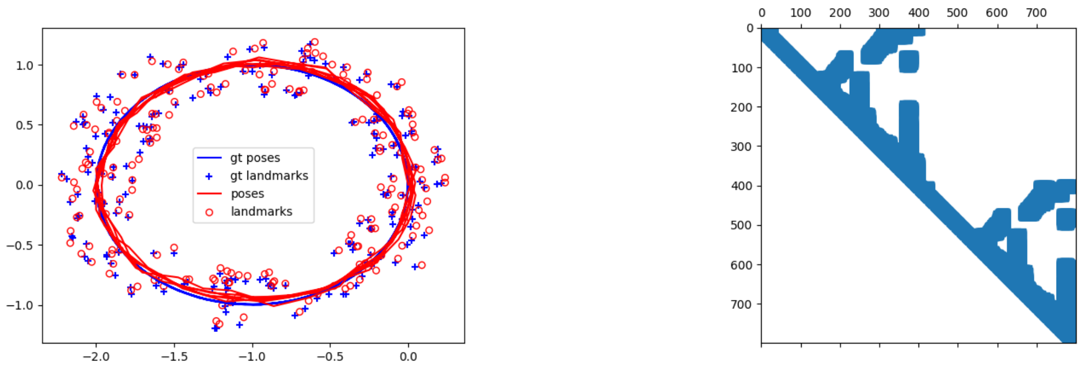
(a) Map



(b) Sparsity

Figure 11: Results with *lu_colamd*

1.4.6 *qr_colamd*



(a) Map (b) Sparsity

Figure 12: Results with *qr_colamd*

1.4.7 Time

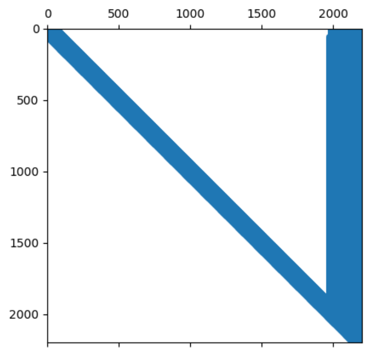
Method	Time(ms)
default	4
pinv	120
lu	190
qr	17
lu_colamd	20
qr_colamd	4

Table 2: Optimization time for each method in milliseconds

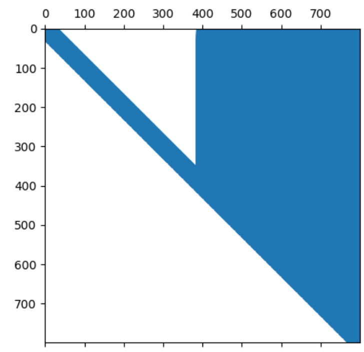
1.4.8 Conclusions

1. The general trend in performance observed for *2d_linear.npz*, i.e, $qr > lu > pinv$ applies to *2d_linear_loop.npz* as well

2. The impact of *colamd* for *2d_linear_loop.npz* is much more significant than in the case of *2d_linear.npz* dataset. This is probably due to A matrix in the *2d_linear_loop.npz* being denser than in *2d_linear.npz* leading to significant gains in performance.



(a) *2d_linear.npz*



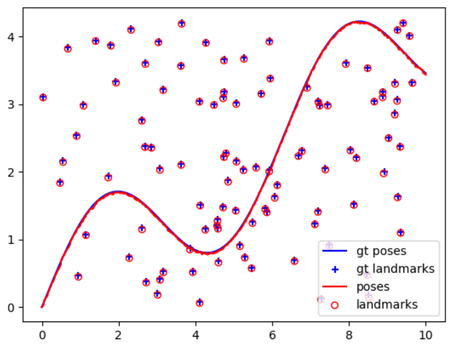
(b) *2d_linear_loop.npz*

Figure 13: Sparsity matrices with *lu* method

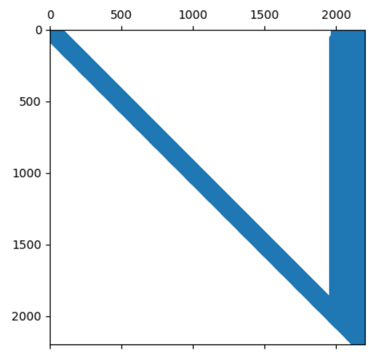
3. The time for *default* and *qr_colamd* methods are surprisingly close for both datasets which *could* mean that *spsolve* relies on QR factorization

1.5 [BONUS] Custom Implementation of Solver

The custom solver for LU factorization is implemented in the *solve_lu_custom_solver* function in *solvers.py*. The function can be used with the *–method lu_custom* argument. The execution time on *2d_linear.npz* is 76 milliseconds which is significantly faster than the *lu.solve* function. The map is shown below:



(a) Map



(b) Sparsity

Figure 14: Results with *lu_custom*

2 Non-Linear SLAM

2.1 Landmark

2.1.1 Jacobian

$$H_l(\mathbf{r}^t, \mathbf{l}^k) = \begin{bmatrix} \frac{l_y - r_y}{(l_x-r_x)^2+(l_y-r_y)^2} & \frac{-(l_x-r_x)}{(l_x-r_x)^2+(l_y-r_y)^2} & \frac{-(l_y-r_y)}{(l_x-r_x)^2+(l_y-r_y)^2} & \frac{l_x-r_x}{(l_x-r_x)^2+(l_y-r_y)^2} \\ \frac{-(l_x-r_x)}{\sqrt{(l_x-r_x)^2+(l_y-r_y)^2}} & \frac{-(l_y-r_y)}{\sqrt{(l_x-r_x)^2+(l_y-r_y)^2}} & \frac{l_x-r_x}{\sqrt{(l_x-r_x)^2+(l_y-r_y)^2}} & \frac{l_y-r_y}{\sqrt{(l_x-r_x)^2+(l_y-r_y)^2}} \end{bmatrix}$$

2.2 Results

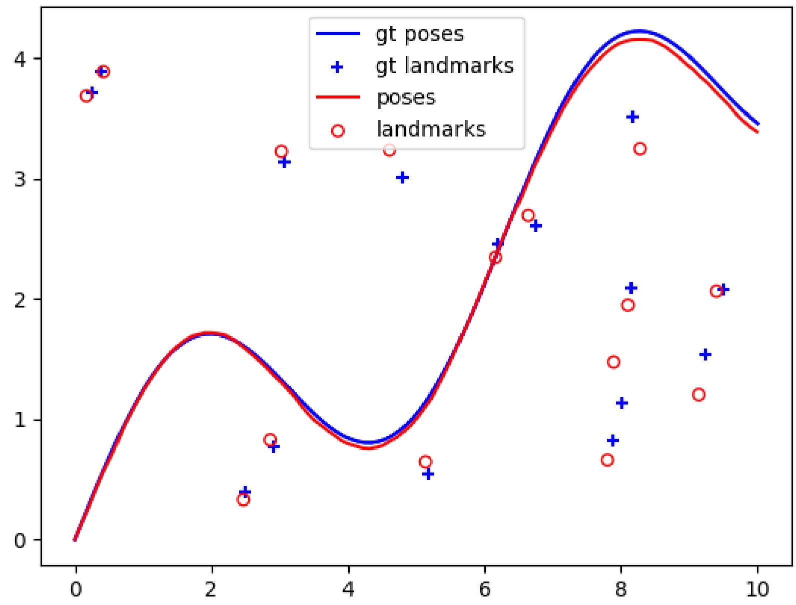


Figure 15: Results with *lu_ colamd* optimizer before optimization

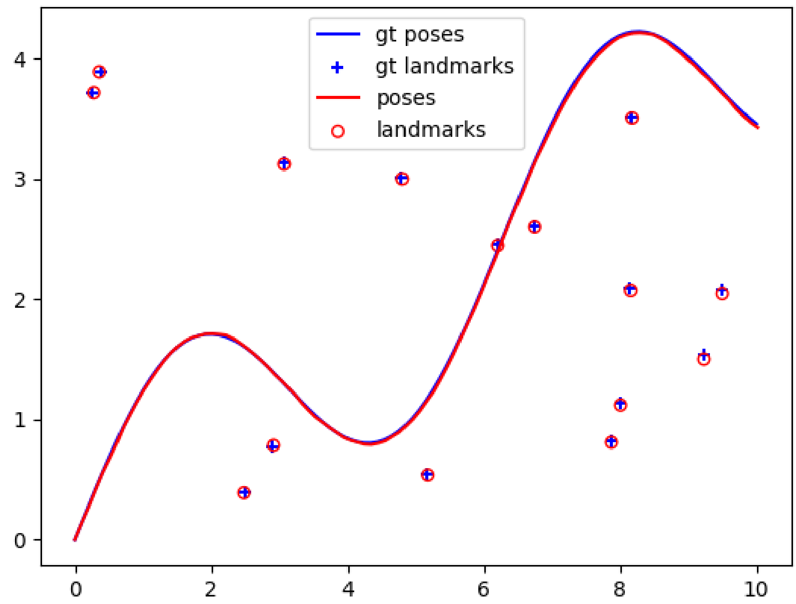


Figure 16: Results with *lu_ colamd* optimizer after optimization

2.3 Comparison

There are two major differences between linear and non-linear optimization approaches

1. Linear optimization involves optimization in one single step whereas non-linear optimization requires refinement of solution over several steps resulting in slower execution

2. In the case of linear optimization problem, the unknown variables that are estimated are the state variables (poses and landmark locations) whereas the unknowns estimated in non-linear optimization problems are the incremental change in the state variables. This requires us to estimate the Jacobian which is time-consuming compared to linear optimization approach.