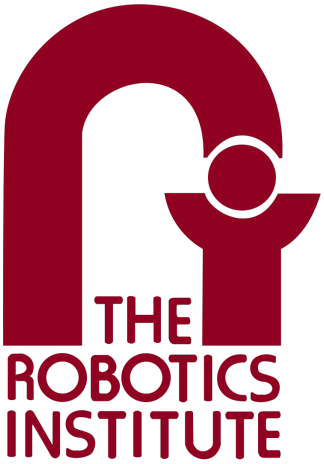
Homework 3



Linear and Nonlinear SLAM Solvers

16-833A Spring 2025

Author: **Boxiang (William) Fu**
Andrew ID: boxiangf
E-mail: *boxiangf@andrew.cmu.edu*

March 22, 2025

Carnegie Mellon University
Robotics Institute

Contents

1 2D Linear SLAM

1.1 Measurement function

We define the robot pose at time t and $t+1$ as $\mathbf{r}^t = [r_x^t, r_y^t]^T$ and $\mathbf{r}^{t+1} = [r_x^{t+1}, r_y^{t+1}]^T$ respectively. The measurement function ($h_o(\mathbf{r}^t, \mathbf{r}^{t+1})$) and its Jacobian ($H_o(\mathbf{r}^t, \mathbf{r}^{t+1})$) is shown below:

$$\begin{aligned} h_o(\mathbf{r}^t, \mathbf{r}^{t+1}) &= \mathbf{r}^{t+1} - \mathbf{r}^t \\ &= \begin{bmatrix} r_x^{t+1} - r_x^t \\ r_y^{t+1} - r_y^t \end{bmatrix} \end{aligned}$$

$$\begin{aligned} H_o(\mathbf{r}^t, \mathbf{r}^{t+1}) &= \frac{\partial h_o}{\partial [\mathbf{r}^t, \mathbf{r}^{t+1}]} \\ &= \begin{bmatrix} \frac{\partial (r_x^{t+1} - r_x^t)}{\partial r_x^t} & \frac{\partial (r_x^{t+1} - r_x^t)}{\partial r_y^t} & \frac{\partial (r_x^{t+1} - r_x^t)}{\partial r_x^{t+1}} & \frac{\partial (r_x^{t+1} - r_x^t)}{\partial r_y^{t+1}} \\ \frac{\partial (r_y^{t+1} - r_y^t)}{\partial r_x^t} & \frac{\partial (r_y^{t+1} - r_y^t)}{\partial r_y^t} & \frac{\partial (r_y^{t+1} - r_y^t)}{\partial r_x^{t+1}} & \frac{\partial (r_y^{t+1} - r_y^t)}{\partial r_y^{t+1}} \end{bmatrix} \\ &= \begin{bmatrix} -1 & 0 & 1 & 0 \\ 0 & -1 & 0 & 1 \end{bmatrix} \end{aligned}$$

Define the k-th landmark as $\mathbf{l}^k = [l_x^k, l_y^k]^T$. The landmark function ($h_l(\mathbf{r}^t, \mathbf{l}^k)$) and its Jacobian ($H_l(\mathbf{r}^t, \mathbf{l}^k)$) is shown below:

$$\begin{aligned} h_l(\mathbf{r}^t, \mathbf{l}^k) &= \mathbf{l}^k - \mathbf{r}^t \\ &= \begin{bmatrix} l_x^k - r_x^t \\ l_y^k - r_y^t \end{bmatrix} \end{aligned}$$

$$\begin{aligned} H_l(\mathbf{r}^t, \mathbf{l}^k) &= \frac{\partial h_l}{\partial [\mathbf{r}^t, \mathbf{l}^k]} \\ &= \begin{bmatrix} \frac{\partial (l_x^k - r_x^t)}{\partial r_x^t} & \frac{\partial (l_x^k - r_x^t)}{\partial r_y^t} & \frac{\partial (l_x^k - r_x^t)}{\partial l_x^k} & \frac{\partial (l_x^k - r_x^t)}{\partial l_y^k} \\ \frac{\partial (l_y^k - r_y^t)}{\partial r_x^t} & \frac{\partial (l_y^k - r_y^t)}{\partial r_y^t} & \frac{\partial (l_y^k - r_y^t)}{\partial l_x^k} & \frac{\partial (l_y^k - r_y^t)}{\partial l_y^k} \end{bmatrix} \\ &= \begin{bmatrix} -1 & 0 & 1 & 0 \\ 0 & -1 & 0 & 1 \end{bmatrix} \end{aligned}$$

1.2 Build a linear system

The implementation is completed under `create_linear_system` in `linear.py`.

1.3 Solvers

The implementation is completed under `solvers.py`.

1.4 Exploit sparsity

The implementation is completed under `solvers.py`. Note that the bonus question was also implemented under the `solve_lu_custom` function without the use of LU's built in solver. A custom function `solve_triangular_custom` was also implemented to solve upper or lower triangular matrix substitution.

Method	Average Time (s)
`qr`	0.4014
`qr_colamd`	0.1815
`lu`	0.0215
`lu_colamd`	0.0364

Table 1: Average runtime of different sparse solvers

1.4.1 `2d_linear.npz` dataset

The runtime performance of different sparse solvers is shown in Table 1. The visualization and analysis are shown below:

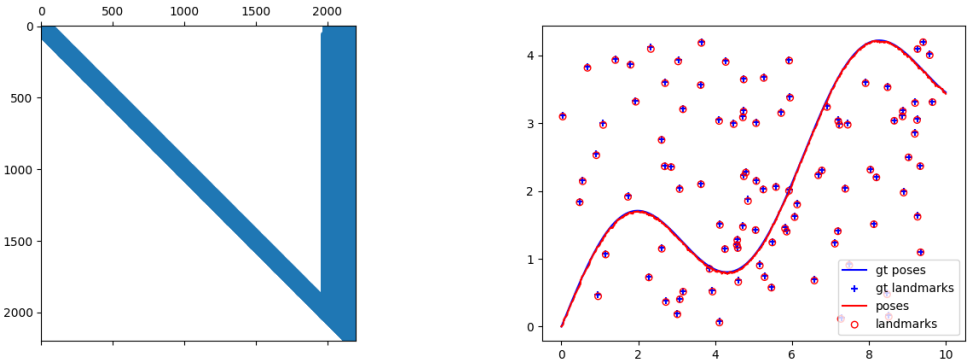


Figure 1: Visualization for `qr` method

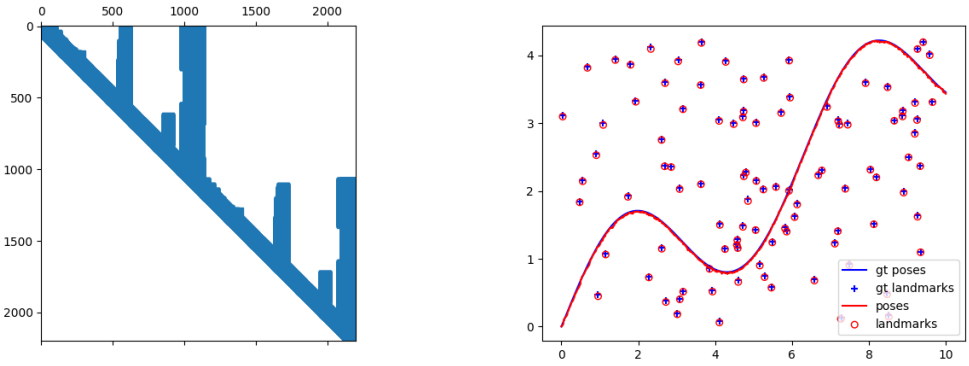


Figure 2: Visualization for `qr_colamd` method

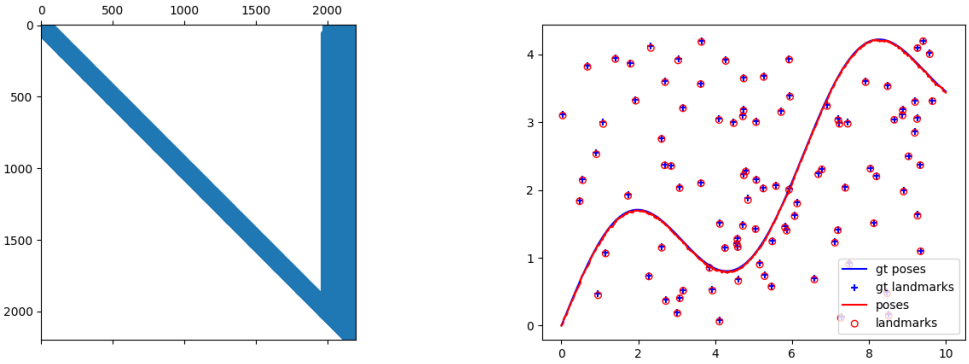


Figure 3: Visualization for `lu` method

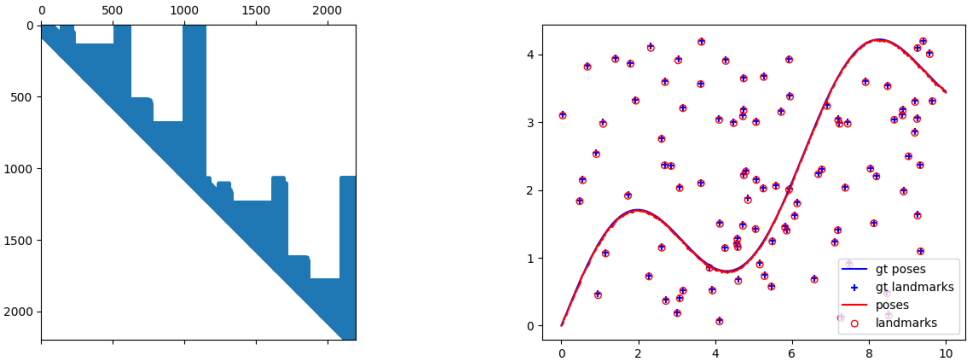


Figure 4: Visualization for `lu_colamd` method

From the above figures and runtime performance, we notice that the LU method computes significantly faster than the QR method. The reason for this is that LU decomposition breaks the matrix down into triangular matrices. This can be done efficiently using only Gaussian elimination row operations. However, for QR decomposition, we need to use either Gram-Schmidt or Householder reflections to compute the orthogonal matrix **Q**. This is about two times more computationally intensive than row operations, hence taking longer to compute (i.e., the operation count is $\frac{2}{3}n^3$ for LU compared to $\frac{4}{3}n^3$ for QR).

However, a disadvantage of LU decomposition is that it may not be numerically stable. If a row operation involves very small or very large values, the decomposition may fail or become unstable. QR decomposition is generally more numerically stable, so a trade-off exists between computational speed and numerical stability. For this particular problem however, we do not see a noticeable difference in accuracy, suggesting that numerical stability is not a issue for this problem.

As for their COLAMD variants, we expect to see a reduction in the average time due to reordering the matrices to make them more sparse (and hence easier to back-substitute and solve). This is indeed what we see for `qr_colamd`, where the average time decreased by almost 50%. However, the time for `lu_colamd` actually increased compared to `lu`. A potential reason is that the COLAMD heuristic guess is a bad guess on an already relatively sparse matrix. Permuting it only made it less sparse (compare Figure 3 with Figure 4). Another reason could be that the COLAMD permutations have a certain overhead computation cost. This overhead is more than the computation saved from the already fast `lu` method, hence increasing the overall average runtime.

1.4.2 `2d_linear_loop.npz` dataset

The runtime performance of different sparse solvers is shown in Table 2.

Method	Average Time (s)
`qr`	0.3542
`qr_colamd`	0.0168
`lu`	0.0171
`lu_colamd`	0.0043

Table 2: Average runtime of different sparse solvers

The visualization and analysis are shown below:

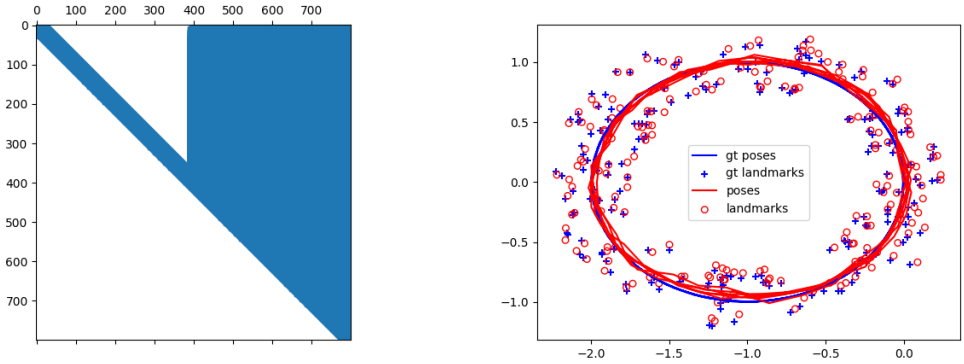


Figure 5: Visualization for `qr` method

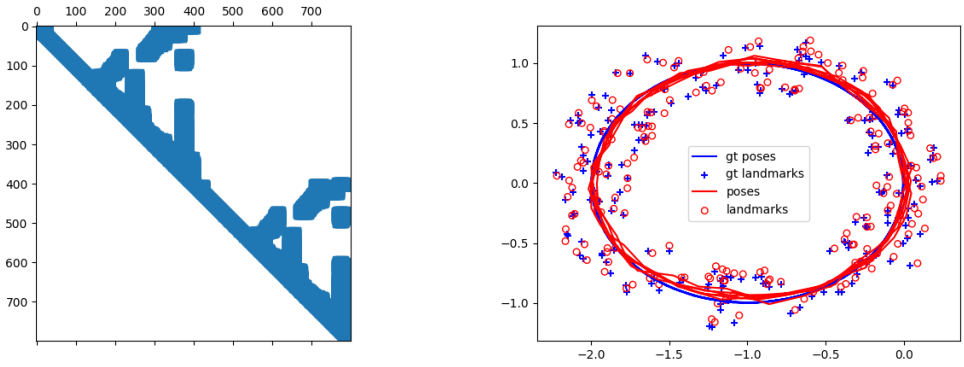


Figure 6: Visualization for `qr_colamd` method

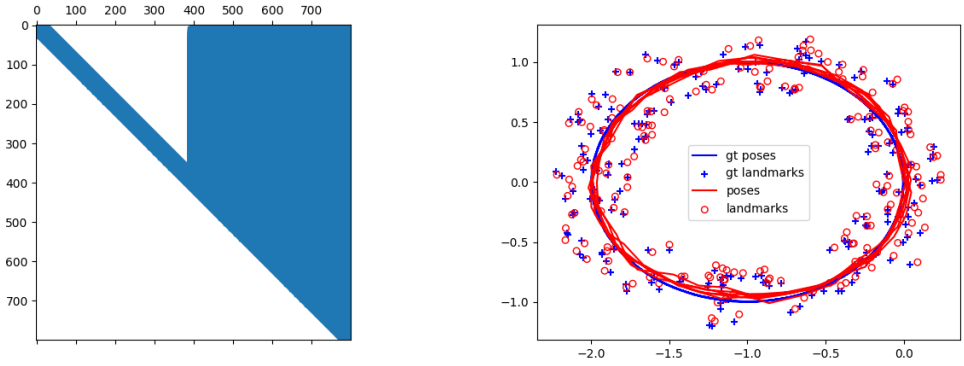


Figure 7: Visualization for `lu` method

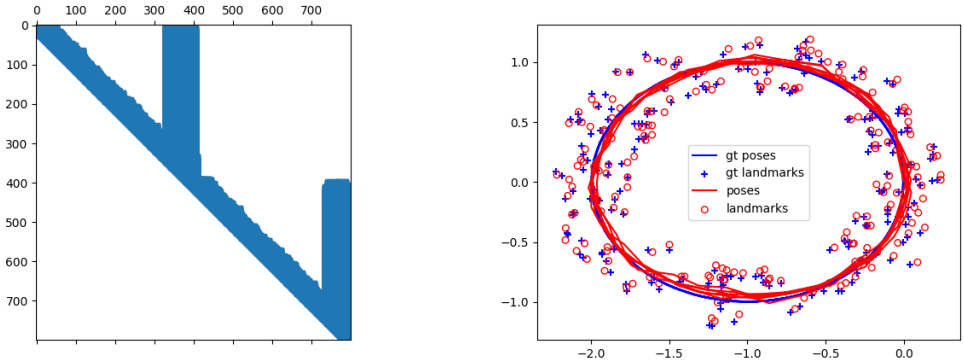


Figure 8: Visualization for `lu_colamd` method

The observations are generally the same as the previous case. The overall average computation time is less compared to the previous case across the board. This is most likely attributed to the circular motion in the dataset revisiting old landmarks and not creating new landmarks (which adds to the dimensionality of the matrix). As with before, LU decomposition is significantly computationally faster than QR decomposition. There is also no discernible difference in accuracy, suggesting that numerical stability is not a issue for this problem. The reason for the faster compute is discussed in the previous section.

What is different however is that both COLAMD versions see a significant runtime performance increase compared to the non-permuted versions. Both LU and QR methods see an order of magnitude decrease in compute time. The reason for this is that the original $\mathbf{U}$ and $\mathbf{R}$ matrices are very dense (see Figures 5 and 7). This makes it very computationally expensive to do back-substitution to solve for the state. The COLAMD heuristic permutations made the $\mathbf{U}$ and $\mathbf{R}$ matrices much more sparse (see Figures 6 and 8), thus requiring much less computation for back-substitution.

2 2D Nonlinear SLAM

2.1 Measurement function

We define the robot pose at time t as $\mathbf{r}^t = [r_x^t, r_y^t]^T$ and the k-th landmark as $\mathbf{l}^k = [l_x^k, l_y^k]^T$. The landmark function Jacobian ($H_l(\mathbf{r}^t, \mathbf{l}^k)$) is shown below:

$$
\begin{aligned}
H_l(\mathbf{r}^t, \mathbf{l}^k) &= \frac{\partial h_l}{\partial [\mathbf{r}^t, \mathbf{l}^k]} \\
&= \begin{bmatrix}
\frac{\partial(\operatorname{atan2}\left(l_y^k - r_y^t, l_x^k - r_x^t\right))}{\partial r_x^t} & \frac{\partial(\operatorname{atan2}\left(l_y^k - r_y^t, l_x^k - r_x^t\right))}{\partial r_y^t} & \frac{\partial(\operatorname{atan2}\left(l_y^k - r_y^t, l_x^k - r_x^t\right))}{\partial l_x^k} & \frac{\partial(\operatorname{atan2}\left(l_y^k - r_y^t, l_x^k - r_x^t\right))}{\partial l_y^k} \\
\frac{\partial\left(\left(l_x^k - r_x^t\right)^2 + \left(l_y^k - r_y^t\right)^2\right)^{\frac{1}{2}}}{\partial r_x^t} & \frac{\partial\left(\left(l_x^k - r_x^t\right)^2 + \left(l_y^k - r_y^t\right)^2\right)^{\frac{1}{2}}}{\partial r_y^t} & \frac{\partial\left(\left(l_x^k - r_x^t\right)^2 + \left(l_y^k - r_y^t\right)^2\right)^{\frac{1}{2}}}{\partial l_x^k} & \frac{\partial\left(\left(l_x^k - r_x^t\right)^2 + \left(l_y^k - r_y^t\right)^2\right)^{\frac{1}{2}}}{\partial l_y^k}
\end{bmatrix} \\
&= \begin{bmatrix}
\frac{l_y^k - r_y^t}{(l_x^k - r_x^t)^2 + (l_y^k - r_y^t)^2} & -\frac{l_x^k - r_x^t}{(l_x^k - r_x^t)^2 + (l_y^k - r_y^t)^2} & -\frac{l_y^k - r_y^t}{(l_x^k - r_x^t)^2 + (l_y^k - r_y^t)^2} & \frac{l_x^k - r_x^t}{(l_x^k - r_x^t)^2 + (l_y^k - r_y^t)^2} \\
-\frac{l_x^k - r_x^t}{\sqrt{(l_x^k - r_x^t)^2 + (l_y^k - r_y^t)^2}} & -\frac{l_y^k - r_y^t}{\sqrt{(l_x^k - r_x^t)^2 + (l_y^k - r_y^t)^2}} & \frac{l_x^k - r_x^t}{\sqrt{(l_x^k - r_x^t)^2 + (l_y^k - r_y^t)^2}} & \frac{l_y^k - r_y^t}{\sqrt{(l_x^k - r_x^t)^2 + (l_y^k - r_y^t)^2}}
\end{bmatrix}
\end{aligned}
$$

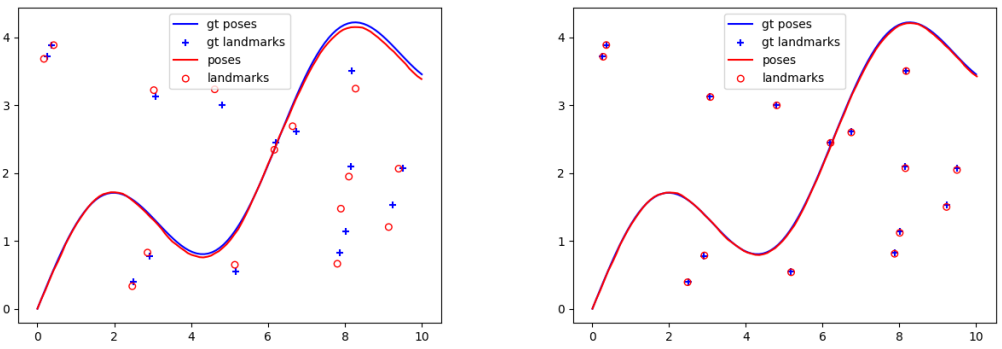
which can be done by applying the chain rule and noting that the derivative of $\arctan(x)$ is $\frac{1}{1+x^2}$ and rearranging to a common denominator. The intermediate steps is trivial and left as an exercise to the reader.

2.2 Build a linear system

The implementation is completed under `create_linear_system` in `nonlinear.py`.

2.3 Solver

The visualization before and after optimization is shown in Figure 9. The solver used is the `lu_custom` solver implemented for the bonus question in Question 1.4.



(a) Before optimization **(b)** After optimization

Figure 9: Visualization for `lu_custom` method

One major difference between the optimization process of the linear and non-linear problem is its ability to obtain a direct solution. For the linear case, a direct solution is possible by using one-shot optimization to solve the normal equation $\operatorname{argmin}_x \|Ax - b\|$. However, for the non-linear case, a direct solution is no longer possible. This is because we do a Taylor expansion around a linearization point. The state obtained from solving the normal equation is now an update vector from the linearization point. Therefore, we need to estimate an initial state linearization point and incrementally update the state vector by repeatedly solving the normal equation and updating the linearization point.

A number of corollaries follow as a result of this. The first is that the optimization would be slower compared to the linear case since we need to perform iterative optimization over the state. This involves using methods such as gradient descent, Gauss-Newton, Levenberg-Marquardt, or Powell's Dog-Leg. Another difference is that for non-linear processes, the Jacobian is no longer constant and updates at each linearization point. This is because the Jacobian is the first-order Taylor expansion term (i.e. gradient) of a non-linear function. This gradient changes as we move along the function, and as such we need to evaluate the Jacobian whenever the linearization point changes.