16-833: Robot Localization and Mapping, Spring 2019 Homework 3 - Linear and Nonlinear SLAM Solvers

Lin, Zhaozhi

1. 2D Linear SLAM

In this problem you will implement your own 2D linear SLAM solver in the `linear/folder`. Everything you need to know to complete this problem was covered in class, so refer to your notes and lecture slides for guidance.

We will be using the least squares formulation of the SLAM problem, which was presented in class:

$$\begin{aligned} x^* &= \underset{x}{\operatorname{argmin}} \sum_{\vdots} \|h_i(x) - z_i\|_{\Sigma_i}^2 \\ &\approx \underset{x}{\operatorname{argmin}} \|\boldsymbol{A}\boldsymbol{x} - \boldsymbol{b}\|^2 \end{aligned} \tag{1}$$

where z_i is the i-th measurement, $h_i(x)$ is the corresponding prediction function, and $\|\boldsymbol{\alpha}\|_{\Sigma}^2$ denotes the squared Mahalanobis distance: $\boldsymbol{\alpha}^{\mathrm{T}}\boldsymbol{\Sigma}^{-1}\boldsymbol{\alpha}$.

In this problem, the state vector is comprised of the trajectory of robot positions and the observed landmark positions. Both positions are simply (x, y) coordinates. There are two types of measurements: odometry and landmark measurements. Odometry measurements give a relative (Δx, Δy) displacement from the previous position to the next position (in global frame). Landmark measurements also give a relative displacement (Δx, Δy) from the robot position to the landmark (also in global frame).

(a) Write out the measurement functions and their Jacobians (for both the odometry measurements and landmark measurements). For simplicity, please parameterize these functions on only the relevant state variables rather than the entire state vector. Use h_o and h_m to denote the odometry and measurement functions, respectively, and their Jacobians are H_o and H_m. Use $\boldsymbol{r} = [r_x \quad r_y]^{\mathrm{T}}$ and $\boldsymbol{l} = [l_x \quad l_y]^{\mathrm{T}}$ to denote robot and landmark positions, respectively. Use $\boldsymbol{r^{t-1}}$ to denote the previous robot pose and $\boldsymbol{r^t}$ the next robot pose in an odometry measurement. (5 points)

Solution:

Write out the odometry and measurement functions

$$\boldsymbol{h}_o = \boldsymbol{r^t} - \boldsymbol{r^{t-1}} = \begin{bmatrix} r_x^t - r_x^{t-1} \\ r_y^t - r_y^{t-1} \end{bmatrix} \tag{1-1}$$

$$\boldsymbol{h}_m = \boldsymbol{l} - \boldsymbol{r} = \begin{bmatrix} l_x - r_x \\ l_y - r_y \end{bmatrix} \tag{1-2}$$

And their corresponding Jacobian matrices

$$\boldsymbol{H}_o = \begin{bmatrix} \frac{\partial \boldsymbol{h}_o}{\partial \boldsymbol{r^{t-1}}} & \frac{\partial \boldsymbol{h}_o}{\partial \boldsymbol{r^t}} \end{bmatrix} = \begin{bmatrix} -1 & 0 & 1 & 0 \\ 0 & -1 & 0 & 1 \end{bmatrix} \tag{1-3}$$

$$\boldsymbol{H}_m = \begin{bmatrix} \frac{\partial \boldsymbol{h}_m}{\partial \boldsymbol{r}} & \frac{\partial \boldsymbol{h}_m}{\partial \boldsymbol{l}} \end{bmatrix} = \begin{bmatrix} -1 & 0 & 1 & 0 \\ 0 & -1 & 0 & 1 \end{bmatrix} \tag{1-4}$$

(b) Once you have these functions, you can complete the file `create_Ab_linear.m` to generate the linear system. Simply choose a variable ordering (for convenience in constructing A and b, not for efficiency) and fill in the correct blocks based on the measurements and Jacobians. Add a prior on the first pose placing it at the origin. If you do not, your system will not be fully constrained (up to an arbitrary global transformation) and will not line up with the ground truth. Finally, complete the function `format_solution()` located in the util/ directory. Simply format the data from your state vector into a trajectory and list of landmarks as specified in the file. The MATLAB function reshape() should be useful here. (5 points)

(c) Now that you have the linear system, experiment with five different ways to solve it. Complete the functions in `solve_pinv.m`, `solve_chol1.m`, `solve_qr1.m`, `solve_chol2.m`, and `solve_qr2.m`. Follow these guidelines for each type of solution:

i) `solve_pinv()` - do not use MATLAB's `pinv()` function - it will not work on sparse matrices. Do not use "\" either. Simply solve the normal equations naively by computing the pseudo-inverse yourself. (5 points)

ii) `solve_chol1()` - use MATLAB's `chol()` function and the provided `forward_sub()` and `back_sub()` functions in the `util/ folder` to solve it. Return the solution as well as the computed R factor. (5 points)

iii) `solve_qr1()` - use MATLAB's `qr()` function. Look at the options listed in the documentation for sparse matrices ("`help qr`" or "`doc qr`"). Since you need to compute Q^T b to solve the system, choose the option that returns such a vector as well as the R factor. Utilize the "economy size" version if it is possible to do so. Why can you or why can you not utilize this version? (10 points)

We can utilize economy size in this case, because the $\boldsymbol{A}$ is an m×n matrix where m > n, so the $\boldsymbol{Q}$ and $\boldsymbol{R}$ matrices can be partitioned in the following way:

$$\boldsymbol{QR} = [\boldsymbol{Q_1} \quad \vdots \quad \boldsymbol{Q_2}] \begin{bmatrix} \boldsymbol{R_1} \\ \cdots \\ \boldsymbol{O} \end{bmatrix} = \boldsymbol{Q_1 R_1} + \boldsymbol{Q_2 O} = \boldsymbol{Q_1 R_1} \tag{1-5}$$

which means we can discard $\boldsymbol{Q_2}$.

iv) `solve_chol2()` - use the same `chol()` function you used for solve_chol1() but with a fill-in-reducing ordering. (5 points)

v) `solve_qr2()` - use the same `qr()` function you used for `qr1()` but with a fill-in-reducing ordering. (5 points)

You can test a specific solution type by running the function slam_2D_linear() with the corresponding solution name (`pinv`, `chol`, `qr1`, `cho2`, or `qr2`).

(d) Once you are sure each of your decompositions is correct, compare the efficiency of these methods using the following datasets.

i) 2D_linear dataset: Compare these methods to each other by running `slam_2D_linear()` with no input argument. The function should print out the run time for each of these algorithms and the evaluation of the resulting solution. Include the two figures that are generated - the resulting trajectory and the sparsity patterns of each decomposition. List the decompositions in order of efficiency, from most efficient to least efficient. Explain why they are in this order. (10 points)

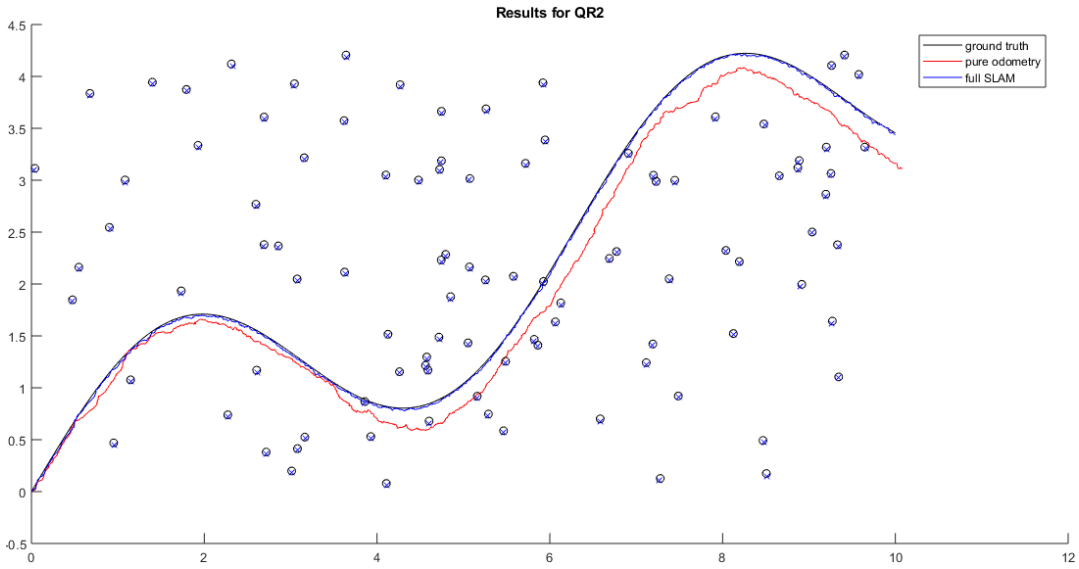

Figure 1 Output Trajectory for 2D_linear Dataset

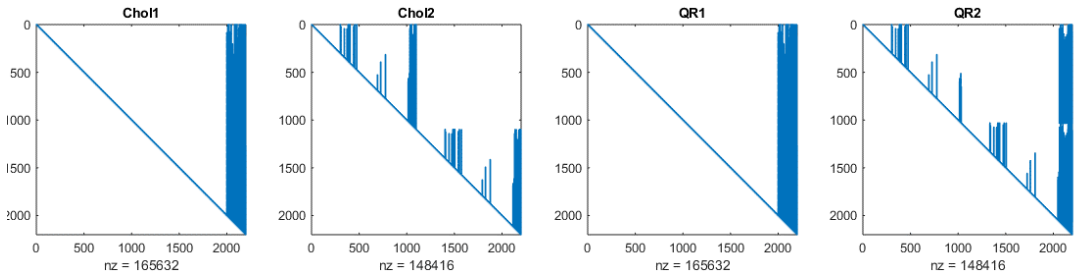

Figure 2 Sparsity Patterns

```
Timing Results
Pinv:  2.649773e+00 sec
Chol1: 3.206756e-01 sec
Chol2: 1.525437e-01 sec
QR1:   3.519269e-01 sec
QR2:   2.151727e-01 sec
```

From most efficient to least efficient: `Chol2`, `QR2`, `Chol1`, `QR1`, `Pinv`.

The R matrices obtained from Cholesky decomposition and QR decomposition have similar sparsity. But when $m>>n$ for QR decomposition the time complexity is $O(2mn^2)$, for Cholesky decomposition it is about $O(mn^2)$ thus QR decomposition is slower than Cholesky decomposition. Obviously when applying fill-in reducing ordering, we could achieve better sparsity and thus the cost is greatly reduced so `QR2` and `chol2` have better performance in runtime than `QR1` and `chol1`. For pseudo-inverse method,

ii) 2D_linear_loop dataset: Change the input data file in `slam_2D_linear()` to the other listed data file, `2D_linear_loop.mat`. Compare your methods on this dataset. Include the two figures that are generated - the resulting trajectory and the sparsity patterns of each decomposition. List the decompositions in order of efficiency, from most efficient to least efficient. Is this order the same as the last dataset? Why or why not? (10 points)

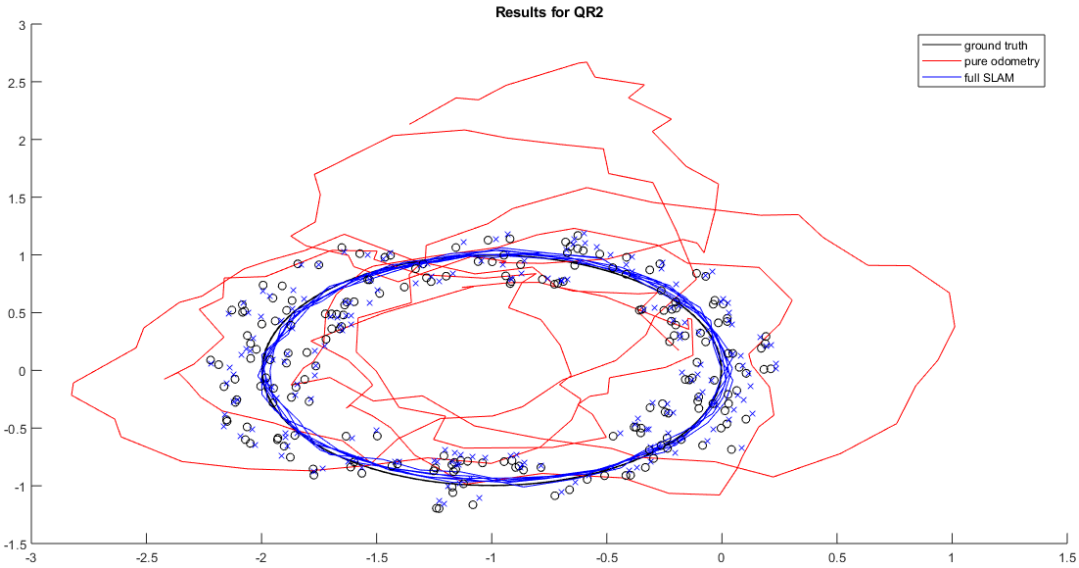

Figure 3 Output Trajectory for 2D_linear_loop Dataset

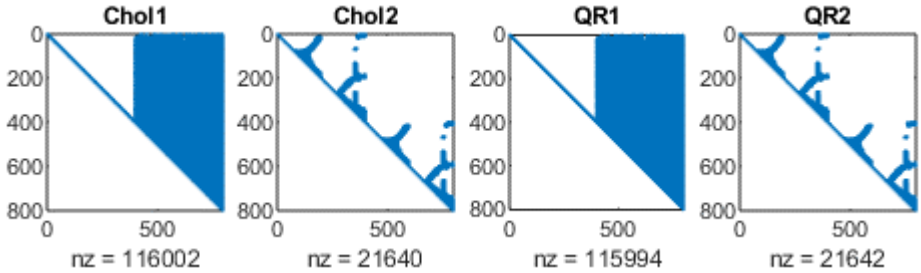

Figure 4 Sparsity Patterns for 2D_linear_loop Dataset

```
Timing Results
Pinv:  8.763299e-02 sec
```

```
Chol1: 1.232674e-01 sec
Chol2: 2.152619e-02 sec
QR1:   1.658966e-01 sec
QR2:   2.076489e-02 sec
```

From most efficient to least efficient: `QR2`, `chol2`, `pinv`, `chol1`, `QR1`.

The order is different from the previou. Here `chol2` and `QR2` still perform better than `chol1` and `QR1` for the same reason mentioned before. But `QR2` performs better than `chol2.` If we look at the *A* matrix in this case and compare it with the one above we will see that this time *m/n* is much smaller. In this case the time complexities of forward substitution and back substitution, which are both $O(n^2)$, matter. And `pinv` is faster than `chol1 and QR1` in this case. The *R* matrices from `chol1` and `QR1` are relatively much denser than the previous ones.

2. 2D Nonlinear SLAM

Now you are going to build off of your linear SLAM algorithm to create a nonlinear SLAM algorithm in the `nonlinear/folder`. The problem set-up is exactly the same as in the linear problem, except we introduce a nonlinear measurement function that returns a bearing angle and range d (in robot's body frame, notice we assume that this robot always perfectly facing the x-direction of the global frame), which together describe the vector from the robot to the landmark:

$$\boldsymbol{h}_m(r_x, r_y, l_x, l_y) = \begin{bmatrix} atan2(l_y - r_y, l_x - r_x) \\ \sqrt{(l_x - r_x)^2 + (l_y - r_y)^2} \end{bmatrix} \tag{2}$$

(a) In your nonlinear algorithm, you'll need to predict measurements based on the current state estimate. Fill in the functions `meas_odom.m` and `meas_landmark.m` with the corresponding measurement functions. The odometry measurement function is the same linear function we used in the linear SLAM algorithm, while the landmark measurement function is the new nonlinear function introduced in Equation 2. (5 points)

(b) Derive the Jacobian of the nonlinear landmark function in your writeup and implement the function `meas_landmark_jacobian.m` to calculate the jacobian at the provided linearization point. (5 points)

$$\boldsymbol{H}_m = \begin{bmatrix} \frac{\partial \boldsymbol{h}_m}{\partial r_x} & \frac{\partial \boldsymbol{h}_m}{\partial r_y} & \frac{\partial \boldsymbol{h}_m}{\partial l_x} & \frac{\partial \boldsymbol{h}_m}{\partial l_y} \end{bmatrix}$$

$$= \begin{bmatrix} \frac{l_y - r_y}{(l_x - r_x)^2 + (l_y - r_y)^2} & -\frac{l_x - r_x}{(l_x - r_x)^2 + (l_y - r_y)^2} & -\frac{l_y - r_y}{(l_x - r_x)^2 + (l_y - r_y)^2} & \frac{l_x - r_x}{(l_x - r_x)^2 + (l_y - r_y)^2} \\ -\frac{l_x - r_x}{\sqrt{(l_x - r_x)^2 + (l_y - r_y)^2}} & -\frac{l_y - r_y}{\sqrt{(l_x - r_x)^2 + (l_y - r_y)^2}} & \frac{l_x - r_x}{\sqrt{(l_x - r_x)^2 + (l_y - r_y)^2}} & \frac{l_y - r_y}{\sqrt{(l_x - r_x)^2 + (l_y - r_y)^2}} \end{bmatrix} \tag{2-1}$$

(c) Implement `create_Ab_nonlinear.m` to generate the linear system A and b at the current linearization point. Remember to include a prior on the first pose to tie it down to the origin. Also implement error_nonlinear:m. This function will be similar to `create_Ab_linear.m` but instead of generating the linear system, it will compute the total sums of squares error evaluated at the current state estimate. This function is necessary for our implementation of the Gauss-Newton algorithm, which we have provided for you in `gauss_newton.m`. (15 points)

Take caution when computing differences between angles in both of these functions. If you take the difference between two angles that are around , but one is positive and another is negative, you may end up with a difference that is close to $\pm 2\pi$, when the real difference should be around 0. To alleviate this, normalize angles to the interval $(-\pi; \pi]$ after taking differences.

(d) Choose your favorite solver from Section 2 (b) and implement it in the file solve_linear_system:m. The function `solve_linear_system()` is applied to solve the linearized system in each Gauss-Newton iteration. Now test your SLAM algorithm by running `slam_2D_nonlinear.m`. MATLAB's debugging tools (setting breakpoints and such) may prove useful if you need to debug your code. You may also want to uncomment some of the fprintf lines in `gauss_newton.m` to get more information about the state of the system. When implemented properly, your solution should follow the ground truth trajectory closely and pass the evaluation at the end of the run. Show the resulting trajectory and report the error of your final solution compared to that of the pure odometry trajectory. (10 points)

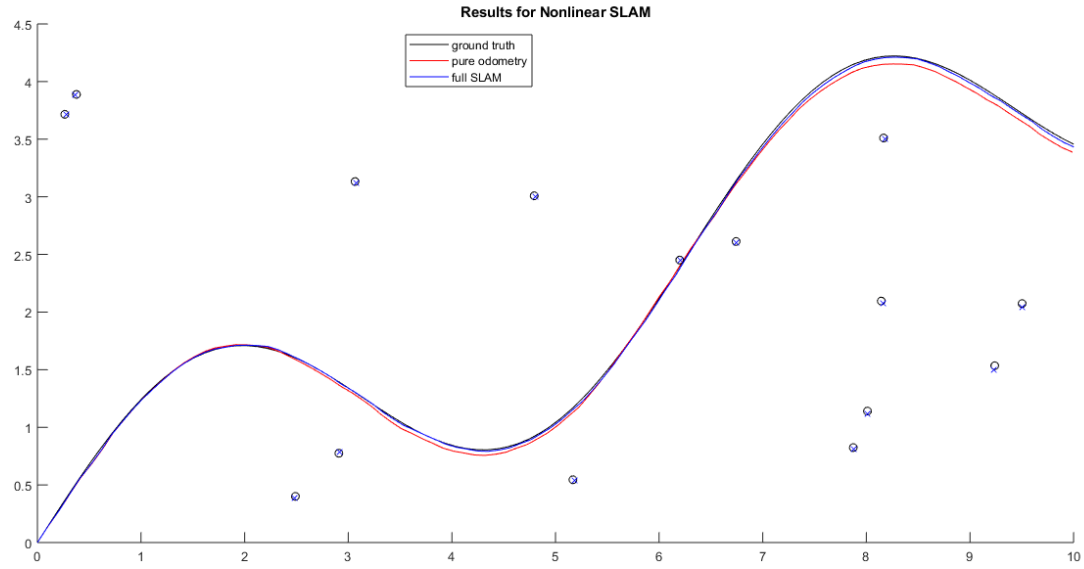

Figure 5 Output Trajectory for Nonlinear Dataset

```
rmse_odom =
    0.0579
rmse_slam =
    0.0153
```

The error from pure odometry is roughly 4 times as great as that from full SLAM.

(e) We implemented this nonlinear SLAM algorithm in an incremental fashion above. Why can't we solve it all at once in a batch optimization? (5 points)

For the nonlinear system, when we linearize the model, the state matrix is not constant. It depends on the values of state variables, the poses of the robots and the landmark positions, at each time point. So we cannot solve it in a batch optimization. In order to solve this problem we will start with an initial guess and update the estimation from previous estimation and current states each iteration to minimize the errors through time.