1.1

$$r^t = \begin{bmatrix} r^t_x \\ r^t_y \end{bmatrix}, \quad r^{t+1} = \begin{bmatrix} r^{t+1}_x \\ r^{t+1}_y \end{bmatrix}$$

$$h_0(r^t, r^{t+1}) = \begin{bmatrix} r^{t+1}_x - r^t_x \\ r^{t+1}_y - r^t_y \end{bmatrix}$$

$$H_0(r^t, r^{t+1}) = \begin{bmatrix} -1 & 0 & 1 & 0 \\ 0 & -1 & 0 & 1 \end{bmatrix}$$

$\tilde{r}^t_x \quad \tilde{r}^t_y \quad \tilde{r}^{t+1}_x \quad \tilde{r}^{t+1}_y$

$$r^t = \begin{bmatrix} r^t_x \\ r^t_y \end{bmatrix}, \quad l^k = \begin{bmatrix} l^k_x \\ l^k_y \end{bmatrix}$$

$$h_1(r^t, l^k) = \begin{bmatrix} l^k_x - r^t_x \\ l^k_y - r^t_y \end{bmatrix}$$

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

$\widetilde{r^t_x}$ $\widetilde{r^t_y}$ $\widetilde{l^k_x}$ $\widetilde{l^k_y}$

2.2 Nonlinear

2.2

$$H_{\ell} =$$

$$\begin{bmatrix} \frac{(l^k_y - r^t_y)}{(l^k_y - r^t_y)^2 + (l^k_x - r^t_x)^2} & -\frac{(l^k_x - r^t_x)}{(l^k_y - r^t_y)^2 + (l^k_x - r^t_x)^2} & -\frac{(l^k_y - r^t_y)}{(l^k_y - r^t_y)^2 + (l^k_x - r^t_x)^2} & \frac{(l^k_x - r^t_x)}{(l^k_y - r^t_y)^2 + (l^k_x - r^t_x)^2} \\ -\frac{(l^k_x - r^t_x)}{\left((l^k_x - r^t_x)^2 + (l^k_y - r^t_y)^2\right)^{\frac{1}{2}}} & \frac{-(l^k_y - r^t_y)}{\left((l^k_x - r^t_x)^2 + (l^k_y - r^t_y)^2\right)^{\frac{1}{2}}} & \frac{(l^k_x - r^t_x)}{\left((l^k_x - r^t_x)^2 + (l^k_y - r^t_y)^2\right)^{\frac{1}{2}}} & \frac{(l^k_y - r^t_y)}{\left((l^k_x - r^t_x)^2 + (l^k_y - r^t_y)^2\right)^{\frac{1}{2}}} \end{bmatrix}$$

Custom Solver

$$A^T A = A^T B$$

↓

$L, U, \quad R, C$

$$A^T A = R^T (L \; U) \; C^T$$

$$L U x = y$$

→ set $Ux = z$

$$L(z) = y$$

↳ Solve with forward subs

$$Ux = z$$

$L z = y$: forward Subs

$N = \text{len}(A^T b)$ ($z = V n_1$, $y = A^T b$)

for $i = 0 : N-1$

$L_i = L(i,i)$

sum = 0

for $i = 0 : i-1$

sum $\pm$ $L(i,j) \times z(j)$

end

sum + $L_i \times z(i) = y(i)$

$\therefore L_i z(i) = y(i) - \text{Sum}$

$$z(i) = \frac{y(i) - \text{Sum}}{L_i}$$

$\Rightarrow z(i)$ computed

end

after computing z, solve $Ux = z$ with Back Substitution:

```
for i = N-1 : 0
    U_i = U[i, i]
    sum = 0
    for j = i+1 : N-1
        sum += U[i,j] x x[j]
    end
    sum + U_i x[i] = z[i]
    x[i] = (z[i] - sum) / U_i
end
```

HW3 – SLAM Solvers

1) Linear model

a) Default: Time to solve: 0.029s

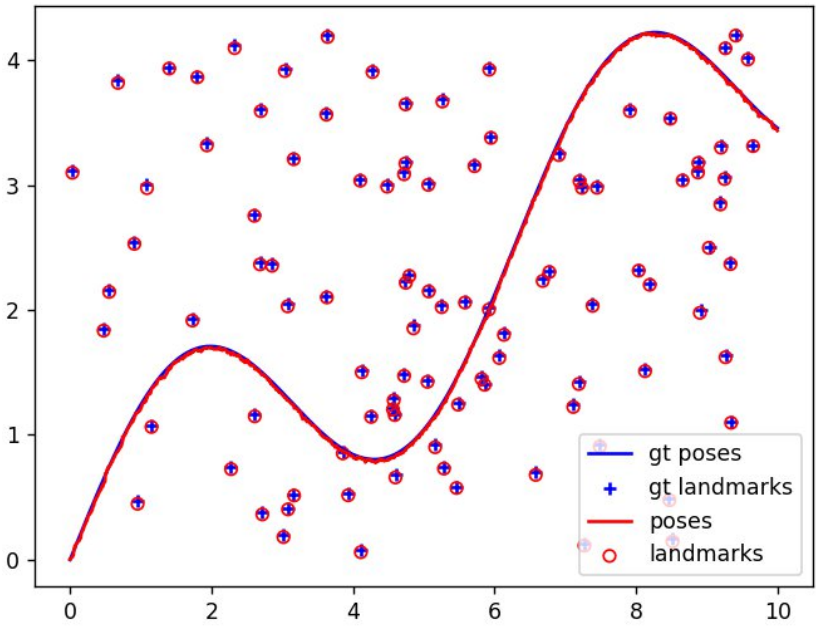

Fig 1: 2d_linear.npz with default solver

b) Pseudo Inverse: Time to solve: 2.07s

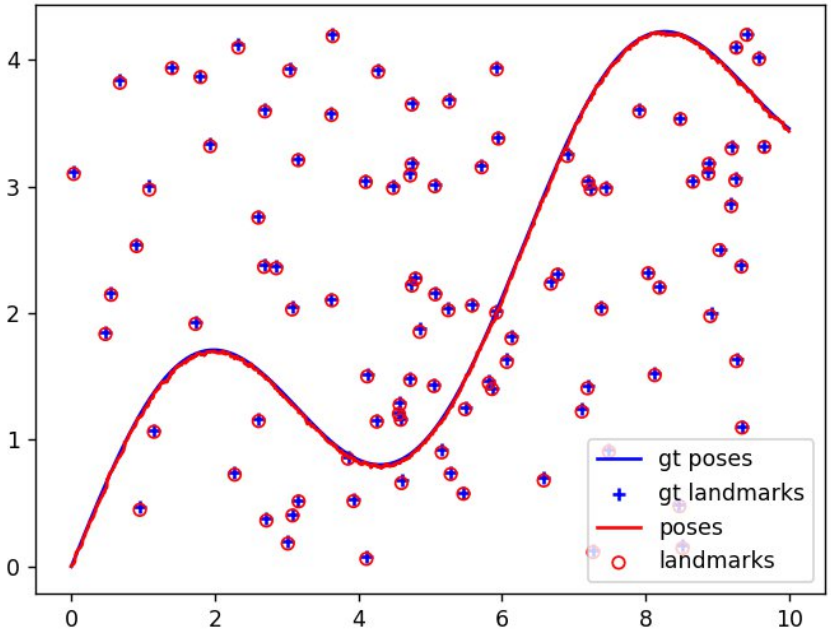

Fig 2: 2d_linear.npz with pinv

c) LU: Time to solve: 0.019s

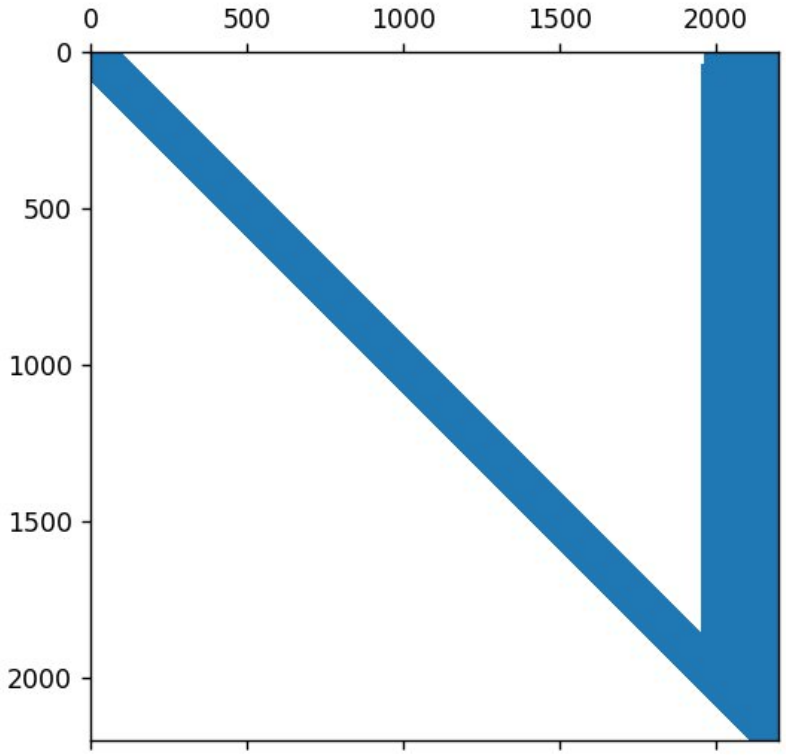

Fig 3. U (upper triangular) Matrix from LU decomposition

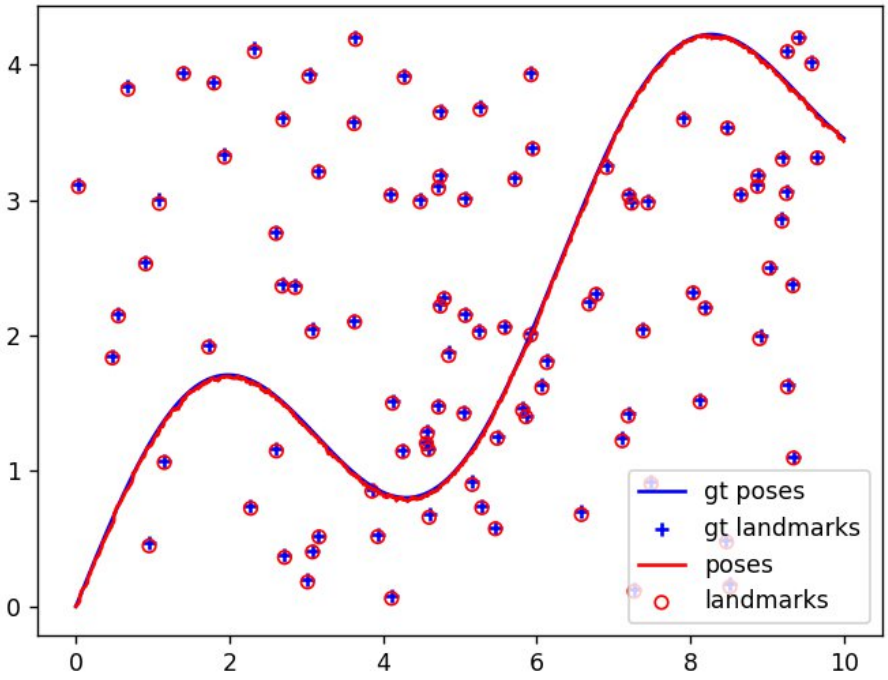

Fig 4: 2d_linear.npz with LU

d) LU-COLAMD: time to solve: 0.032s

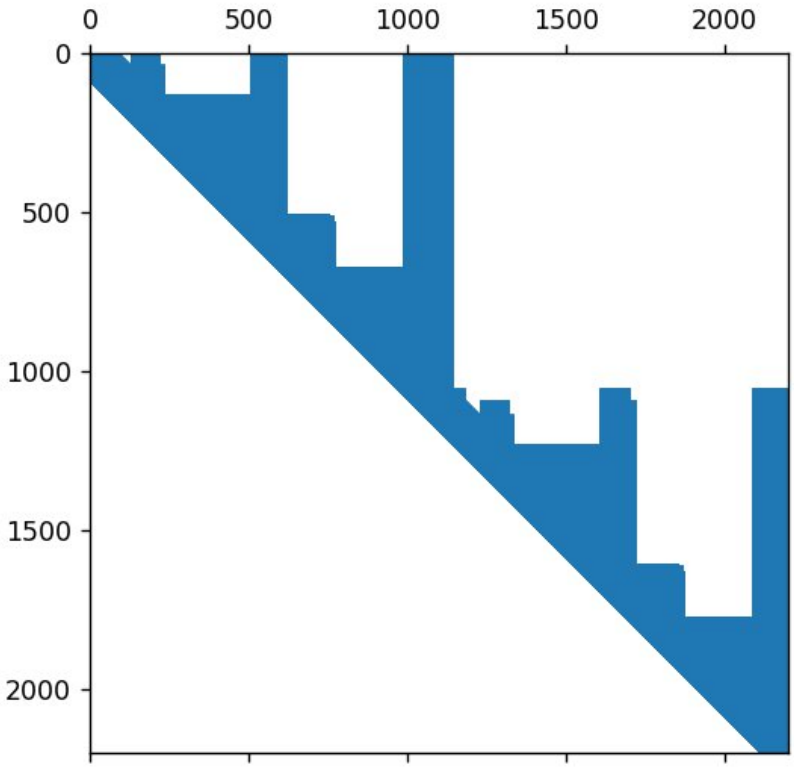

Fig 5: U (Upper triangular matrix)

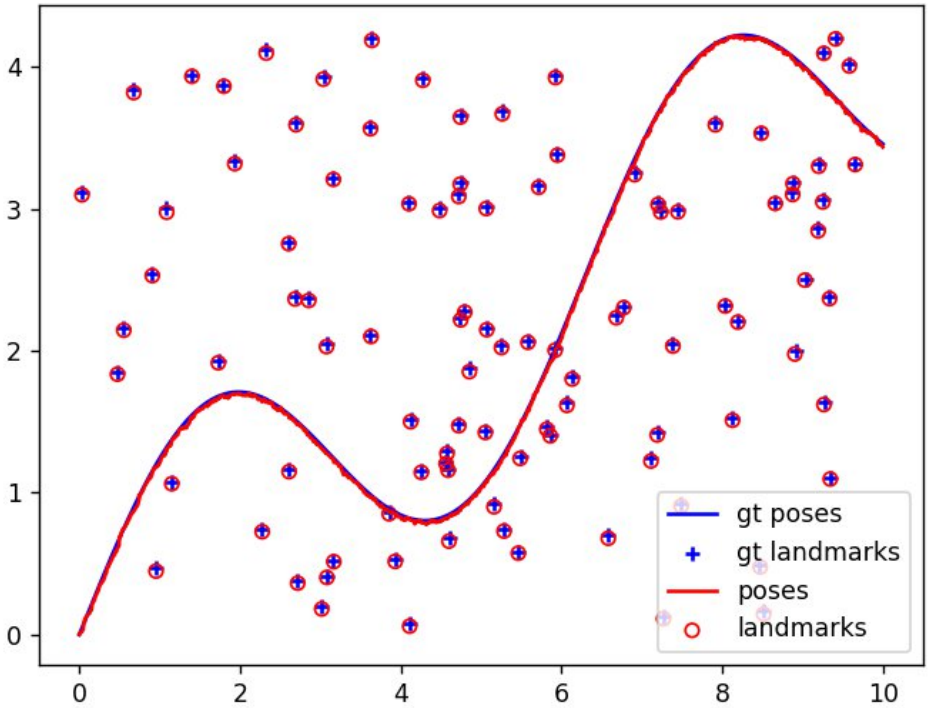

Fig 6: 2d_linear_npz with LU + COLAMD

e) QR: Time to solve: 0.39s

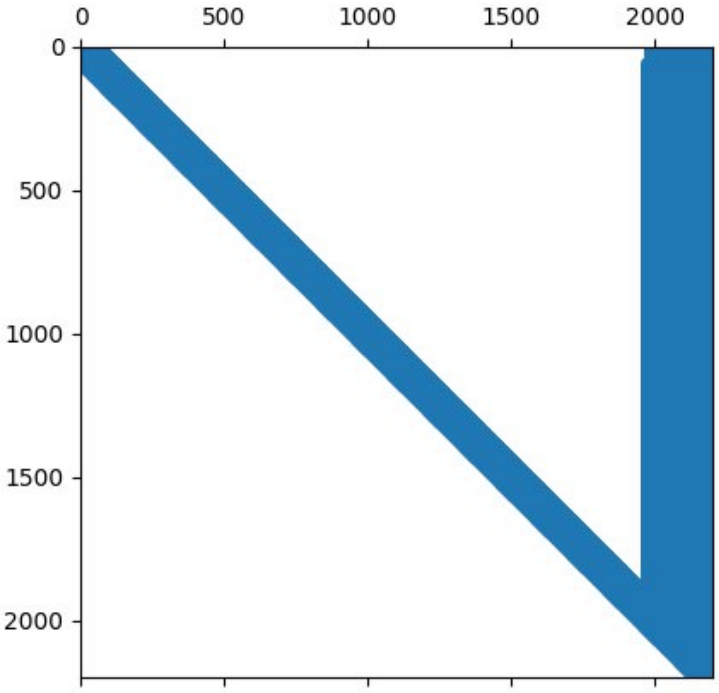

Fig 7. R matrix from QR factorization

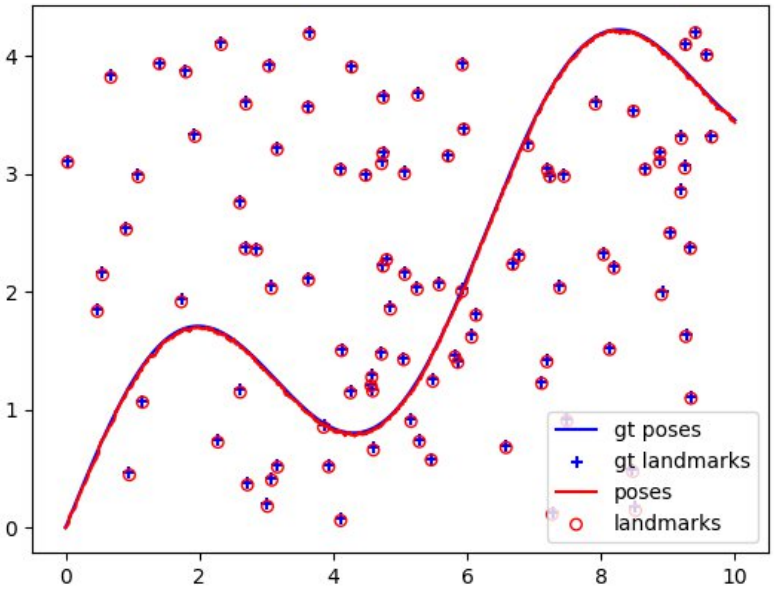

Fig 8. 2d_linear_npz with QR factorization

d) QR with COLAMD: Time to solve: 0.33 s

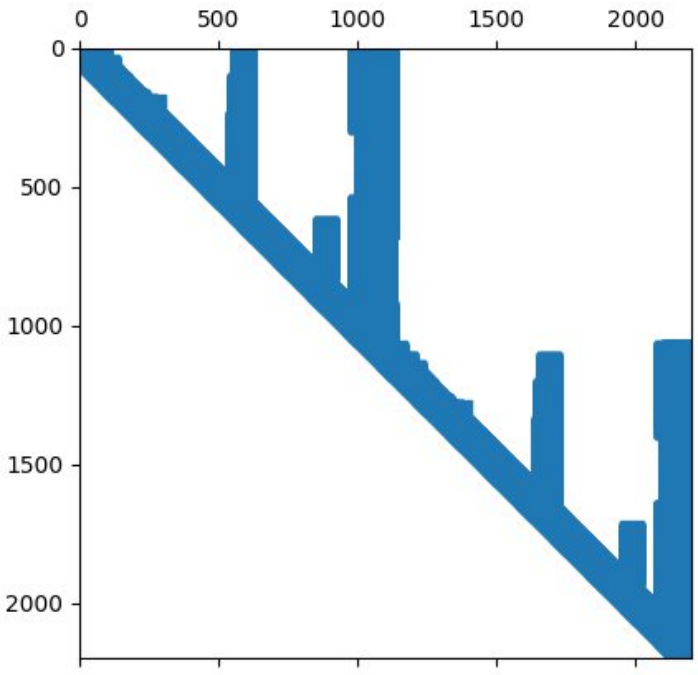

Fig 9. R matrix from QR factorization + COLAMD

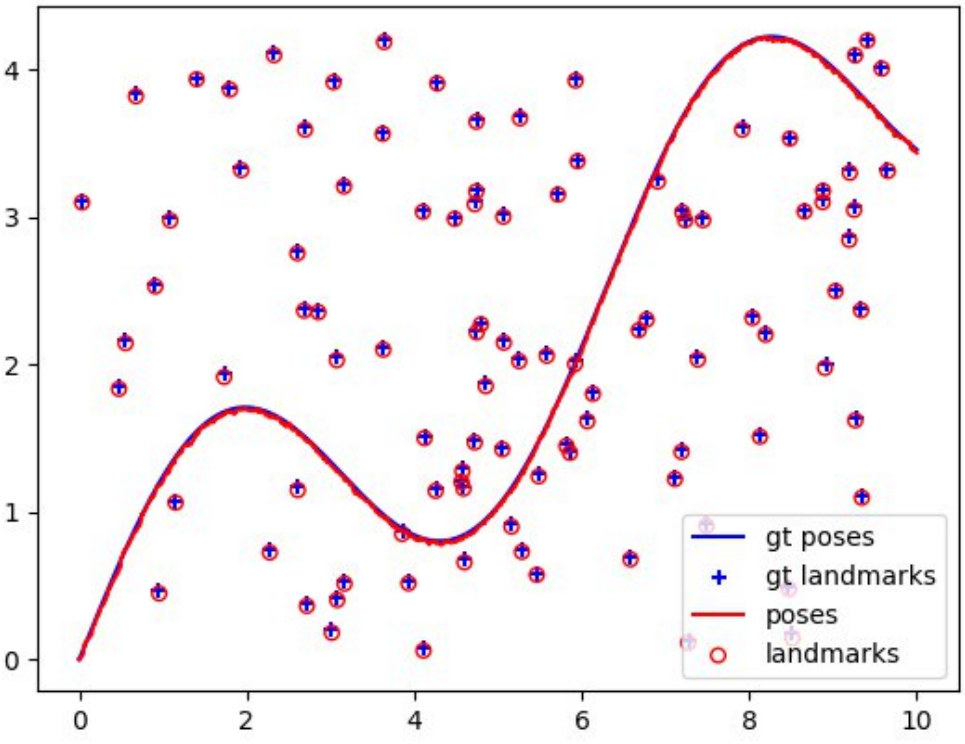

Fig 10. 2d_linear_npz with QR factorization + COLAMD

2) Linear model with loop data

a) Default: Time to solve: 0.016s

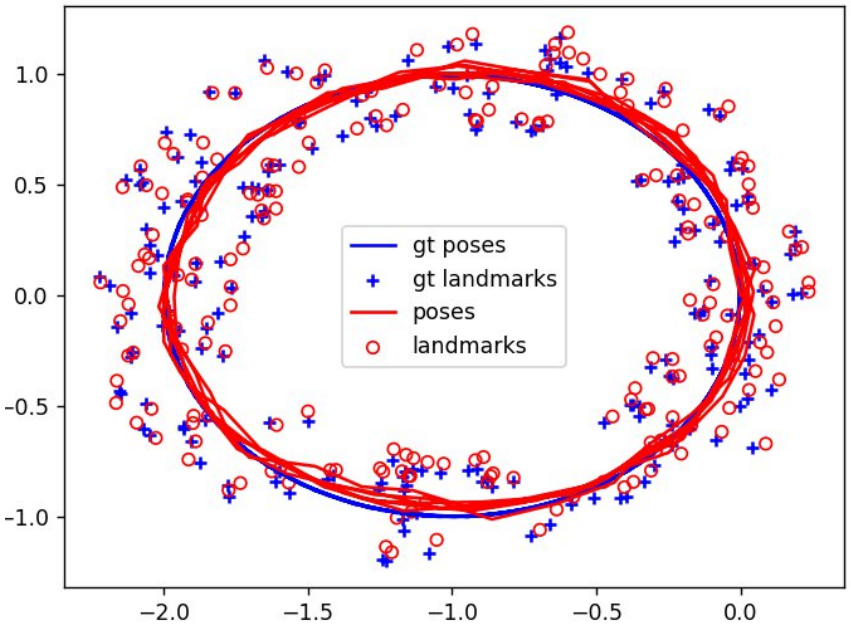

Fig 11: 2d_linear_loop.npz with Default solver

b) Pseudo-inverse: Time to solve: 0.45s

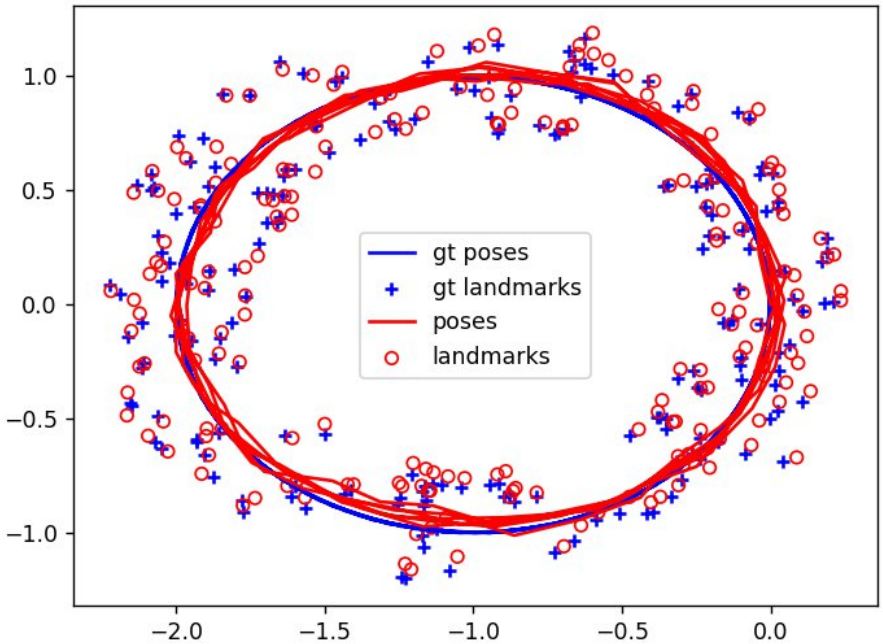

Fig 12: 2d_linear_loop.npz with Pseudo inverse solver

c) LU: Time to solve: 0.04s

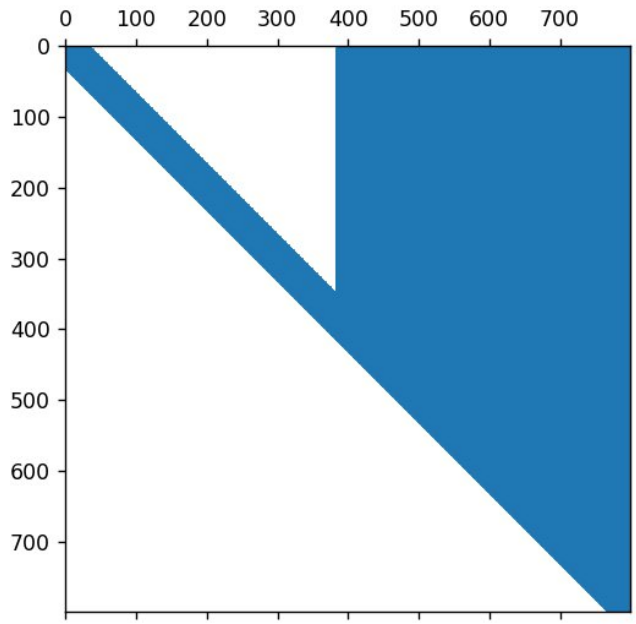

Fig 13: U (upper triangular matrix from LU decomposition)

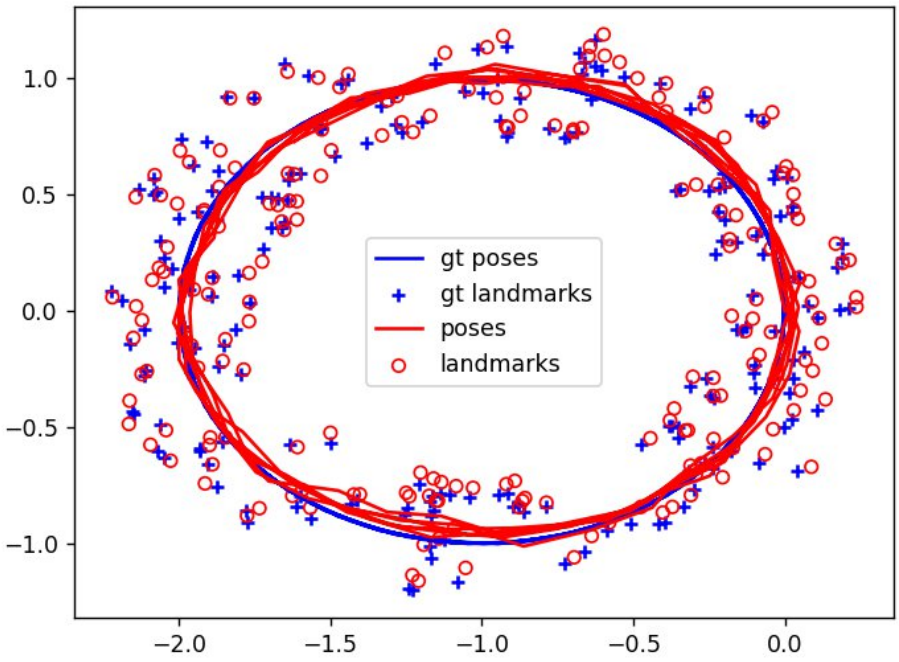

Fig 14: 2d_linear_loop.npz with LU solver

d) LU-COLAMD: Time to solve: 0.009s

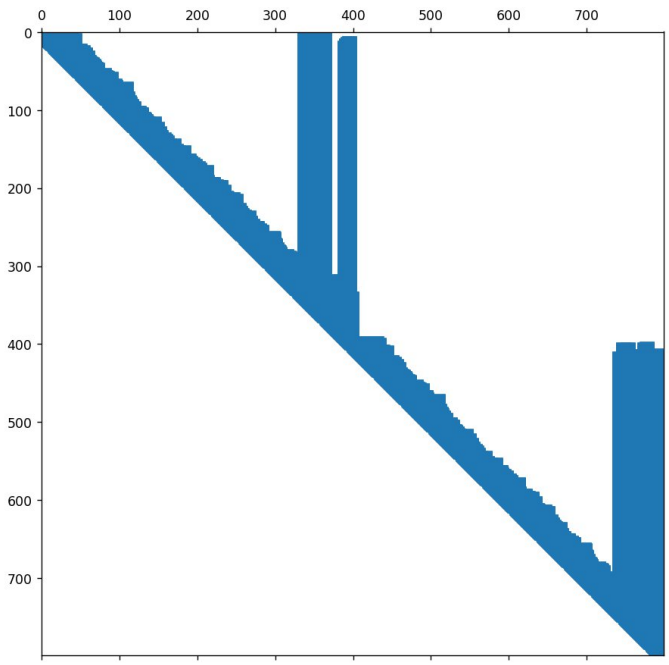

Fig 15: U (upper triangular matrix from LU-COLAMD decomposition)

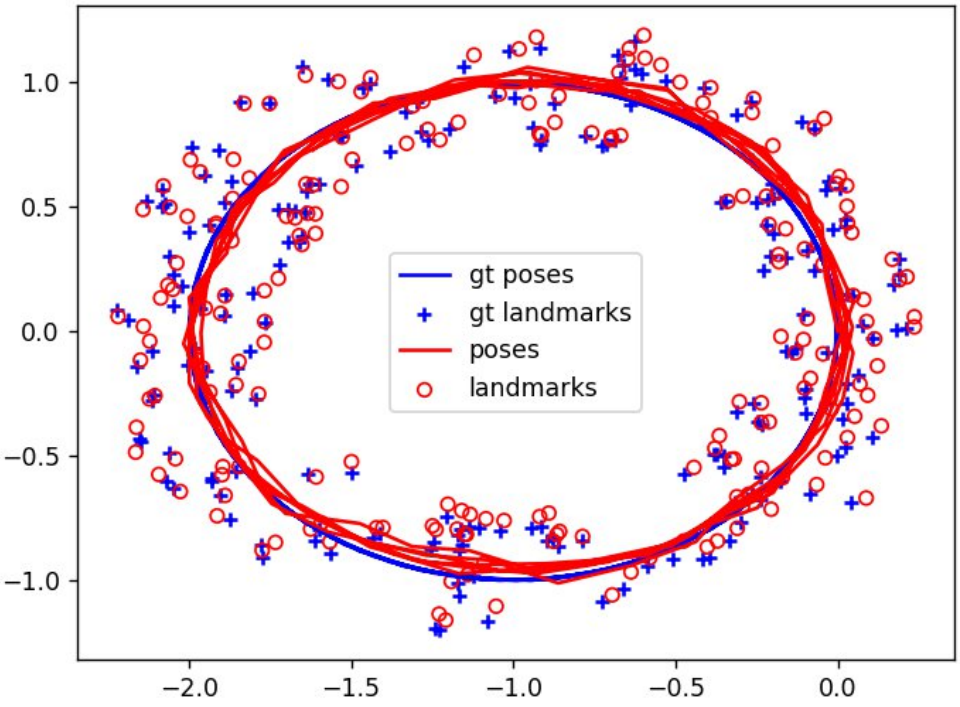

Fig 16: 2d_linear_loop.npz with LU-COLAMD

e) QR: Time to solve: 0.25 s

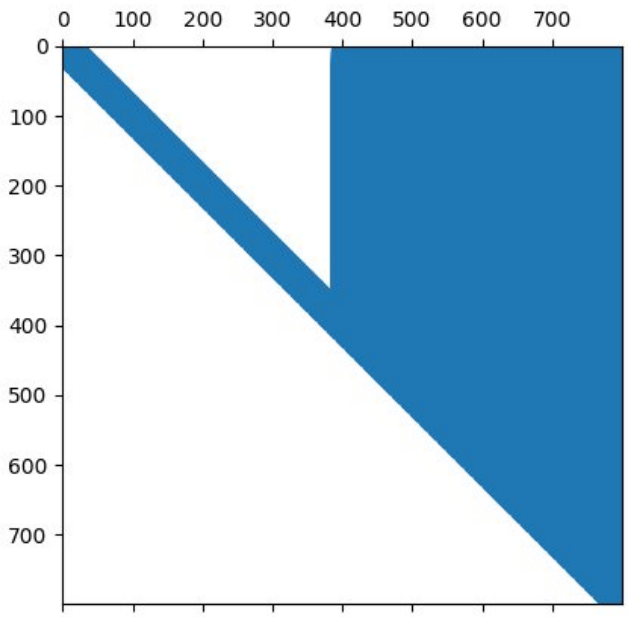

Fig 17: R Matrix from QR

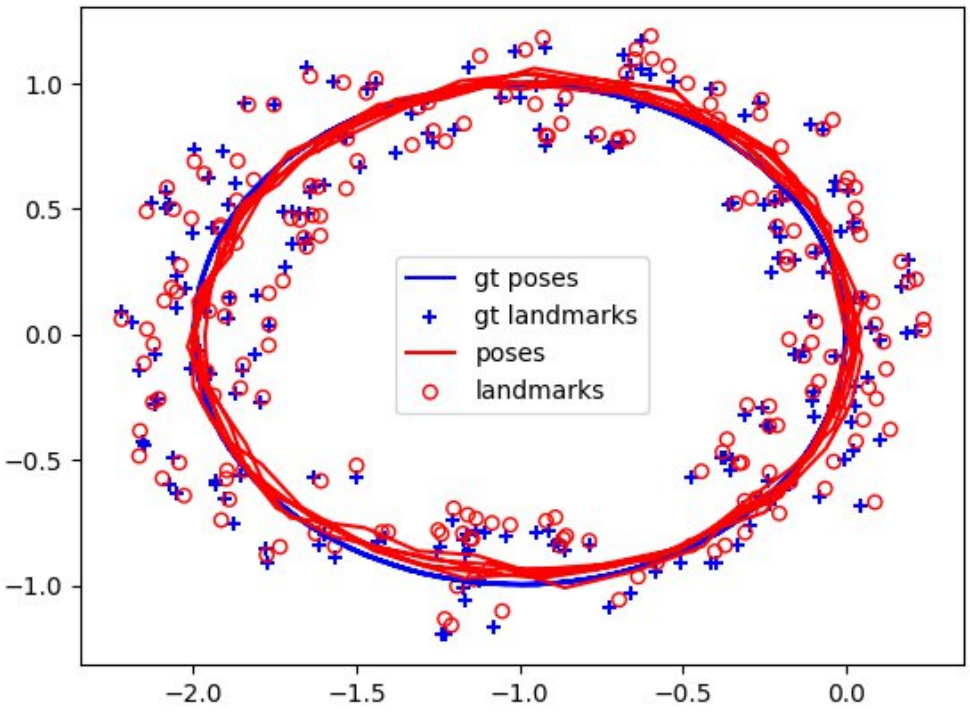

Fig 18. 2d_linear_loop with QR

f) QR + COLAMD: Time to Solve: 0.019s

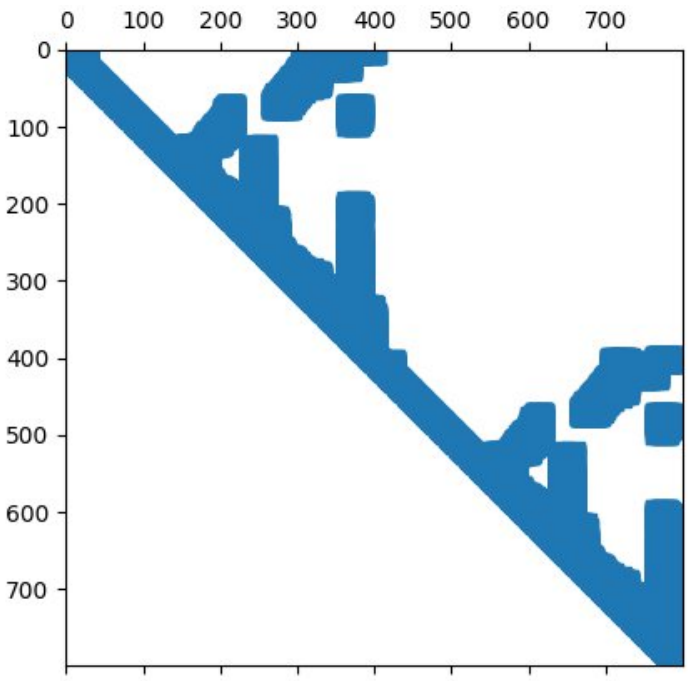

Fig 19: R Matrix from QR + COLAMD

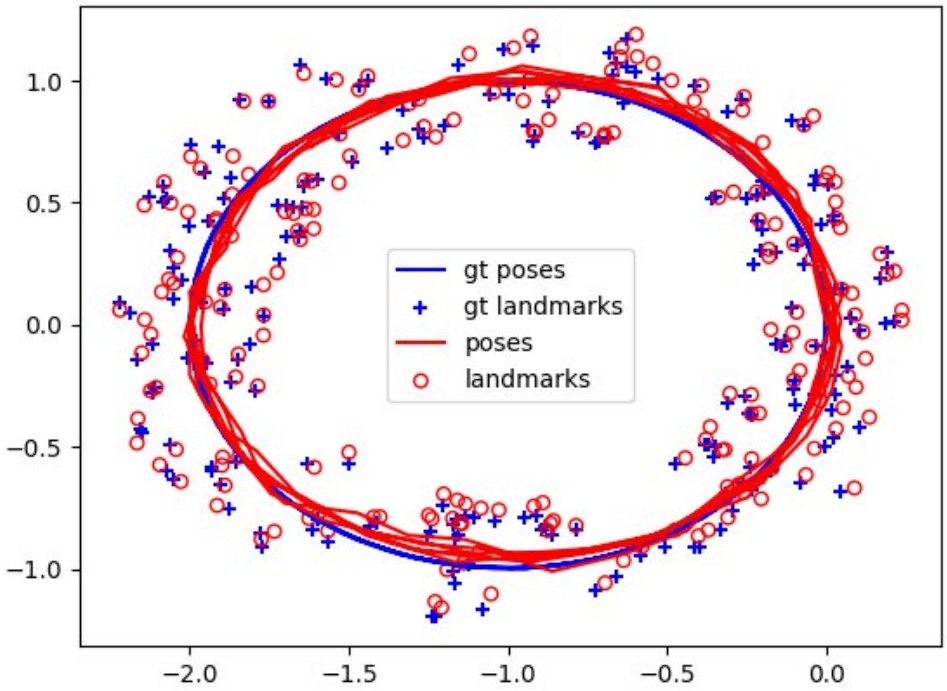

Fig 20. 2d_lin-eaFigure 20. QR + COLAMD

3) Non-Linear model

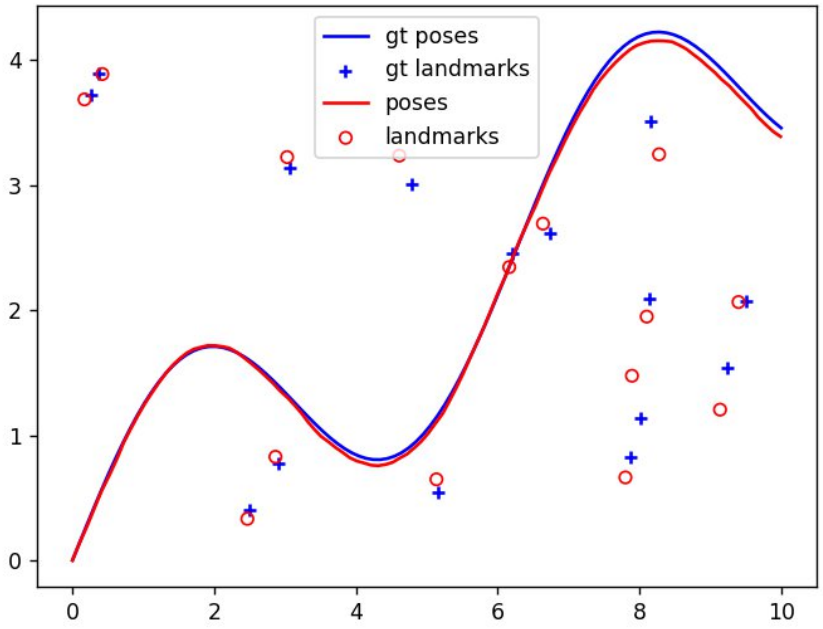

Fig 21: Before optimization

a) Default:

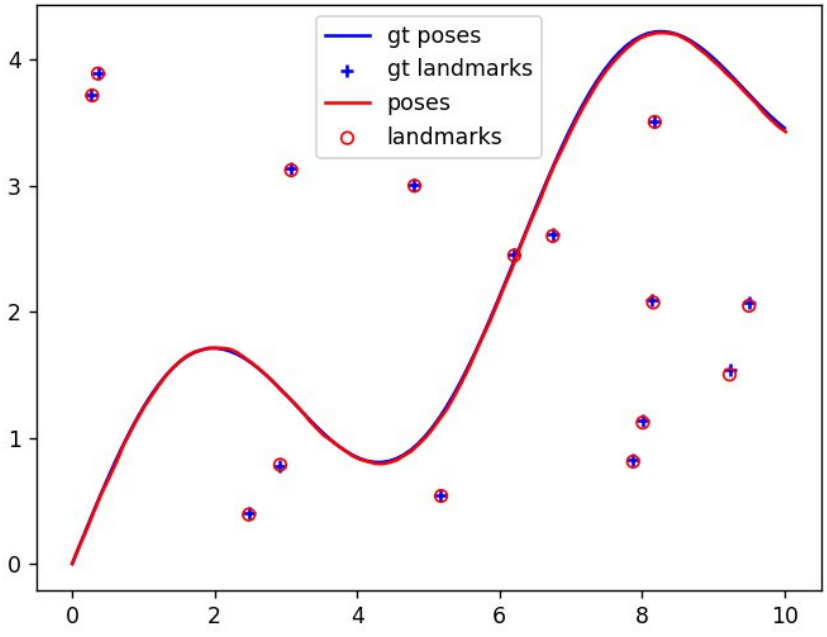

Fig 22: Default solver, After optimization

b) Pseudo Inverse

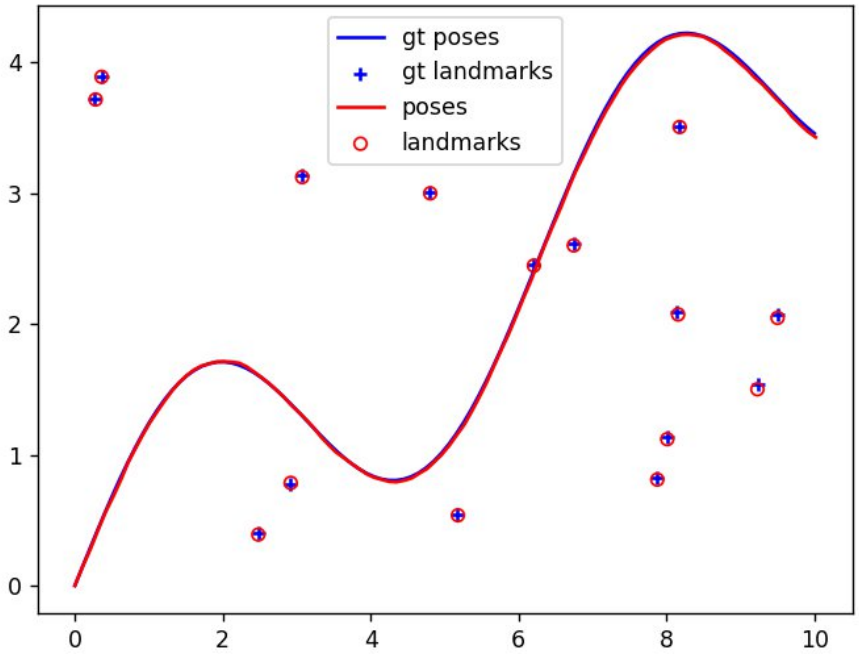

Fig 23: Pseudo Inverse solver, after optimization

c) LU

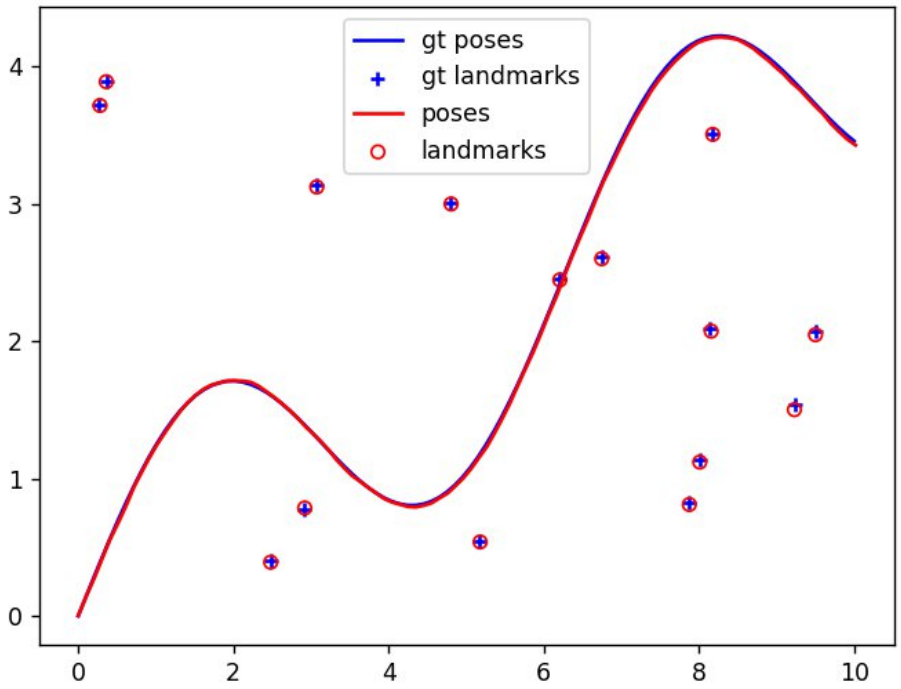

Fig 24: LU Solver, after optimization

d) LU- COLAMD

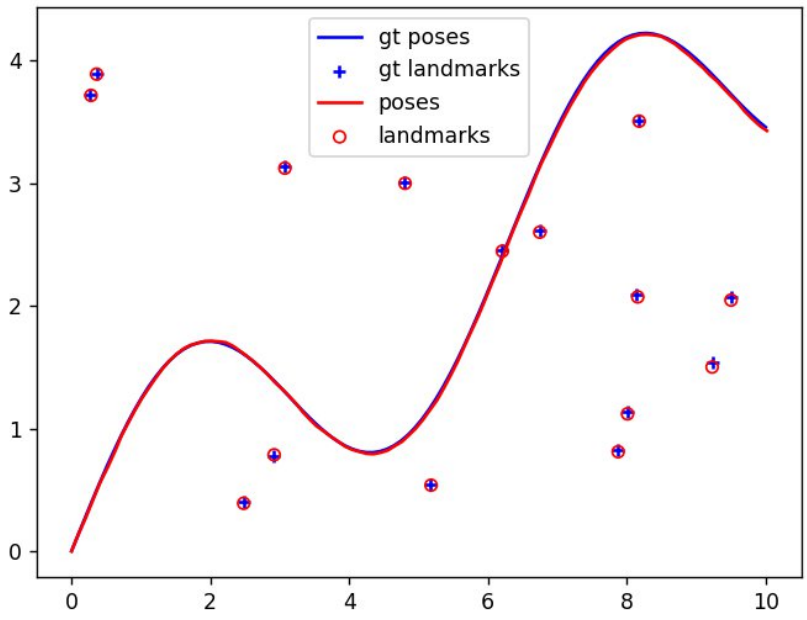

Fig 25:LU-COLAMD After optimization

e) QR

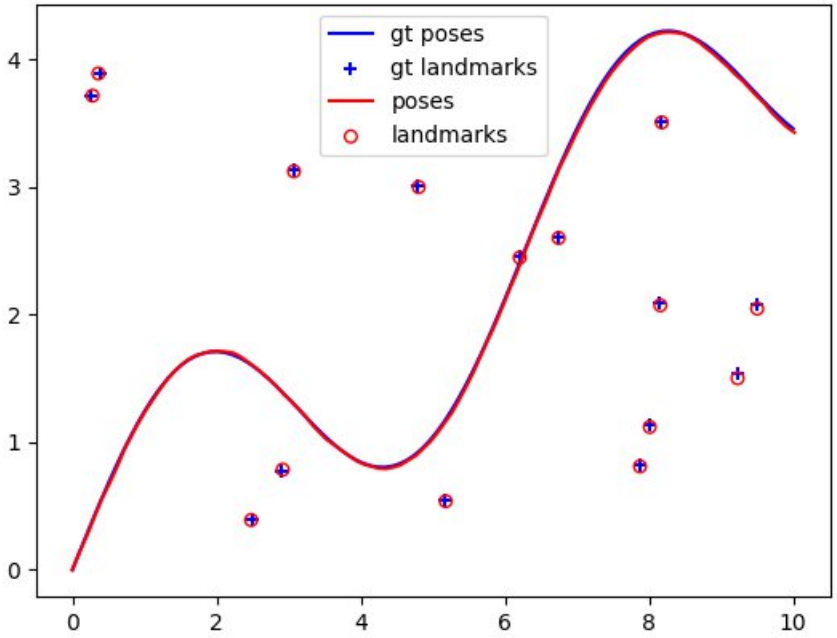

Fig 26: QR after optimization

f) QR COLAMD

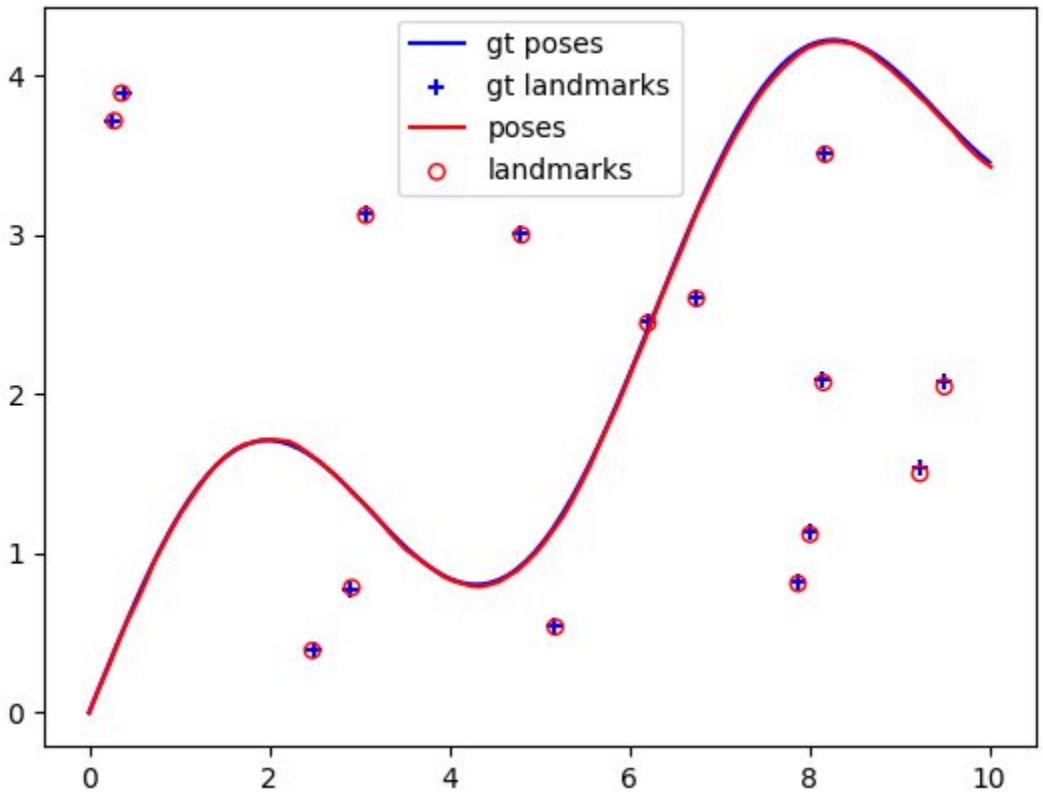

Fig 27: QR after opt

Fig 27: QR with COLAMD after optimization

Discussion

1.4, part 5

For the linear data, LU solved the optimization in 0.019 s, while LU + COLAMD took slightly longer (0.032s). QR took 0.39s, and QR with COLAMD took 0.33 s. LU without COLAMD was faster than LU + COLAMD because the U matrix seen in Figure 3 was already quite sparse. However, QR with COLAMD was slightly faster than without. This might be because I switched to my Linux dual boot to run QR.

2.3

There was a noticeable difference in the loop data, since the triangular matrices were denser. LU took 0.04 seconds to solve, while LU + COLAMD took 0.009 seconds to solve. QR took 0.25 seconds to solve, while QR + COLAMD took 0.019 seconds to solve. In this case, the triangular matrices were quite dense

(higher correlation between states and landmarks since robot is travelling in loops), so using COLAMD helped improve time efficiency considerably.

In the linear case, we used the observations and odometry data to optimize the robot pose and landmark position in a single step, by solving a least squares problem.
In the non linear case, we are setting up and solving a minimization problem, where we are minimizing the residual / error between observed and estimated robot pose and landmark observation. We are also linearizing our system at the x estimated in the previous iteration. (We initialize the x with a guess). This residual minimization is then ran for 10 iterations.