

16-833: Robot Localization and Mapping, Fall 2024

**Solution 3 - Linear and Nonlinear
SLAM Solvers**

1 2D Linear SLAM

1.1 Measurement function (10 points)

Given robot poses $\mathbf{r}^t = [r_x^t, r_y^t]^\top$ and $\mathbf{r}^{t+1} = [r_x^{t+1}, r_y^{t+1}]^\top$ at time t and $t + 1$, write out the measurement function and its Jacobian. (5 points)

$$h_o(\mathbf{r}^t, \mathbf{r}^{t+1}) : \begin{bmatrix} \mathbf{r}_x^{t+1} - \mathbf{r}_x^t \\ \mathbf{r}_y^{t+1} - \mathbf{r}_y^t \end{bmatrix}$$

For the state-vector $\mathbf{s} = [r_x^t, r_y^t, r_x^{t+1}, r_y^{t+1}]^\top$, the Jacobian matrix is,

$$H_o(\mathbf{r}^t, \mathbf{r}^{t+1}) : \begin{bmatrix} \frac{\partial h_1(r^t, r^{t+1})}{\partial r_x^t} & \frac{\partial h_1(r^t, r^{t+1})}{\partial r_y^t} & \frac{\partial h_1(r^t, r^{t+1})}{\partial r_x^{t+1}} & \frac{\partial h_1(r^t, r^{t+1})}{\partial r_y^{t+1}} \\ \frac{\partial h_2(r^t, r^{t+1})}{\partial r_x^t} & \frac{\partial h_2(r^t, r^{t+1})}{\partial r_y^t} & \frac{\partial h_2(r^t, r^{t+1})}{\partial r_x^{t+1}} & \frac{\partial h_2(r^t, r^{t+1})}{\partial r_y^{t+1}} \end{bmatrix}$$
$$H_o(\mathbf{r}^t, \mathbf{r}^{t+1}) : \begin{bmatrix} -1 & 0 & 1 & 0 \\ 0 & -1 & 0 & 1 \end{bmatrix}$$

Similarly, given the robot pose $\mathbf{r}^t = [r_x^t, r_y^t]^\top$ at time t and the k -th landmark $\mathbf{l}^k = [l_x^k, l_y^k]^\top$. (5 points)

$$h_l(\mathbf{r}^t, \mathbf{l}^k) : \begin{bmatrix} \mathbf{r}_x^t - \mathbf{l}_x^k \\ \mathbf{r}_y^t - \mathbf{l}_y^k \end{bmatrix}$$

Similarly, for the state-vector $\mathbf{s} = [r_x^t, r_y^t, l_x^k, l_y^k]^\top$, the Jacobian matrix is,

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

1.2 Build a linear system

Implemented in `linear.py` code.

1.3 Solvers

Implemented in `solvers.py` code.

1.4 Exploit sparsity

1.4.2

(Bonus) Implemented successfully in `solvers.py` file.

Functions: `forward_substitution(L, b)` and `backward_substitution(U, y)`

1.4.4

Table 1 shows the visualization plot and average run-time results for multiple matrix decomposition methods for `2d_linear.npz` trajectory.

Efficiency Analysis: From the results, it is evident that the QR and QR with `colamd` methods are slower compared to LU and LU with `colamd`. The primary reasons are:

- **QR Factorization:** This method involves complex orthogonal transformations, making it computationally expensive, although it provides better numerical stability.
- **LU Factorization:** Generally faster because it relies on simpler row operations and that U is an upper triangular matrix.

Impact of `colamd` Reordering

Using NATURAL ordering often results in significant fill-in, making upper triangular matrices denser and increasing computational cost. On the other hand, COLAMD reordering reduces fill-in, creating sparser matrices and improving efficiency, though it does introduce some overhead due to the reordering process. This overhead can account for differences in computation time.

1.4.5

Table 2 Compares the number of *measurement* and *pose* states for `2d_linear.npz` and `2d_linear_loop.npz`.

Table 3 Compares the average time for `2d_linear_loop.npz` and `2d_linear.npz`.

Table 4 shows the visualization plot and average run-time results for multiple matrix decomposition methods for `2d_linear_loop.npz` trajectory.

Observations and Reasoning

The solvers generally take less time on average for the loop trajectory compared to the linear one, likely because the loop trajectory involves fewer state variables, such as measurements and poses. Additionally, COLAMD ordering proves to be more efficient than NATURAL ordering for both LU and QR factorizations in the loop case. This could be because, with the robot following a loop, the matrix A becomes denser, and COLAMD is effective at reordering the matrix to increase sparsity, thereby improving computational efficiency.

Another reasoning is that COLAMD is particularly advantageous for looped trajectories because the increased structural dependencies make the matrix more complex. COLAMD reduces fill-in effectively in these scenarios, optimizing performance. However, for a simpler linear trajectory where the matrix is already relatively sparse, the overhead introduced by COLAMD reordering may not provide any significant benefit, making it slower compared to the NATURAL ordering.

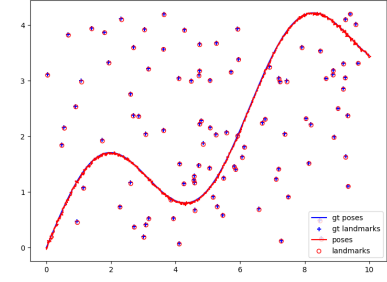
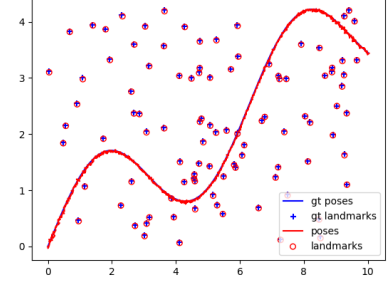
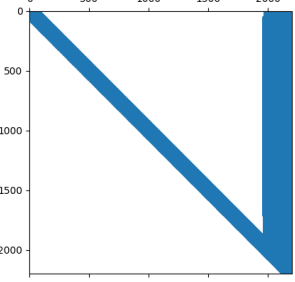
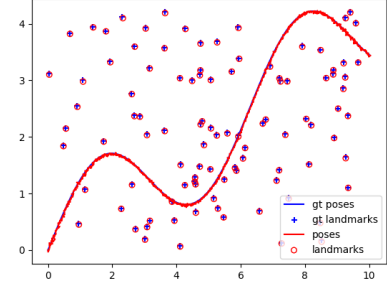
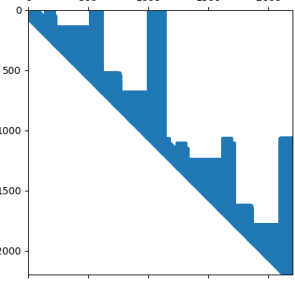
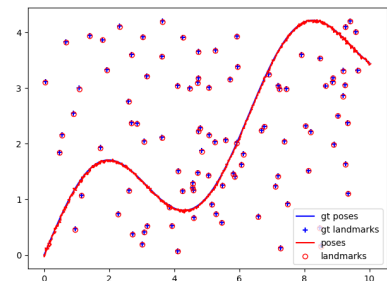
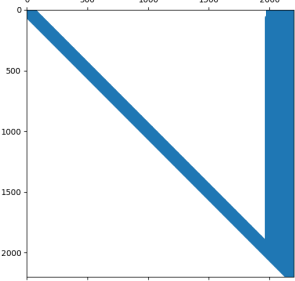
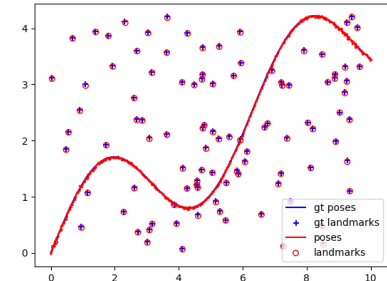
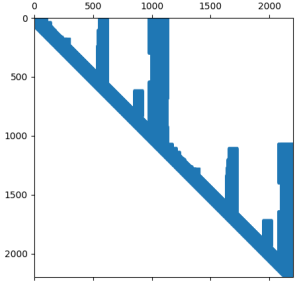
Method	Average Time	Visualization	U/R
pinv	1.75s		None
lu	0.0238s		
lu_colamd	0.0318s		
qr	0.387s		
qr_colamd	0.2551		

Table 1: Visualization and Average Time for 2d_linear.npz

Trajectory	Number of States
2d_linear	File :../data/2d_linear.npz Trajectory Samples :1000 Trajectory Landmark: 100
2d_linear_loop	File :../data/2d_linear_loop.npz Trajectory Samples :200 Trajectory Landmark: 200

Table 2: Number of states for both the trajectories

Trajectory	Average Time
2d_linear	Applying pinv pinv takes 1.7801127433776855s on average Applying lu lu takes 0.02363872528076172s on average Applying qr qr takes 0.3790404796600342s on average Applying lu_colamd lu_colamd takes 0.03342247009277344s on average Applying qr_colamd qr_colamd takes 0.26108336448669434s on average
2d_linear_loop	Applying pinv pinv takes 0.15702128410339355s on average Applying lu lu takes 0.016137123107910156s on average Applying qr qr takes 0.2830636501312256s on average Applying lu_colamd lu_colamd takes 0.0037386417388916016s on average Applying qr_colamd qr_colamd takes 0.012253046035766602s on average

Table 3: Average Time for both the trajectories

2 2D Nonlinear SLAM

2.1 Measurement function (10 points)

1. Implemented in **nonlinear.py** code.
2. Jacobian of the nonlinear landmark function,

$$h_l(\mathbf{r}^t, \mathbf{l}^k) = \begin{bmatrix} \text{atan2}(l_y^k - r_y^t, l_x^k - r_x^t) \\ \left((l_x^k - r_x^t)^2 + (l_y^k - r_y^t)^2 \right)^{\frac{1}{2}} \end{bmatrix} = \begin{bmatrix} \theta \\ d \end{bmatrix}. \quad (1)$$

$$H_l = \begin{bmatrix} \frac{\partial \theta}{\partial r_x^t} & \frac{\partial \theta}{\partial r_y^t} & \frac{\partial \theta}{\partial l_x^k} & \frac{\partial \theta}{\partial l_y^k} \\ \frac{\partial d}{\partial r_x^t} & \frac{\partial d}{\partial r_y^t} & \frac{\partial d}{\partial l_x^k} & \frac{\partial d}{\partial l_y^k} \end{bmatrix}$$

After solving the partial differential equations, we get

$$H_l = \begin{bmatrix} \frac{l_y - y_t}{(l_x - x_t)^2 + (l_y - y_t)^2} & -\frac{l_x - x_t}{(l_x - x_t)^2 + (l_y - y_t)^2} & -\frac{l_y - y_t}{(l_x - x_t)^2 + (l_y - y_t)^2} & \frac{l_x - x_t}{(l_x - x_t)^2 + (l_y - y_t)^2} \\ \frac{x_t - l_x}{\sqrt{(l_x - x_t)^2 + (l_y - y_t)^2}} & \frac{y_t - l_y}{\sqrt{(l_x - x_t)^2 + (l_y - y_t)^2}} & \frac{l_x - x_t}{\sqrt{(l_x - x_t)^2 + (l_y - y_t)^2}} & \frac{l_y - y_t}{\sqrt{(l_x - x_t)^2 + (l_y - y_t)^2}} \end{bmatrix}$$

Code is implemented in **nonlinear.py** code.

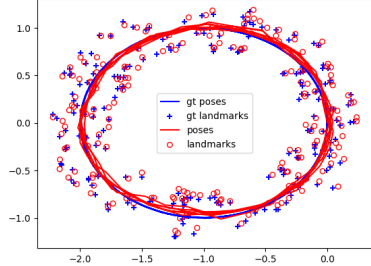
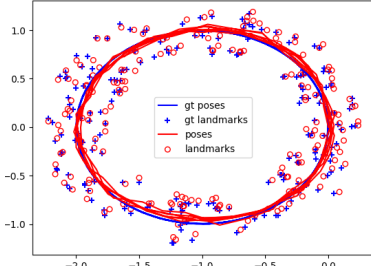
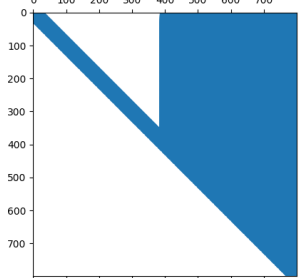
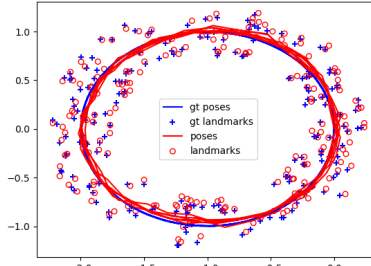
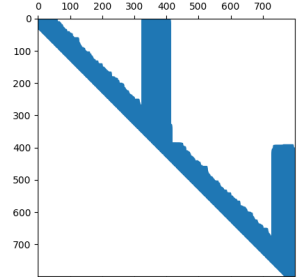
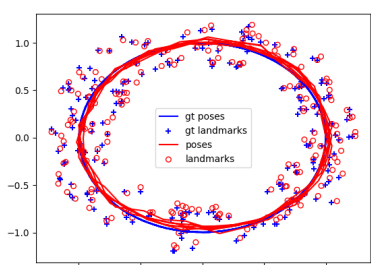
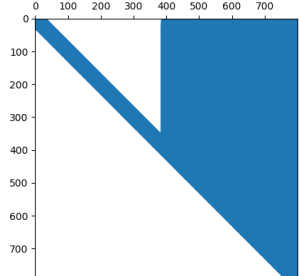
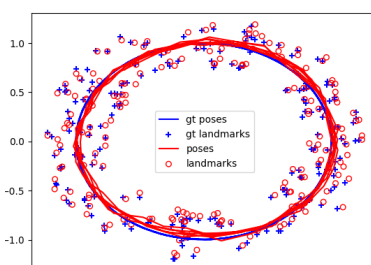
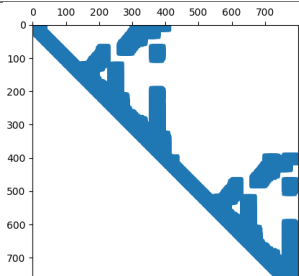
Method	Average Time	Visualization	U/R
pinv	0.157s		None
lu	0.0161s		
lu_colamd	0.00373s		
qr	0.283s		
qr_colamd	0.0122		

Table 4: Visualization and Average Time for 2d_linear_loop.npz.

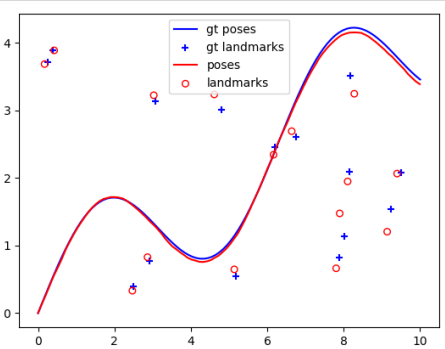
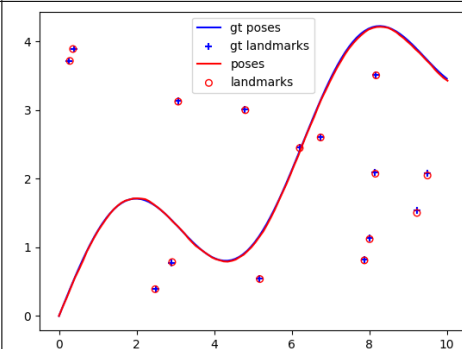
Solver	Before Optimization	After Optimization
qr		

Table 5: Non-linear SLAM using QR solver

2.2 Build a linear system

Code is implemented in `nonlinear.py` code.

2.3 Solver

Table 5 shows the results of `2d_linear_loop.npz` using the 2D nonlinear SLAM method and QR factorization.

In **linear SLAM**, the optimization process assumes a linear relationship between the state variables and directly minimizes the error between predicted and observed measurements. Here, we optimize the state variables (measurements and poses) directly, as the system does not exhibit non-linearity.

Whereas, for **2D non-linear SLAM**, our measurement model is non-linear. The optimization process first linearizes the system around the current state estimate using a first-order Taylor expansion and solves the linearized problem iteratively (in 10 iterations, as given in our code-base). In this case, we optimize the error between the predicted and measured values, rather than directly optimizing the state variables. A good initial estimate is crucial for non-linear SLAM to ensure convergence and avoid local minima during the optimization.