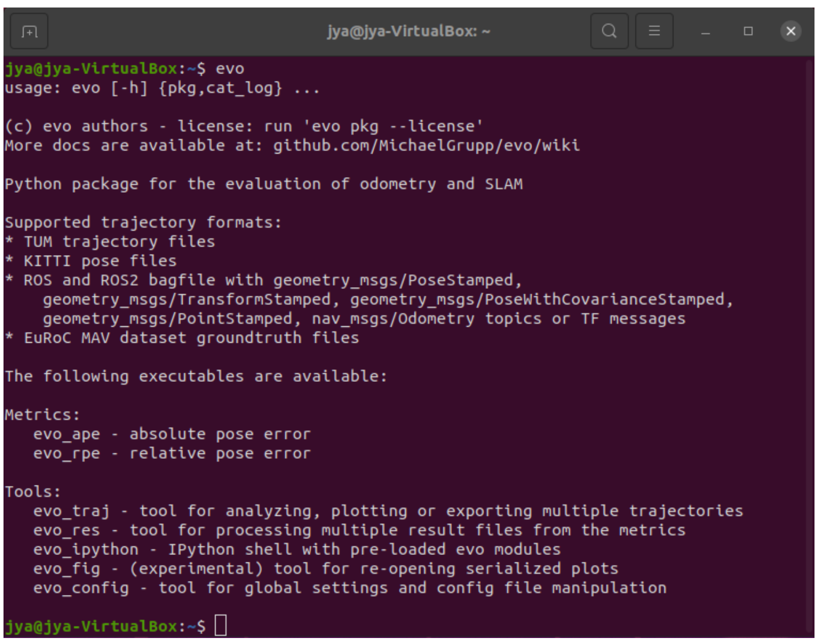
SLAM-homework1

YA JU
juya@shanghaitech.edu.cn
2021533088

March 2024

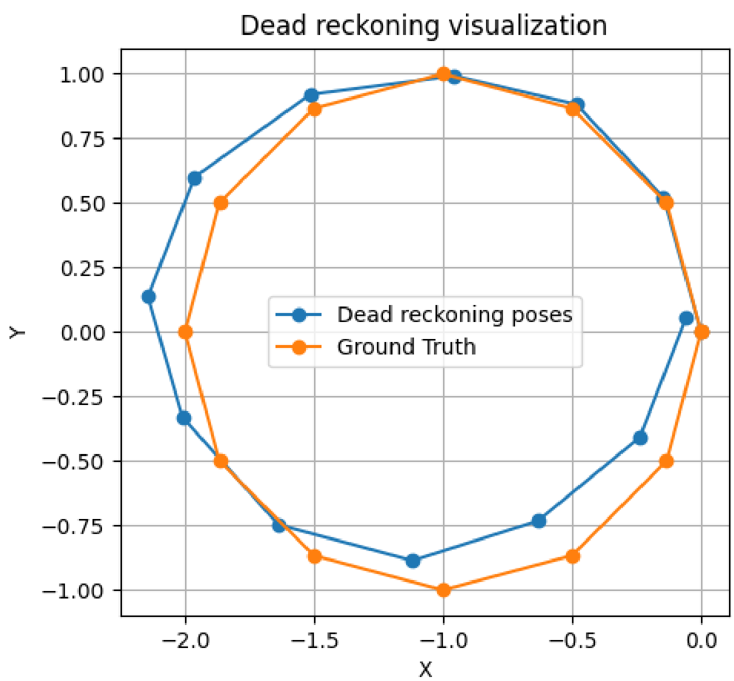
1 Install evo library

Install evo library by cloning from https://github.com/MichaelGrupp/evo.git
As the screenshot below shows, the evo library has already been installed:



2 Dead reckoning using the relative poses

In this question, I use python(with numpy, matplotlib) to implement dead reckoning. Visualization result is shown in the following figure:



I've noticed that spatial drift occurs in the trajectory, especially in the thirteenth pose. It deviates significantly from the actual circular trajectory and exhibits the most significant drift among all poses. This could be caused by the integration of a noisy process.

3 Graph optimization

3.1 Calculation of Jacobian

$$e_{i,j}(x_i, x_j) = t2v(Z_{i,j}^{-1}(T_i^{-1} \cdot T_j)) = t2v(Z_{i,j}^{-1}(v2t(x_i)^{-1} \cdot v2t(x_j))$$

$$v2t(x_i) = \begin{bmatrix} R_i & t_i \\ 0 & 1 \end{bmatrix} \Rightarrow v2t(x_i)^{-1} = \begin{bmatrix} R_i^T & -R_i^T t_i \\ 0 & 1 \end{bmatrix}$$

$$v2t(x_j) = \begin{bmatrix} R_j & t_j \\ 0 & 1 \end{bmatrix}$$

$$v2t(x_i)^{-1} \cdot vt2(x_j) = \begin{bmatrix} R_i^T & -R_i^T t_i \\ 0 & 1 \end{bmatrix} \begin{bmatrix} R_j & t_j \\ 0 & 1 \end{bmatrix} = \begin{bmatrix} R_i^T R_j & R_i^T(t_j - t_i) \\ 0 & 1 \end{bmatrix}$$

$$t2v((v2t(x_i)^{-1} \cdot vt2(x_j)) = \begin{bmatrix} R_i^T(t_j - t_i) \\ \theta_j - \theta_i \end{bmatrix}$$

$t2v(Z_{i,j}^{-1}(v2t(x_i)^{-1} \cdot vt2(x_j))$ can be obtained by by adding transformation $Z_{i,j}^{-1}$

$$\Rightarrow e_{i,j} = \begin{bmatrix} R_{i,j}^T(R_i^T(t_j - t_i) - t_{i,j}) \\ \theta_j - \theta_i - \theta_{i,j} \end{bmatrix}$$

we can then write out the jacobian matrix with respect to x_i and x_j:

$$\frac{\partial e_{i,j}}{\partial x_i} = \begin{bmatrix} -R_{i,j}^T R_i^T & R_{i,j}^T \frac{\partial R_i^T}{\partial \theta}(t_j - t_i) \\ 0 & -1 \end{bmatrix}$$

$$\frac{\partial e_{i,j}}{\partial x_j} = \begin{bmatrix} R_{i,j}^T R_i^T & 0 \\ 0 & 1 \end{bmatrix}$$

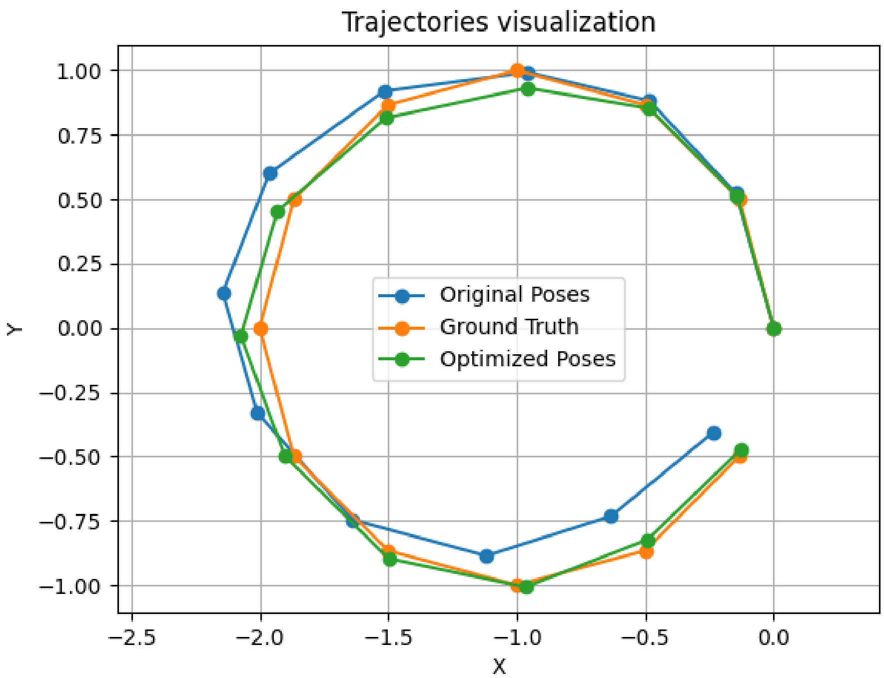
3.2 Overall Algorithm

while $\neg$converged **do**
  $\mathbf{b} \leftarrow \mathbf{0}$ $\mathbf{H} \leftarrow \mathbf{0}$
  for all $\langle \mathbf{e}_{ij}, \mathbf{\Omega}_{ij} \rangle \in \mathcal{C}$ **do**
    // Compute the Jacobians $\mathbf{A}_{ij}$ and $\mathbf{B}_{ij}$ of the error function
    $\mathbf{A}_{ij} \leftarrow \left.\frac{\partial \mathbf{e}_{ij}(\mathbf{x})}{\partial \mathbf{x}_i}\right|_{\mathbf{x}=\breve{\mathbf{x}}}$ $\mathbf{B}_{ij} \leftarrow \left.\frac{\partial \mathbf{e}_{ij}(\mathbf{x})}{\partial \mathbf{x}_j}\right|_{\mathbf{x}=\breve{\mathbf{x}}}$.
    // compute the contribution of this constraint to the linear system
    $\mathbf{H}_{[ii]} \mathrel{+}= \mathbf{A}_{ij}^T \mathbf{\Omega}_{ij} \mathbf{A}_{ij}$ $\mathbf{H}_{[ij]} \mathrel{+}= \mathbf{A}_{ij}^T \mathbf{\Omega}_{ij} \mathbf{B}_{ij}$
    $\mathbf{H}_{[ji]} \mathrel{+}= \mathbf{B}_{ij}^T \mathbf{\Omega}_{ij} \mathbf{A}_{ij}$ $\mathbf{H}_{[jj]} \mathrel{+}= \mathbf{B}_{ij}^T \mathbf{\Omega}_{ij} \mathbf{B}_{ij}$
    // compute the coefficient vector
    $\mathbf{b}_{[i]} \mathrel{+}= \mathbf{A}_{ij}^T \mathbf{\Omega}_{ij} \mathbf{e}_{ij}$ $\mathbf{b}_{[j]} \mathrel{+}= \mathbf{B}_{ij}^T \mathbf{\Omega}_{ij} \mathbf{e}_{ij}$
  end for
  // keep the first node fixed
  $\mathbf{H}_{[11]} \mathrel{+}= \mathbf{I}$
  // solve the linear system using sparse Cholesky factorization
  $\mathbf{\Delta x} \leftarrow \text{solve}(\mathbf{H}\,\mathbf{\Delta x} = -\mathbf{b})$
  // update the parameters
  $\breve{\mathbf{x}} \mathrel{+}= \mathbf{\Delta x}$
end while

In each iteration, calculate the $e_{i,j}$ vector and the Jacobian matrices A_i and B_i for all edges, construct a linear system to solve for Δx, and update x

3.3 Optimization result(Accuracy)

As depicted in the figure below, the implemented graph optimization algorithm has corrected the drifted original poses. However, the optimized poses still exhibit differences from the ground truth due to the noise present in the relative poses used for reference.



Accuracy test using evo(comparision to gt):

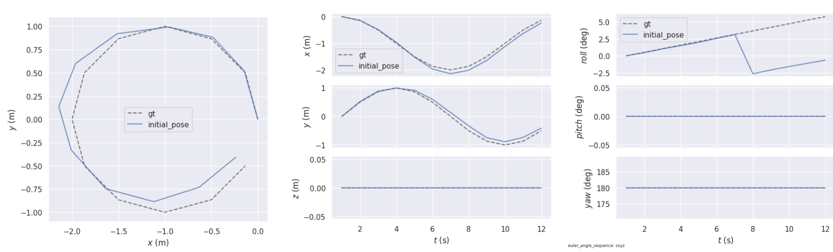


Figure 1: Original Poses compared with GT

max 0.221481
mean 0.114939
median 0.138625
min 0.000000
rmse 0.138007
sse 0.228553
std 0.076387

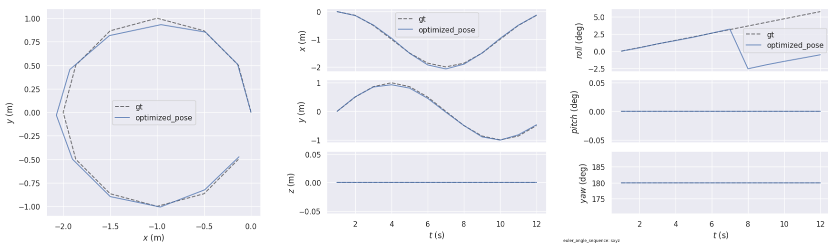


Figure 2: Optimized Poses compared with GT

max 0.080027
mean 0.041242
median 0.035334
min 0.000000
rmse 0.048337
sse 0.028038
std 0.025211

3.4 Optimization result(Time efficiency)

In my graph optimization program, I utilized the sparsity of the Jacobian matrix. By manually deriving the sparse Jacobian matrix, I obtained a sparse H matrix, which I then solved using sparse Cholesky factorization. Below, I will compare the time efficiency of using sparse Cholesky factorization with other methods.

Method Name	Time(s)
Sparse Cholesky Factorization	0.00119927
LU	0.00393712
LDLT	0.00327482

Table 1: Time efficiency of different methods

Because a sparse H matrix is built, we can use sparse Cholesky factorization to accelerate the solving process.

4 g2o for graph optimization

4.1 g2o method

Running g2o with the first point fixed, the optimization process converges in 5 iterations and produce the same accuracy as the optimization in q3.
max 0.080027
mean 0.041242
median 0.035334

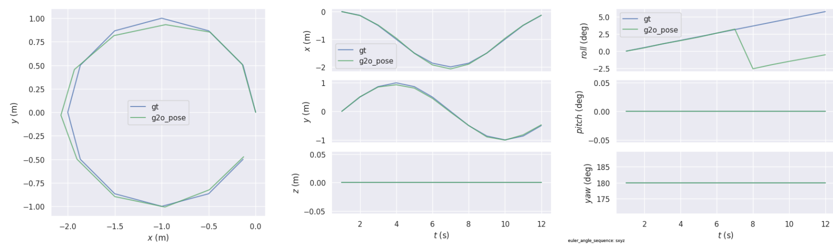


Figure 3: g2o Poses compared with GT

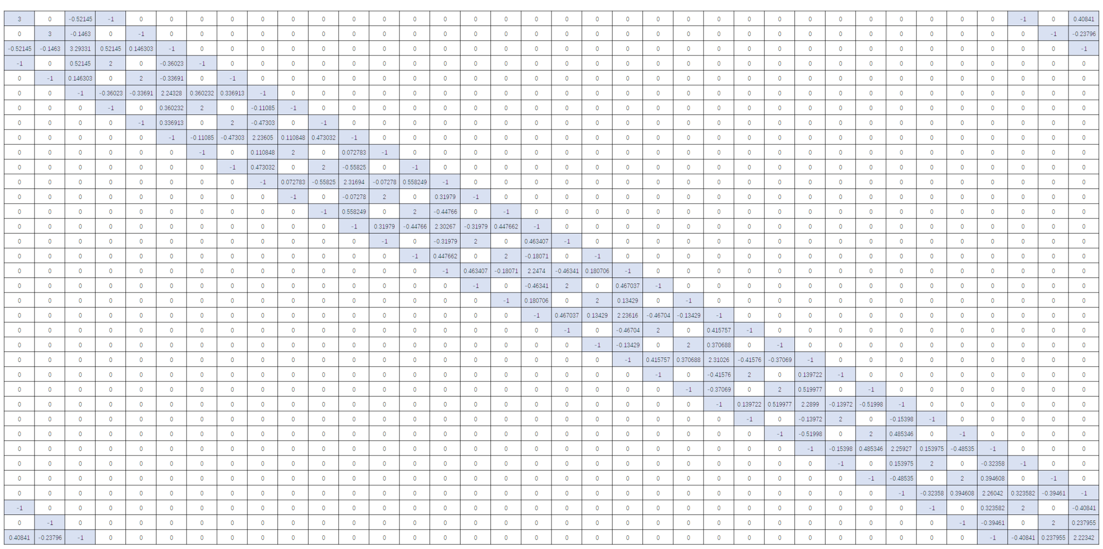
min 0.000000
rmse 0.048337
sse 0.028038
std 0.025211

4.2 comparison with q3

The optimized trajectory is almost exactly the same as q3 method.
running time of g2o is 0.0525685 seconds, meaning that its much slower than q3 method(0.00119927 seconds).

5 Visualization of information matrix

The matrix should look like:



3	0	-0.52145	-1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	-1	0	0.40841
0	3	-0.1463	0	-1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	-1	-0.23796
-0.52145	-0.1463	3.29331	0.52145	0.146303	-1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	-1
-1	0	0.52145	2	0	-0.36023	-1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	-1	0.146303	0	2	-0.33691	0	-1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	-1	-0.36023	-0.33691	2.24328	0.360232	0.336913	-1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	-1	0	0.360232	2	0	-0.11085	-1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	-1	0.336913	0	2	-0.47303	0	-1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	-1	-0.11085	-0.47303	2.23605	0.110848	0.473032	-1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	-1	0	0.110848	2	0	0.072783	-1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	-1	0.473032	0	2	-0.55825	0	-1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	-1	0.072783	-0.55825	2.31694	-0.07278	0.558249	-1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	-1	0	-0.07278	2	0	0.31979	-1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	-1	0.558249	0	2	-0.44766	0	-1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	-1	0.31979	-0.44766	2.30267	-0.31979	0.447662	-1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	-1	0	-0.31979	2	0	0.463407	-1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	-1	0.447662	0	2	-0.18071	0	-1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	-1	0.463407	-0.18071	2.2474	-0.46341	0.180706	-1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	-1	0	-0.46341	2	0	0.467037	-1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	-1	0.180706	0	2	0.13429	0	-1	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	-1	0.467037	0.13429	2.23616	-0.46704	-0.13429	-1	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	-1	0	-0.46704	2	0	0.415757	-1	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	-1	-0.13429	0	2	0.370688	0	-1	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	-1	0.415757	0.370688	2.31026	-0.41576	-0.37069	-1	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	-1	0	-0.41576	2	0	0.139722	-1	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	-1	-0.37069	0	2	0.519977	0	-1	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	-1	0.139722	0.519977	2.2899	-0.13972	-0.51998	-1	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	-1	0	-0.13972	2	0	-0.15398	-1	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	-1	-0.51998	0	2	0.485346	0	-1	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	-1	-0.15398	0.485346	2.25927	0.153975	-0.48535	-1	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	-1	0	0.153975	2	0	-0.32358	-1	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	-1	-0.48535	0	2	0.394608	0	-1	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	-1	-0.32358	0.394608	2.26042	0.323582	-0.39461	-1
-1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	-1	0	0.323582	2	0	-0.40841
0	-1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	-1	-0.39461	0	2	0.237955
0.40841	-0.23796	-1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	-1	-0.40841	0.237955	2.22342

I have painted non-zero elements; essentially, the matrix exhibits a diagonal band, along with two blocks situated in the top-right and bottom-left corners.

The diagonal band comes from sequential connections between nodes i and i+1 (where i ranges from 1 to 11). Additionally, there's an extra loop closure edge between node 11 and node 1, resulting in non-empty blocks H[1][11] and H[11][1].