Hands-on Tutorial for Global Optimization in Matlab

Heng Yang

Massachusetts Institute of Technology

Table of Contents

This hands-on tutorial gives an example on how to use global optimization tools in Matlab. It uses the rotation averaging problem as an illustrative example and shows that this problem can be solved to *global optimality with certificates*, using *semidefinite relaxation*. Further, it demonstrates that large-scale semidefinite programming can be solved fast by exploiting low-rank structure of the solution.

Rotation Averaging

The problem of rotation averaging is defined as the task of determining a set of n absolute rotations $R_1, \ldots, R_n, R_i \in SO(d)$, given distinct estimated relative rotations $\widetilde{R}_{ij} \in SO(d)$, where $SO(d) \doteq \{R \in \mathbb{R}^{d\times d} | RR^T = R^T R = I_d, det(R) = +1\}$ is the *Special Orthogonal* group of dimension $d = \{2, 3\}$ that contains proper rotation matrices. We can define a graph $G = (V, E)$, where each vertex $i \in V$ is assigned an absolute rotation R_i, and each edge $(i, j) \in E$ is assigned a relative rotation $\widetilde{R}_{ij}$.

In the absence of measurement noise, it holds that

$$R_i \widetilde{R}_{ij} = R_j, \text{ for all } (i, j) \in E. \quad (1)$$

However, due to measurement noise, we do not expect the above equation to be true. Therefore, to obtain the optimal estimation of absolute rotations at each vertex, we adopt a least squares estimation:

$$R_1^\star, \ldots, R_n^\star = \operatorname{argmin}_{R_1,\ldots,R_n \in SO(d)} \sum_{(i,j)\in E} \| R_i \widetilde{R}_{ij} - R_j \|_F^2 . \quad (2)$$

where $\|R_i\widetilde{R}_{ij} - R_j\|_F$ denotes the chordal distance between two rotations $R_i\widetilde{R}_{ij}$ and R_j (in the absence of noise, the chordal distance is zero due to Eq. (1)). With straightforward calculations, the optimization (2) can be simplified as:

$$R_1^\star, \ldots, R_n^\star = \operatorname{argmin}_{R_1,\ldots,R_n \in SO(d)} \quad 2d|E| - 2\sum_{(i,j)\in E} \operatorname{tr}\left(R_i\widetilde{R}_{ij}R_j^T\right). \ (3)$$

where $|E|$ denotes the cardinality of the edge set E. Since the constant $2d|E|$ will not affect the result of the optimization, the final optimization problem is:

$$R_1^\star, \ldots, R_n^\star = \operatorname{argmin}_{R_1,\ldots,R_n \in SO(d)} \quad -\sum_{(i,j)\in E} \operatorname{tr}\left(R_i\widetilde{R}_{ij}R_j^T\right). \ (4)$$

Due to the non-convexity of $SO(d)$ constraints, it is hard to globally optimize problem (4). Our goal is to apply semidefinite relaxation so that we arrive at a convex SDP. Before applying SDP relaxation, we first write the problem as a quadratically constrained quadratic program (QCQP).

Quadratically Constrained Quadratic Program (QCQP)

The first simplification we apply is to remove the determinant constraint in $SO(d)$ and replace it with $O(d)$:

$$\begin{aligned} R_1^\star, \ldots, R_n^\star = \operatorname{argmin}_{R_1,\ldots,R_n} \quad & -\sum_{(i,j)\in E} \operatorname{tr}\left(R_i\widetilde{R}_{ij}R_j^T\right) \\ \text{subject to} \quad & R_iR_i^T = I_d, \forall i = 1,\ldots,n \end{aligned} \ (5)$$

where we have explicitly written the $R_i \in O(d)$ constraint as $R_iR_i^T = I_d$. Now we can stack all the unknown rotations $R_1, \ldots, R_n$ into a single matrix R:

$$R = [R_1^T,\ R_2^T,\ \ldots,\ R_n^T]^T \in O(d)^n \subset \mathbb{R}^{dn\times d}.$$

and all the measurements $\widetilde{R}_{ij}$ into a single matrix $\widetilde{R}$ (also including the "–" sign of the cost function):

$$\widetilde{R} = \begin{bmatrix} 0 & -a_{12}\widetilde{R}_{12} & \ldots & -a_{1n}\widetilde{R}_{1n} \\ -a_{21}\widetilde{R}_{21} & 0 & \ldots & -a_{2n}\widetilde{R}_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ -a_{n1}\widetilde{R}_{n1} & -a_{n2}\widetilde{R}_{n2} & \ldots & 0 \end{bmatrix}, \ \text{(R\_tilde)}$$

where $\widetilde{R}_{ij} = \widetilde{R}^T{}_{ji}$ and $a_{ij} = 1$ if $(i,j) \in E$, $a_{ij} = 0$ if $(i,j) \notin E$. Then problem (5) is equivalent to the following optimization:

$$\begin{aligned} R^\star = \operatorname{argmin}_R \quad & \operatorname{tr}\left(R^T\widetilde{R}R\right) \\ \text{subject to} \quad & R \in O(d)^n \end{aligned} \ (6).$$

Problem (6) is a QCQP because both the objective function and the constraints are quadratic functions of the unknown R.

Semidefinite Relaxation

Once we arrive at a QCQP, applying the semidefinite relaxation is straightforward.

Step 1: Lift

The first step is the lifting step. In this step, we create a new matrix variable $Z = RR^T$:

$$Z = \begin{bmatrix} R_1R_1^T & R_1R_2^T & \dots & R_1R_n^T \\ \star & R_2R_2^T & \dots & R_2R_n^T \\ \vdots & \vdots & \ddots & \vdots \\ \star & \star & \dots & R_nR_n^T \end{bmatrix} = \begin{bmatrix} I_d & R_1R_2^T & \dots & R_1R_n^T \\ \star & I_d & \dots & R_2R_n^T \\ \vdots & \vdots & \ddots & \vdots \\ \star & \star & \dots & I_d \end{bmatrix} \quad (7)$$

where the second equality comes from the constraint that $R \in O(d)^n$. Eq. (7) shows that the diagonal $d \times d$ blocks of matrix Z, denoted as $[Z]_{ii}$, are all identity matrices. At the same time, by the construction $Z = RR^T$ (Z is called the gram matrix of R), we have:

$$\begin{gathered} Z \succcurlyeq 0 \\ \operatorname{rank}(Z) = d \end{gathered}$$

In fact, one can show that $[Z]_{ii} = I_d, Z \succcurlyeq 0, \operatorname{rank}(Z) = d$ is equivalent to saying that Z has a rank d factorization $Z = RR^T$ such that $R \in O(d)^n$. Therefore, Problem (6) is equivalent to the following rank-constrained optimization:

$$\begin{aligned} R^\star(R^\star)^T = Z^\star = \operatorname{argmin}_Z \quad & \operatorname{tr}\left(\widetilde{R}Z\right) \\ \text{subject to} \quad & Z \succcurlyeq 0 \\ & [Z]_{ii} = I_d, \forall i = 1 \dots, n \\ & \operatorname{rank}(Z) = d \end{aligned} \quad (8)$$

Problem (8) is NP-hard in general due to the rank constraint.

Step 2: Drop

The second step is the dropping step, in which we drop the rank constraint. Then we arrive at a semidefinite program that we know how to solve in polynomial time:

$$\begin{aligned} R^\star(R^\star)^T = Z^\star = \operatorname{argmin}_Z \quad & \operatorname{tr}\left(\widetilde{R}Z\right) \\ \text{subject to} \quad & Z \succcurlyeq 0 \\ & [Z]_{ii} = I_d, \forall i = 1 \dots, n \end{aligned} \quad (9)$$

Problem (9) can be readily implemented in Matlab and solved to global minimum. However, usually an SDP relaxation can only guaranttee a lower bound of the original non-convex problem, which means that the optimal cost of Problem (9), $f^\star_{SDP}$, is a lower bound for the optimal cost of problem (8) and problem (6) $f^\star$:

$$f^\star_{SDP} \leq f^\star.$$

However, due to the structure of the rotation averaging problem, solving the SDP (9) actually solves the original pronblem (8) exactly.

Step 3: Certify Global Optimality

The third step is to certify global optimality by checking the rank of the solution of the SDP. If $\mathrm{rank}(Z^\star) = d$, then this means that dropping the rank constraint actually does not affect the optimization, because we end up with a rank d solution.

Step 4: Round

The last step is called rounding, which is to extract solution to the original non-convex problem given solution of the SDP. Using the rank d solution of the SDP, we can factorize $Z^\star = R_{est} R_{est}^T$ and extract the solution to the original nonconvex problem (note that the factorization is not unique unless we fix the first rotation R_1 to be identity). It is usually the case that R_{est} may not satisfy the original non-convex constraints, therefore, we sometimes need to project R_{est} into the feasible set of the original optimization (in the rotation averaging case, project to $SO(d)$). Using (the projected) R_{est}, we can evaluate the objective function and obtain the estimated cost f_{est}, then we have the following inequality:

$$f^\star_{SDP} \leq f^\star \leq f_{est} \quad (10)$$

where the first inequality comes from the SDP relaxation and the second inequality comes from the fact the $f^\star$ is the global minimum and hence must be smaller than the objective function evaluated at any point inside the feasible set. Using eq. (10), we can also calculate the relative duality gap:

$$\text{relative duality gap} = \frac{f_{est} - f^\star_{SDP}}{f_{est}} \quad (11)$$

The smaller the relaitive duality gap is, the tighter the SDP relaxation is. When the relative duality gap is zero, we can also certify global optimality and strong duality. In the next section, we will implement the SDP relaxation in Matlab.

Matlab Implementation

Prerequisite: Please download CVX and put the folder at the same level as `GlobalOptimizationTutorial`, then run `cvx_setup` from the Matlab console in the `cvx` folder.

We first setup this tutorial.

```
% setup path
setup
```

For a first example, let us consider rotation averaging on a simple 2D grid graph with noisy relative rotation and translation measurements. The following code snippt generates a random 2D grid graph, with ground truth poses, and the first rotation R_1 is set to be identity.

```
% build a random grid graph
```

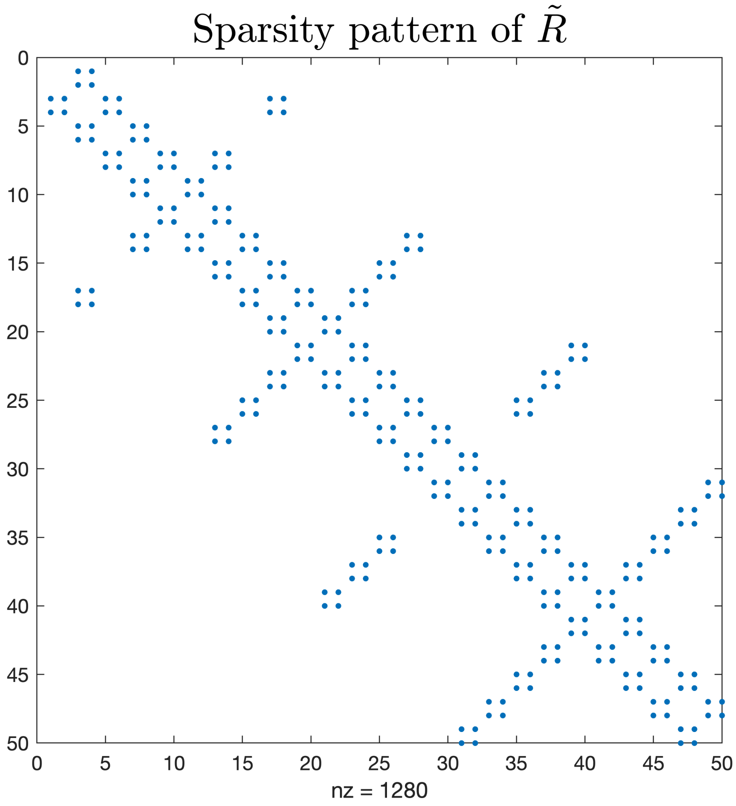
```
d = 2; % 2D planar graph
nrNodes = 100; % graph has n = 100 vertices
probLC = 0.8; % probability of loop closures in the grid graph
rotStd = 0.01; % noise standard deviation for rotation
tranStd = 0.01; % noise standard deviation for translation
graph = grid_random_graph_2D(nrNodes, ...
            'RotationStd', rotStd, ...
            'TranslationStd', tranStd, ...
            'Scale', 1.0, ...
            'LoopClosureProbability', probLC);
nrEdges = size(graph.edges,1);
% save the ground-truth rotations
R_gt = zeros(d*nrNodes,d);
for i=1:nrNodes
   R_gt(blkIndices(i,d),:) = rot2D(graph.poses_gt(i,3));
end
if norm(R_gt(1:2,1:2,1)-eye(2)) > 1e-6
   error('first rotation != identity')
end
fprintf(['Random 2D grid graph: number of nodes: %d, ' ...
    'number of edges: %d.\n'], ...
    nrNodes, nrEdges);
```

```
Random 2D grid graph: number of nodes: 100, number of edges: 160.
```

In the rotation averaging problem, we will not estimate the full pose (rotation and translation) of each node, but only focus on estimating the rotation of each node (one can actually marginalize out translations first).

In the `lib` folder, we provide a procedure `build_R_tilde` that builds the matrix $\widetilde{R}$ from edge measurements $\widetilde{R}_{ij}$. The following line build the data matrix $\widetilde{R}$ from the edge measurements, and the readers can check the sparsity pattern satisfies Eq. (R_tilde).

```
% build the data matrix R_tilde (d=2 for 2D planar graph)
R_tilde = build_R_tilde(graph.edges,nrNodes,d);
spy(R_tilde);
ylim([0, 50]), xlim([0,50]); % only visualize part of the sparsity pattern
title('Sparsity pattern of $\tilde{R}$',...
    'Interpreter','latex','fontsize',20)
```



Now we are ready to implement the core part of our algorithm -- solving the SDP relaxation in problem (9), using the random generated data matrix $\tilde{R}$:

```
% implement the SDP in problem (9)
t_start = tic;

cvx_quiet(true)
cvx_begin sdp % start CVX in SDP mode
cvx_precision best % ask for the best precision
cvx_solver sedumi % choose solver sedumi (sdpt3, mosek et al.)

% define the decision variable Z
variable Z(d*nrNodes, d*nrNodes) symmetric

% define the objective function (we include the constant term in Eq.(3) as well)
f_cost = trace(R_tilde * Z) + 2 * d * nrEdges;
minimize(f_cost)

% define the constraints of the SDP
subject to
Z >= 0 % the semidefinite constraint
for i = 1:nrNodes
    Z(blkIndices(i,d),blkIndices(i,d)) == eye(2)
end

% then solve the SDP!
```

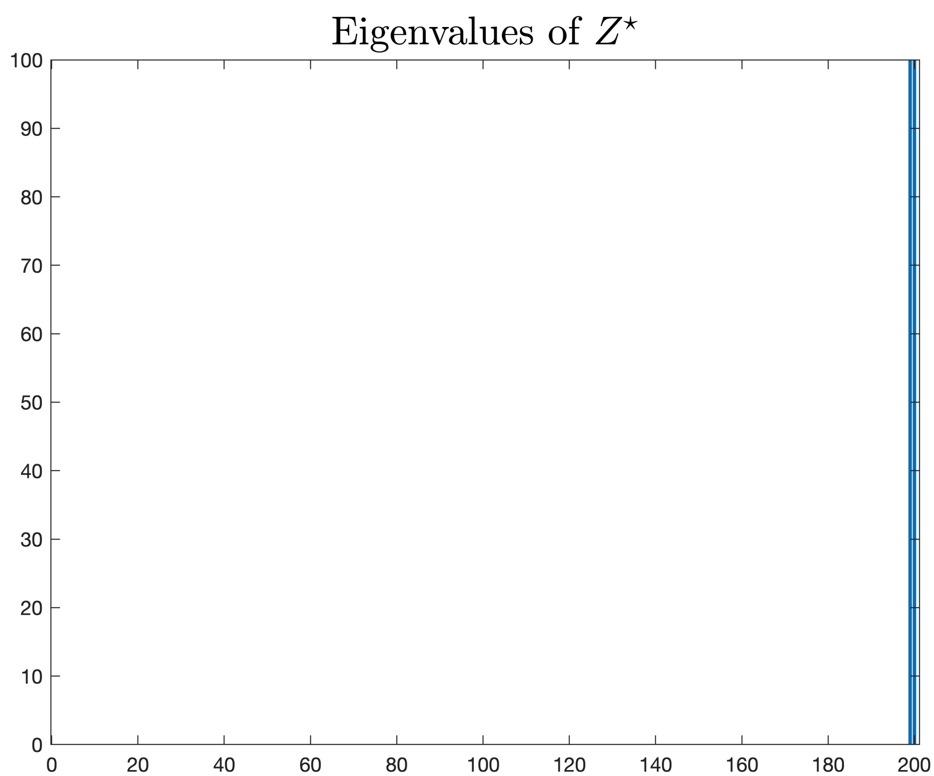
```
cvx_end

t_sdp = toc(t_start);
```

Now we can extract solutions of the SDP and check if the relaxation is exact:

```
% extract SDP solutions and check relaxation tightness
f_sdp = cvx_optval;
Z_star = full(Z);
rank_Z = rank(Z_star, 1e-3);
isExact = (rank_Z == d);
fprintf(['f_sdp = %g, rank_Z = %d, solver time = %g[s], ' ...
    'relaxation tightness: %s.\n'], ...
    f_sdp, rank_Z, t_sdp, string(isExact));
```

```
f_sdp = 0.0113658, rank_Z = 2, solver time = 2.12152[s], relaxation tightness: true.
```

```
figure;
bar(eig(Z_star));
title('Eigenvalues of $Z^\star$','Interpreter','latex','FontSize',20)
```



Because the rank of the SDP solution $\mathrm{rank}(Z^\star) = d = 2$, we can certify that the relaxation is indeed exact/tight! Now we can extract the optimal solution to the original non-convex problem (6) from the SDP solution. Because the first rotation $R_1 = I_d$ is identity, from Eq. (7) we can see that the first block row of Z contains all the rotation matrices $R_2, \ldots, R_n$.

```
% Extract solution to the original problem from SDP solution
R_est = zeros(d*nrNodes,d);
for i = 1:nrNodes
    R_est(blkIndices(i,d),:) = project2SO2( Z_star(1:d, blkIndices(i,d))' );
end
% evaluate the objective function on R_est to get f_est
f_est = trace(R_tilde * R_est * R_est') + 2 * d * nrEdges;
relative_duality_gap = abs(f_est - f_sdp) / f_est;
fprintf(['SDP relaxation: f_sdp = %g, f_est = %g, ' ...
    'relative duality gap = %g.\n'],...
    f_sdp,f_est,relative_duality_gap);
```

```
SDP relaxation: f_sdp = 0.0113658, f_est = 0.0113658, relative duality gap = 6.02008e-09.
```

From this result, we see that the relative duality gap is indeed zero (up to numerical accuracy). Now we can compare the estimated rotations with the ground-truth rotations to see how accurate our estimation is.

```
R_err = rad2deg( get_rot_error(R_est,R_gt,d) );
min_R_err = min(R_err);
max_R_err = max(R_err);
mean_R_err = mean(R_err);
fprintf(['SDP estimation: min R_err = %g[deg], ' ...
    'max R_err = %g[deg], mean R_err = %g[deg].\n'],...
    min_R_err,max_R_err,mean_R_err);
```

```
SDP estimation: min R_err = 1.04902e-13[deg], max R_err = 2.56078[deg], mean R_err = 1.08877[deg].
```

Fast Low-Rank SDP Solver

Solve Pose Graph Optimization on 2D Grids

We now use SE-Sync to solve the above problem and show that exploiting the low-rank and manifold property of the SDP structure can significantly speed up the process of solving large-scale SDP.

```
% convert data to SE-Sync type
sesync_measurements = convert2sesync(graph.measurements);
% default solver settings for SE-Sync and Manopt
default_solver_settings;
% call SE-Sync with random initialization
[SDPval, Yopt, xhat, Fxhat, SE_Sync_info, problem_data] = ...
    SE_Sync(sesync_measurements, Manopt_opts, SE_Sync_opts);
```

```
========== SE-Sync ==========

ALGORITHM SETTINGS:

SE-Sync settings:
 Initial level of Riemannian Staircase: 5
 Final level of Riemannian Staircase: 10
 Maximum number of iterations to perform for minimum eigenvalue computation in test for positive semidefi
 Tolerance for accepting an eigenvalue as numerically nonnegative in optimality verification: 0.0001 [def
 Using QR decomposition to compute orthogonal projection
```

```
 Initialization method: chordal [default]

Manopt settings:
 Stopping tolerance for norm of Riemannian gradient: 0.01
 Stopping tolerance for relative function decrease: 1e-05
 Maximum number of Hessian-vector products to evaluate in each truncated Newton iteration: 500
 Minimum number of trust-region iterations: 1
 Maximum number of trust-region iterations: 300
 Using incomplete zero-fill Cholesky preconditioner for truncated conjugate gradient inexact Newton step


INITIALIZATION:

Constructing auxiliary data matrices from raw measurements...
Constructed rotational connection Laplacian in 0.00763504 seconds
Constructed oriented incidence matrix in 0.000367127 seconds
Constructed translational observation and measurement precision matrices in 0.00267079 seconds
Constructed Laplacian for the translational weight graph in 2.3541e-05 seconds
Computed lower-triangular factor of reduced translational weight graph Laplacian in 5.282e-05 seconds
Caching additional product matrices ...
Auxiliary data matrix construction finished.  Elapsed computation time: 0.0124905 seconds

Constructing incomplete Cholesky preconditioner... Elapsed computation time: 0.000154657 seconds

Computing chordal initialization...
Elapsed computation time: 0.00615837 seconds

Solving Riemannian optimization problems using Manopt's "trustregions" solver


RIEMANNIAN STAIRCASE (level r = 5):
                                          f: +2.610026e+02   |grad|: 1.226332e+03
acc TR+   k:     1     num_inner:     1     f: +2.418179e+02   |grad|: 3.452916e+02   exceeded trust regi
acc       k:     2     num_inner:     3     f: +2.400533e+02   |grad|: 3.366358e+01   reached target resi
acc       k:     3     num_inner:     6     f: +2.400178e+02   |grad|: 1.169148e+00   reached target resi
acc       k:     4     num_inner:     3     f: +2.400178e+02   |grad|: 7.844562e-02   reached target resi
User defined stopfun criterion triggered; see options.stopfun.
Total time is 0.029158 [s] (excludes statsfun)

Checking second-order optimality...
Found second-order critical point! (minimum eigenvalue = -1.86265e-09, elapsed computation time 0.0620744


===== END RIEMANNIAN STAIRCASE =====

Rounding solution...
Elapsed computation time: 0.00195787 seconds

Recovering translational estimates...
Elapsed computation time: 0.000299107 seconds

Value of SDP solution F(Y): 240.018
Norm of Riemannian gradient grad F(Y): 0.0784456
Value of rounded pose estimate xhat: 240.018
Suboptimality bound of recovered pose estimate: -3.23496e-10

Total elapsed computation time: 0.131232 seconds


===== END SE-SYNC =====
```

```
% extract SE-Sync solution and time
t_sesync = SE_Sync_info.total_computation_time;
R_sesync = zeros(d*nrNodes,d);
```

```
R_1 = xhat.R(:,blkIndices(1,d));
for i = 1:nrNodes
    R_sesync(blkIndices(i,d),:) = R_1' * xhat.R(:,blkIndices(i,d));
end
R_sesync_err = rad2deg( get_rot_error(R_sesync,R_gt,d) );
min_R_sesync_err = min(R_sesync_err);
max_R_sesync_err = max(R_sesync_err);
mean_R_sesync_err = mean(R_sesync_err);
fprintf(['SDP estimation: min R_err = %g[deg], ' ...
    'max R_err = %g[deg], mean R_err = %g[deg].\n'],...
    min_R_err,max_R_err,mean_R_err);
```

```
SDP estimation: min R_err = 1.04902e-13[deg], max R_err = 2.56078[deg], mean R_err = 1.08877[deg].
```

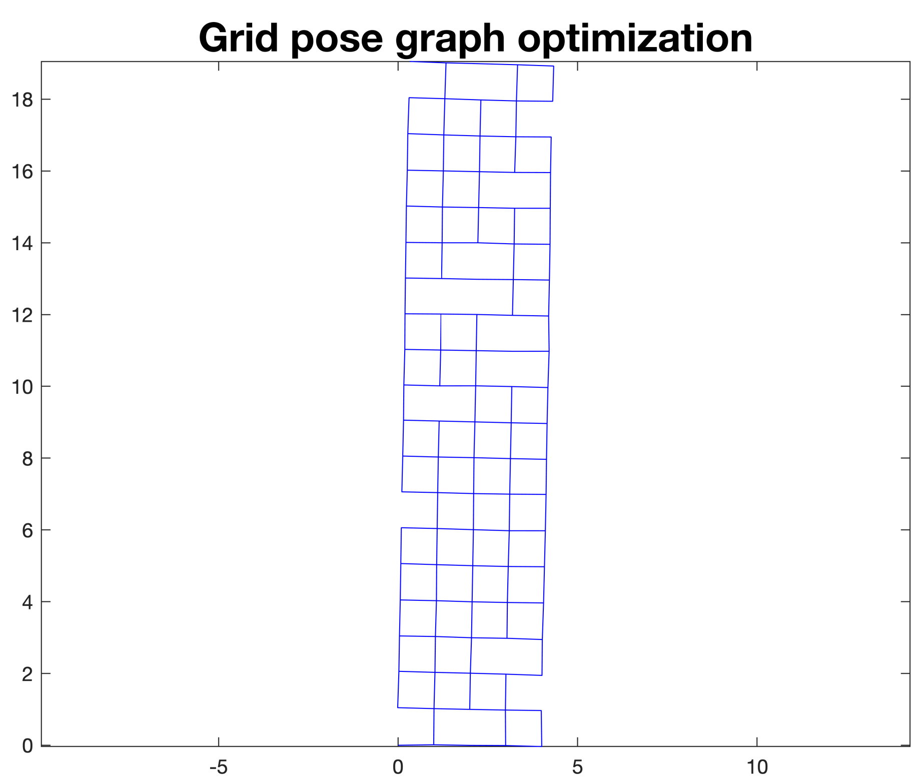
```
fprintf(['SE-Sync estimation: min R_err = %g[deg],' ...
    'max R_err = %g[deg], mean R_err = %g[deg].\n'],...
    min_R_sesync_err,max_R_sesync_err,mean_R_sesync_err);
```

```
SE-Sync estimation: min R_err = 1.72943e-14[deg],max R_err = 2.09956[deg], mean R_err = 1.01834[deg].
```

```
fprintf('SDP time: %g[s], SE-Sync time: %g[s].\n',t_sdp,t_sesync);
```

```
SDP time: 2.12152[s], SE-Sync time: 0.131232[s].
```

```
% visualize trajectories
plot_poses(xhat.t, xhat.R, sesync_measurements.edges, '-b', .25);
title('Grid pose graph optimization','FontSize',20)
```



Solve Pose Graph Optimization on Real Datasets

The power of SE-Sync is not limited to simple 2D grid graphs. We now show that SE-Sync can solve large-scale real-world datasets in seconds. Note that we are using the Matlab version of SE-Sync. With the C++ version of SE-Sync, it can solve these problems in a fraction of second. The readers are encouraged to try out SE-Sync's C++ version for real-time applications.

```
% load real dataset from g2o file
% g2o_file = './PGO_data/intel.g2o';
% g2o_file = './PGO_data/MIT.g2o';
g2o_file = './PGO_data/CSAIL.g2o';

real_pgo_data = load_g2o_data(g2o_file);
% get some information about the data
nrNodes = max(max(real_pgo_data.edges));
nrEdges = length(real_pgo_data.kappa);
d = length(real_pgo_data.t{1});
fprintf('Processed input file %s: # of poses: %d, # of measurements: %d.\n', ...
    g2o_file, nrNodes, nrEdges);
```

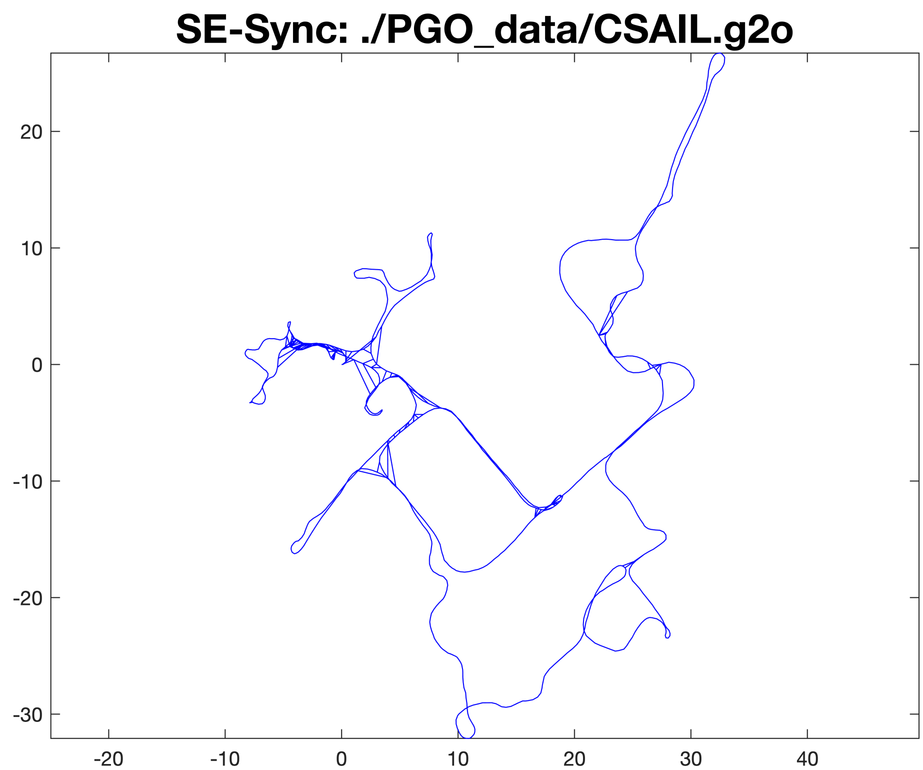
```
Processed input file ./PGO_data/CSAIL.g2o: # of poses: 1045, # of measurements: 1172.
```

```
% call SE-Sync to solve
[SDPval, Yopt, xhat, Fxhat, SE_Sync_info, problem_data] = ...
    SE_Sync(real_pgo_data, Manopt_opts, SE_Sync_opts);
```

```
========== SE-Sync ==========

ALGORITHM SETTINGS:

SE-Sync settings:
 Initial level of Riemannian Staircase: 5
 Final level of Riemannian Staircase: 10
 Maximum number of iterations to perform for minimum eigenvalue computation in test for positive semidefi
 Tolerance for accepting an eigenvalue as numerically nonnegative in optimality verification: 0.0001 [def
 Using QR decomposition to compute orthogonal projection
 Initialization method: chordal [default]

Manopt settings:
 Stopping tolerance for norm of Riemannian gradient: 0.01
 Stopping tolerance for relative function decrease: 1e-05
 Maximum number of Hessian-vector products to evaluate in each truncated Newton iteration: 500
 Minimum number of trust-region iterations: 1
 Maximum number of trust-region iterations: 300
 Using incomplete zero-fill Cholesky preconditioner for truncated conjugate gradient inexact Newton step


INITIALIZATION:

Constructing auxiliary data matrices from raw measurements...
Constructed rotational connection Laplacian in 0.0378988 seconds
Constructed oriented incidence matrix in 0.00026227 seconds
Constructed translational observation and measurement precision matrices in 0.0219566 seconds
Constructed Laplacian for the translational weight graph in 8.2033e-05 seconds
Computed lower-triangular factor of reduced translational weight graph Laplacian in 0.00160875 seconds
Caching additional product matrices ...
Auxiliary data matrix construction finished.  Elapsed computation time: 0.0640126 seconds
```

```
Constructing incomplete Cholesky preconditioner... Elapsed computation time: 0.000266319 seconds

Computing chordal initialization...
Elapsed computation time: 0.037951 seconds

Solving Riemannian optimization problems using Manopt's "trustregions" solver


RIEMANNIAN STAIRCASE (level r = 5):
                                            f: +3.171810e+01   |grad|: 1.090066e+01
acc       k:     1     num_inner:     8     f: +3.170426e+01   |grad|: 8.203761e-01   reached target resi
acc       k:     2     num_inner:    22     f: +3.170372e+01   |grad|: 7.493091e-02   reached target resi
acc       k:     3     num_inner:    16     f: +3.170372e+01   |grad|: 4.317102e-03   reached target resi
Gradient norm tolerance reached; options.tolgradnorm = 0.01.
Total time is 0.243796 [s] (excludes statsfun)

Checking second-order optimality...
Found second-order critical point! (minimum eigenvalue = 4.36557e-10, elapsed computation time 1.24701 se


===== END RIEMANNIAN STAIRCASE =====

Rounding solution...
Elapsed computation time: 0.0152591 seconds

Recovering translational estimates...
Elapsed computation time: 0.000412364 seconds

Value of SDP solution F(Y): 31.7037
Norm of Riemannian gradient grad F(Y): 0.0043171
Value of rounded pose estimate xhat: 31.7037
Suboptimality bound of recovered pose estimate: -3.83338e-12

Total elapsed computation time: 1.63348 seconds


===== END SE-SYNC =====
```

```
plot_poses(xhat.t, xhat.R, real_pgo_data.edges, '-b', .25);
title(['SE-Sync: ',g2o_file],'Interpreter','none','FontSize',20)
```



References

[1] Rosen, D. M., Carlone, L., Bandeira, A. S., & Leonard, J. J. (2018). SE-Sync: a certifiably correct algorithm for synchronization over the Special Euclidean group. *Intl. J. of Robotics Research*.

[2] Eriksson, A., Olsson, C., Kahl, F., & Chin, T.-J. (2018). Rotation averaging and strong duality. In *IEEE Conf. on Computer Vision and Pattern Recognition (CVPR)*.