Carnegie Mellon University

Spring 2022

16-833: Simultaneous Localization and Mapping

HW3 Linear and Non-Linear Solvers

Prof. Michael Kaess

Chinmay Garg

`Andrew ID : chinmayg`

2022-03-30

1. 2D Linear SLAM

1.1 Measurement Function

Given the robot poses at time 't', $r^t = [r_x^t, r_y^t]^\top$ and at time 't+1', $r^{t+1} = [r_x^{t+1}, r_y^{t+1}]^\top$ the linear measurement function the the Jacobian is simply given by :

$$h_o = r^{t+1} - r^t = \begin{bmatrix} r_x^{t+1} - r_x^t \\ r_y^{t+1} - r_y^t \end{bmatrix} \tag{1}$$

$$H_o = \begin{bmatrix} -1 & 0 & 1 & 0 \\ 0 & -1 & 0 & 1 \end{bmatrix} \tag{2}$$

Similarly, the measurement function and Jacobian between the robot pose $r^t = [r_x^t, r_y^t]^\top$ and k-th landmark $l^k = [l_x^k, l_y^k]^{top}$ are given by :

$$h_o = r^{t+1} - r^t = \begin{bmatrix} l_x^k - r_x^t \\ l_y^k - r_y^t \end{bmatrix} \tag{3}$$

$$H_o = \begin{bmatrix} -1 & 0 & 1 & 0 \\ 0 & -1 & 0 & 1 \end{bmatrix} \tag{4}$$

1.2 Build a Linear System

The completed `create_linear_system` function can be found in the file `linear.py`.

1.3 Solvers

The completed solvers for `pseudoinverse`, `LU factorization` and `QR factorization` can be found in `solvers.py`.

1.4 Exploit Sparsity

The implementations of solvers exploiting sparsity using the `COLAMD` heuristic for `LU factorization` and `QR factorization` can be found in `solvers.py`. The bonus implementation for LU factorization can also be found in `solvers.py`.

Bonus

Bonus implementation of forward and backward substitution can be found in the `solve_custom_lu` function in `solvers.py` along with functions for forward and backward substitution as shown in figure 1. Given the Jacobian matrix A and measurements matrix b, we need to solve for $A^\top AX = A^\top b$. LU decomposition with COLAMD permutations given by P (row permutations) and Q (column permutations) can be solved by :

$$PA^\top AQ = LUA^\top A = P^\top LUQ^\top \tag{5}$$

Using above in $A^\top AX = A^\top b$, we get:

$$P^\top LUQ^\top X = A^\top b \Rightarrow LUQ^\top X = PA^\top b \tag{6}$$

Forward Substitution equation : $Lz = PA^\top b$
Backward Substitution equation : $Uy = z$
Applying permutation to get X : $X = Qy$

To run the custom LU factorization, the command will be same as others, but with the method as 'custom_lu':

```
python linear.py <data_path> --method 'custom_lu'
```

```python
def forward_substitution(L, b):
    y = np.zeros_like(b)
    num = y.shape[0]
    y[0] = b[0] / L[0, 0]
    for i in range(1, num):
        temp = b[i]
        for j in range(0, i):
            temp -= L[i, j] * y[j]
        y[i] = temp / L[i, i]
    return y


def backward_substitution(R, b):
    x = np.zeros_like(b)
    num = x.shape[0]
    x[num - 1] = b[num - 1] / R[num - 1, num - 1]
    for i in range(num - 2, -1, -1):
        temp = b[i]
        for j in range(i + 1, num):
            temp -= R[i, j] * x[j]
        x[i] = temp / R[i, i]
    return x
```

(a) Forward and backward substitution functions

```python
def solve_custom_lu_colamd(A, b):
    lu = splu(csc_matrix(A.T @ A), permc_spec="COLAMD")
    L = lu.L
    U = lu.U

    # Permutaion Matrices
    P = csc_matrix((np.ones(A.shape[1]), (lu.perm_r, np.arange(A.shape[1]))))
    Q = csc_matrix((np.ones(A.shape[1]), (np.arange(A.shape[1]), lu.perm_c)))

    # Custom forward and backward substitution (Slower than scipy's)
    z = forward_substitution(L, P @ A.T @ b)
    y = backward_substitution(U, z)
    x = Q @ y

    # In-built forward and backward substitution (Faster)
    # z = spsolve_triangular(L, P @ A.T @ b, lower=True)
    # y = spsolve_triangular(U, z, lower=False)
    # x = Q @ y

    return x, U
```

(b) LU Factorization

Figure 1: Bonus implementation code

Visualizations and Observations (2d_linear.npz)

Pseudo-inverse

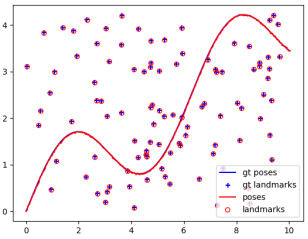
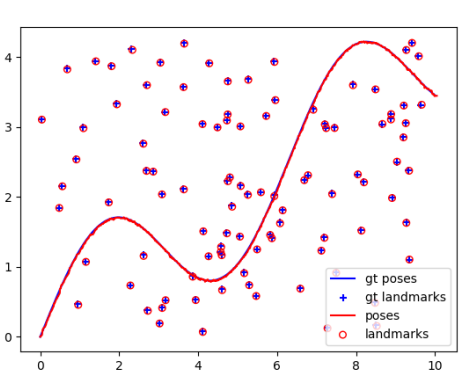
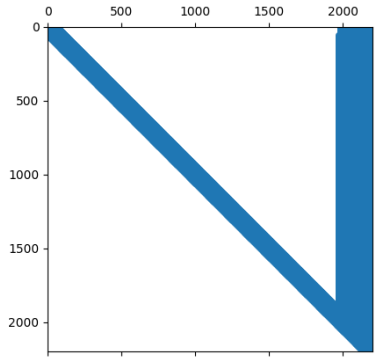


Figure 2: Pseudo-inverse solver Trajectory and Landmark Visualization

LU Factorization (NATURAL and COLAMD)

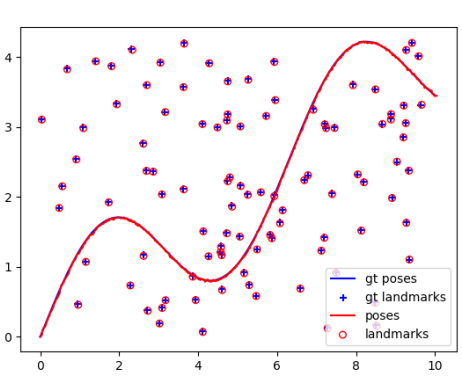


(a) Trajectory and Landmark Visualization

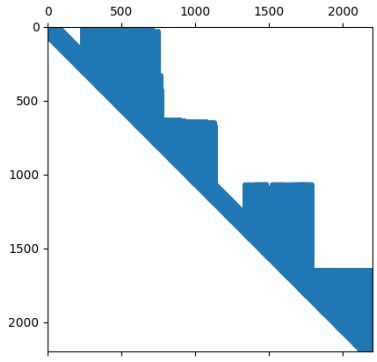


(b) U Matrix sparsity and fill-in

Figure 3: LU Factorization solver



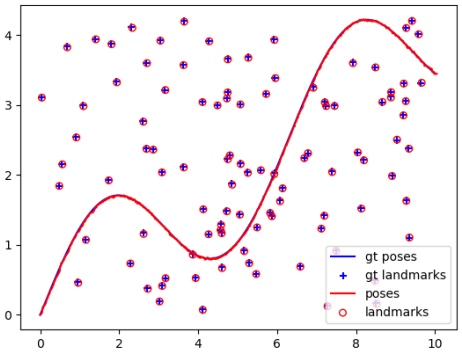
(a) Trajectory and Landmark Visualization



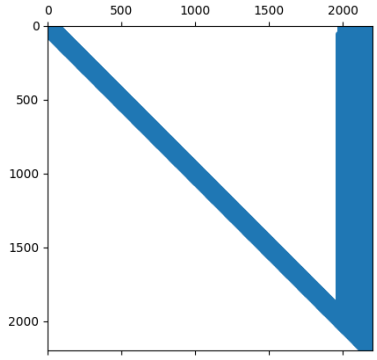
(b) U Matrix sparsity and fill-in

Figure 4: LU COLAMD Factorization solver

QR Factorization (NATURAL and COLAMD)

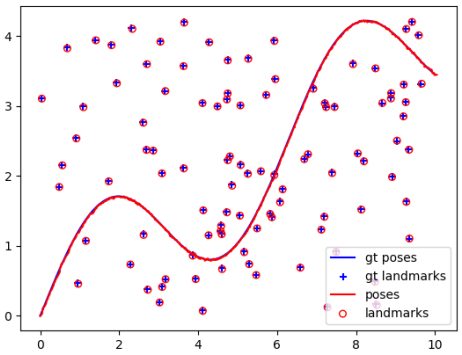


(a) Trajectory and Landmark Visualization

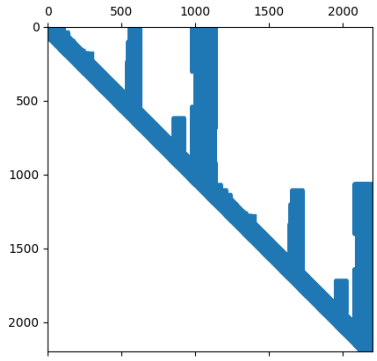


(b) R Matrix sparsity and fill-in

Figure 5: QR Factorization solver



(a) Trajectory and Landmark Visualization



(b) R Matrix sparsity and fill-in

Figure 6: QR COLAMD Factorization solver

It is evident that LU factorization is the fastest with NORMAL permutation, possibly because COLAMD leads to denser matrix. QR factorization is slower than LU with NORMAL permutation, but with COLAMD it is much faster than LU.

Solver	**Time Taken (secs)**
LU	0.208
LU COLAMD	1.526
QR	0.919
QR COLAMD	0.548

Table 1: Linear Times

Visualizations and Observations (2d_linear_loop.npz)

Pseudo-inverse

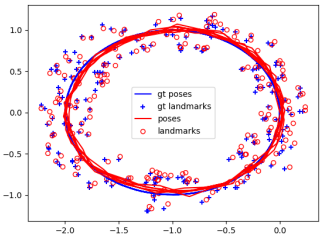
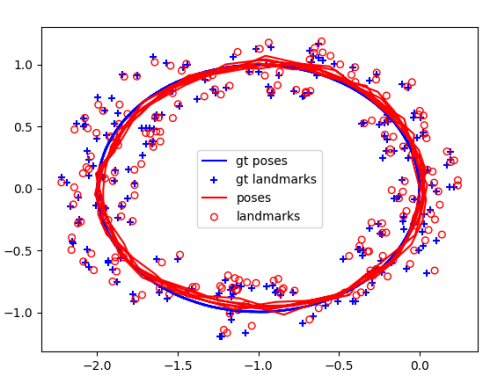
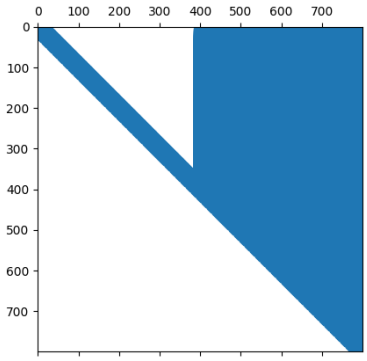


Figure 7: Pseudo-inverse solver Trajectory and Landmark Visualization

LU Factorization (NATURAL and COLAMD)

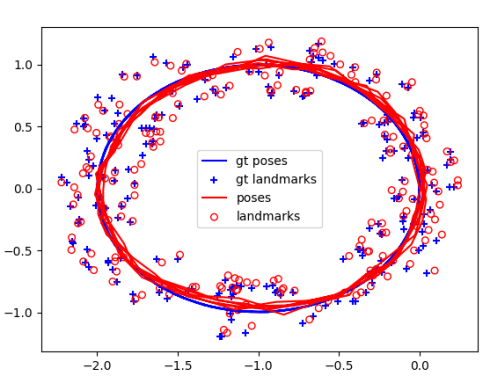


(a) Trajectory and Landmark Visualization

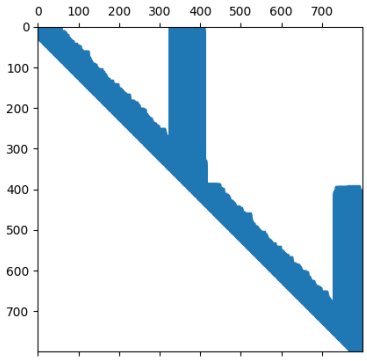


(b) U Matrix sparsity and fill-in

Figure 8: LU Factorization solver



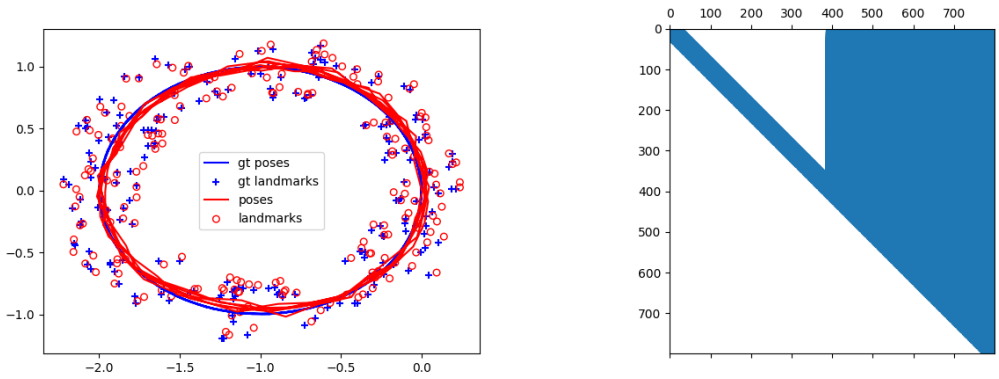
(a) Trajectory and Landmark Visualization



(b) U Matrix sparsity and fill-in

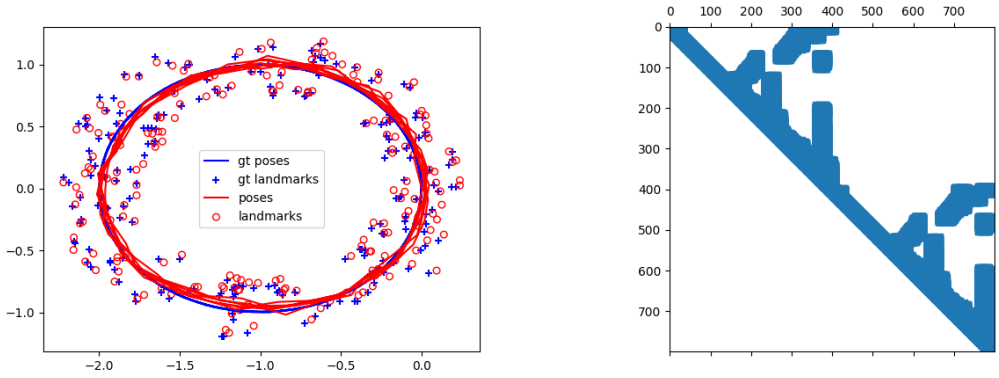
Figure 9: LU COLAMD Factorization

QR Factorization (NATURAL and COLAMD)



(a) Trajectory and Landmark Visualization (b) R Matrix sparsity and fill-in

Figure 10: QR Factorization solver



(a) Trajectory and Landmark Visualization (b) R Matrix sparsity and fill-in

Figure 11: QR COLAMD Factorization solver

For the *linear loop* data LU decomposition is still the fastest, with COLAMD being slightly faster than NORMAL. Moreover, the COLAMD permuted matrix leads to faster computation in this case compared to NATURAL matrix which is possibly because of the more dense information matrix. Unlike *linear* data where LU factorization was fastest, here the most efficient solver here is LU factorization with COLAMD permutation. In this case the computation is faster in general owing to smaller Jacobian compared to *linear* data.

Solver	**Time Taken (secs)**
LU	0.057
LU COLAMD	0.055
QR	0.652
QR COLAMD	0.158

Table 2: Linear Loop Times

2. 2D Non-Linear SLAM

2.1 Measurement Function

Given the following measurement function :

$$h_l(r_t, l^k) = \begin{bmatrix} atan2(l_y^k - r_y^t, l_x^k - r_x^k) \\ \sqrt{(l_y^k - r_y^t)^2 + (l_x^k - r_x^k)^2} \end{bmatrix} = \begin{bmatrix} \theta \\ d \end{bmatrix} \tag{7}$$

The Jacobian H_l for the above measurement function w.r.t $r_x^k, r_y^t, l_x^k, l_y^k$ is :

$$H_l = \begin{bmatrix} \frac{\partial\theta}{\partial r_x^t} & \frac{\partial\theta}{\partial r_y^t} & \frac{\partial\theta}{\partial l_x^k} & \frac{\partial\theta}{\partial l_y^k} \\ \frac{\partial d}{\partial r_x^t} & \frac{\partial d}{\partial r_y^t} & \frac{\partial d}{\partial l_x^k} & \frac{\partial d}{\partial l_y^k} \end{bmatrix} \tag{8}$$

Let $\Delta_x = l_x^k - r_x^t$ and $\Delta_y = l_y^k - r_y^t$, with $D = \sqrt{(\Delta_x)^2 + (\Delta_y)^2}$ the Jacobian is given by:

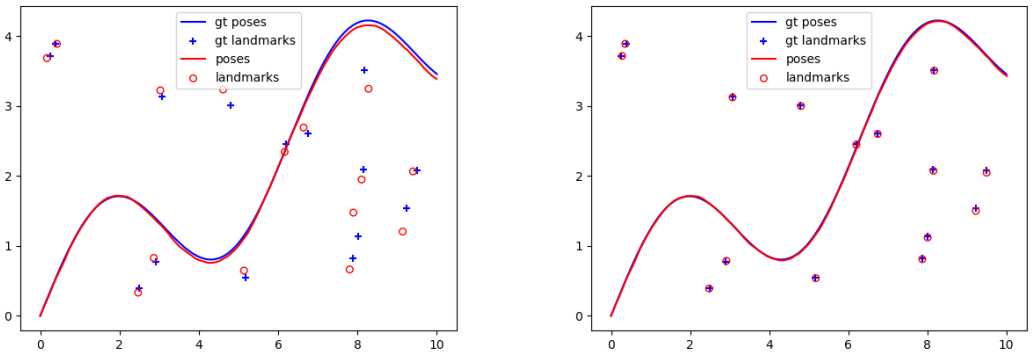
$$H_l = \begin{bmatrix} \frac{\Delta_y}{D^2} & -\frac{\Delta_x}{D^2} & -\frac{\Delta_y}{D^2} & \frac{\Delta_x}{D^2} \\ -\frac{\Delta_x}{D} & -\frac{\Delta_y}{D} & \frac{\Delta_x}{D} & \frac{\Delta_y}{D} \end{bmatrix} \tag{9}$$

2.2 Build a Linear System

The completed `create_linear_system` function can be found in the file `nonlinear.py`.

2.3 Solvers

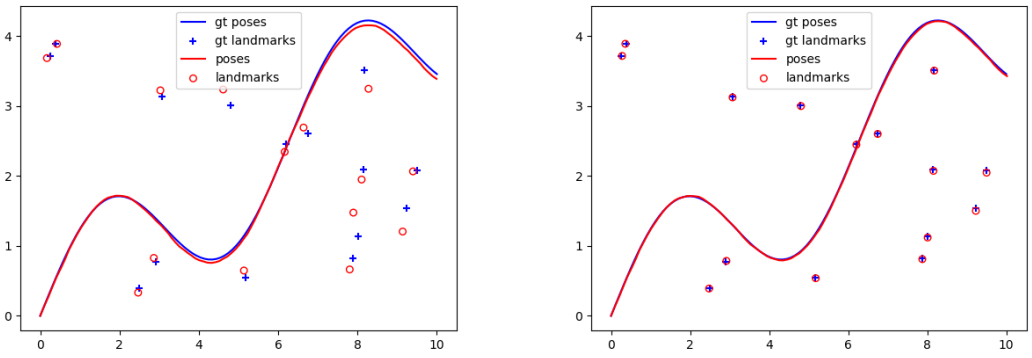
The results of using LU solver with the non-linear data can be seen in figures 12 and 13.



(a) Before Optimization (b) After Optimization

Figure 12: LU Factorization solver

In the case of linear least squares optimization, we solve the minimization problem to directly get the complete state of the robot and environment in a batch. Whereas, in non-linear case, there is no direct solution and we instead get updates which are used to update the current state estimate. This is because in



(a) Before Optimization

(b) After Optimization

Figure 13: LU COLAMD Factorization solver

non-linear case, the functions are non-convex in general and finding the minimum in a closed-form solution is not possible, thus we perform iterations to update the state. Thus, due to this non-linear optimization needs an initial estimate to start with as it affects the final convergence.

Code Submission and Instructions

Minor modifications to the `install_deps.sh` for using tqdm and making sure the sparseqr github repo installation works properly.

```
pip install numpy
pip install cffi
pip install scipy
pip install matplotlib
pip install wheel
pip install tqdm
pip install git+https://github.com/theNded/PySPQR.git
```