# SQUIC

SOFTWARE FOR SPARSE PRECISION MATRIX ESTIMATION

## User Manual

Version 1.0

November 20, 2023

ARYAN EFTEKHARI | LISA GAEDKE-MERZHÄUSER | DIMOSTHENIS PASADAKIS

MATTHIAS BOLLHÖFER | SIMON SCHEIDEGGER | OLAF SCHENK

# Table of Contents



**About this document**

This file is intended as a user manual for the ACM TOMS software package submission.

# 1 Introduction

This software is an implementation of the **S**parse **QU**adratic approximation of **I**nverse **C**ovariance matrix (SQUIC) algorithm [3], a second-order method designed for the solving $\ell_1$-regularized Gaussian maximum likelihood problem. Based on the foundational work of [5], this algorithm is developed for large-scale, and performance-oriented sparse inverse covariance or precision matrix estimation. This user manual serves as a resource for the SQUIC package and its interfaces.

For the dataset $\mathbf{Y} \in \mathbb{R}^{p \times n}$, consisting of $p$ random variables and $n$ samples, SQUIC computes a sparse estimate of the precision matrix and its sparse approximate inverse, denoted as $\widehat{\mathbf{\Theta}}$ and $\widehat{\mathbf{\Theta}}^{\text{inv}}$, respectively. The estimation process employs a *matrix tuning parameter* $\mathbf{\Lambda} \in \mathbb{R}^{p \times p}$, which is indirectly defined using $\mathbf{M} \in \mathbb{R}^{p \times p}$ and a *scalar tuning parameter* $\lambda > 0$. This relationship is summarized as follows:

$$\mathbf{\Lambda}_{ij} := \begin{cases} \mathbf{M}_{ij}, & \text{if } \mathbf{M}_{ij} \neq 0, \\ \lambda & \text{otherwise.} \end{cases} \tag{1}$$

In many cases, it is appropriate to define the nonzero entries of $\mathbf{M}$ as constants, such that, for example, for $\eta \in (0, \lambda]$, $\mathbf{M}_{ij} = \eta$ for all its nonzero entries. The larger the value of $\mathbf{\Lambda}_{ij}$, the more the corresponding value $\widehat{\mathbf{\Theta}}_{ij}$ is penalized for being nonzero. If $\mathbf{M}$ is not specified and only the scalar tuning parameter $\lambda$ is specified, we set $\mathbf{\Lambda}_{ij} = \lambda$ for all its entries. In Figure 1, we visualize the main inputs and outputs of SQUIC, alongside its primary computational components.

The source code for the software described in this manuscript is available under general public licenses (GPL 3). The software is packaged as `libSQUIC` and intended for use on Linux and Mac systems (both X86 and ARM architectures). It is written in C++, parallelized with OpenMP, and features interface packages for R and Python. This codebase incorporates the SQUIC algorithm from [1], along with parallel and blocking components in [2] and [4], respectively.

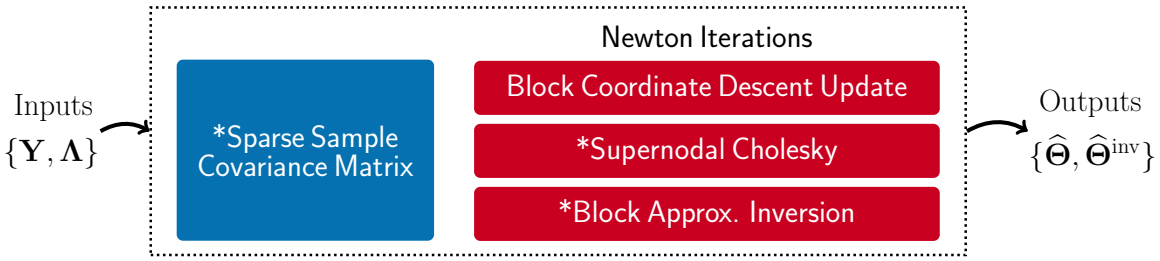


Figure 1: A visualization of the SQUIC package which includes the main inputs $\mathbf{Y}$ and $\mathbf{\Lambda}$, as well as the outputs $\widehat{\mathbf{\Theta}}$ and $\widehat{\mathbf{\Theta}}^{\text{inv}}$, corresponding to the data matrix, matrix tuning parameter, estimated precision matrix and its sparse approximate inverse, respectively. Note that we can set $\mathbf{\Lambda}_{ij} = \lambda$ for all its entries, where $\lambda$ is the scalar tuning parameter; see (1) for details. Also shown are the main computational components, which include the upfront sparse sample covariance matrix computation (blue), alongside the Newton iteration (red), which includes block coordinating descent, supernodal Cholesky factorization, and block approximate matrix inversion. Shared-memory parallel components are marked with an "*".

# 2 Compiling and Installation

The source code for the SQUIC library is located in `Src/lib`. This section provides instructions for compiling and installing the `libSQUIC` shared library, along with its R and Python interfaces. The necessary external packages for compilation include:

- BLAS, LAPACK, and OpenMP
- C/C++/Fortran compilers and Cmake

Mac users are recommended to use Homebrew to install the GNU Fortran compilers, OpenMP, and Cmake by running `brew install gcc libomp cmake`. The default Mac distribution includes both the C/C++ compilers and the BLAS/LAPACK libraries. Linux users can install necessary packages with their system's package manager. For example, on Ubuntu, this is done using the command `sudo apt-get install build-essential libblas-dev liblapack-dev cmake`. [1]

The R interface library dependencies include `Matrix`, `Rcpp` and `RcppArmadillo`, and the Python interface dependencies include the `pip3` package manager, `NumPy` and `SciPy`. Note that, with the exception of `pip3`, which is included with Python version 3, all dependencies for both R and Python are automatically installed as part of the setup process described below.

❶ Navigate to the `Src/lib` directory within the source code to compile `CHOLMOD`, its dependencies, and `libSQUIC`. To complete the process, execute the following commands:

```
cd Src/lib
make
```

❷ Install `libSQUIC` in the default `~/` directory.

```
make install
```

❸ Install the R and Python interface packages (discussed in further detail in Sections 3 and 4), and run a simple test for each. By default, the interface packages are installed in a local user directory. For R, this is `~/R`, and for Python, it's the platform-specific user install directory (see, `pip3 help` `--user` flag for details).[2]

```
make install_r
make install_py
```

[1] If different types of BLAS or LAPACK are used, the `LDFLAGS` in `Src/lib/makefile` must be adjusted accordingly. To use Intel Math Kernel Library (MKL), set the `$MKLROOT` environment variable by executing `source /opt/intel/oneapi/mkl/latest/env/vars.sh intel64` (note: this command may vary based on your MKL version). Then, adjust your compiler and linker flags according to your system. For specific flag details, refer to the Intel MKL Link Line Advisor.

[2] If you wish to perform a global installation of the R and Python packages (which may require root access), use, e.g., the command `make install_r R_PYTHON_INTERFACE_GLOBAL_INSTALL=true`.

# 3 SQUIC for R

The R interface is located at `Src/r`. The installation procedure of `libSQUIC`, as outlined in Section 2, automatically installs R interface package and required dependencies. The user can manually install or reinstall the interface package by following these steps:

❶ Install `libSQUIC` in the recommended home directory, i.e., `~/`.

❷ Install dependencies `Matrix`, `Rcpp` and `RcppArmadillo` from the R command-line using

```
install.packages(c('Matrix','Rcpp', 'RcppArmadillo'))
```

❸ Install the library from the R command-line using

```
install.packages('Src/r', repos = NULL, type = 'source')
```

The environment variable `SQUIC_LIB_PATH` defines the location of `libSQUIC`. If this variable is not set, then the default location of `libSQUIC` is `~/`. To change the default location, in R use the command `Sys.setenv(SQUIC_LIB_PATH="/path/to/squic/")`, replacing `/path/to/squic` with the path of where `libSQUIC` can be found. If you do not have root access you will be prompted for a local installation of SQUIC, along with any required dependencies.

## 3.1 List of variables

The `SQUIC()` function call serves as the interface function for `libSQUIC`, invoking the SQUIC algorithm for sparse precision matrix estimation. The list of input and output variables is shown below and can also be viewed via the help command `help(SQUIC)`.

```
out1 <- SQUIC(Y,lambda,max_iter,tol,tol_inv,verbose,M,X0,W0);
out2 <- SQUIC(Y,lambda,max_iter = 0,tol,verbose,M)
```

All sparse matrix formats utilize the standard `Matrix` package.

### Inputs

- `Y` : Input data matrix ($p \times n$ dense matrix).
- `lambda` : Scalar tuning parameter a nonzero positive number.
- `max_iter` [default 100]: Maximum number of Newton iterations[3]
- `tol` [default $10^{-3}$]: Tolerance for convergence.
- `tol_inv` [default $10^{-4}$]: Tolerance for approximate matrix inversion.
- `verbose` [default 1]: Verbosity level as 0 or 1.
- `M` [default `NULL`]: Symmetric sparse matrix ($p \times p$ sparse matrix).[4]
- `X0` [default `NULL`]: Initial value of precision matrix ($p \times p$ sparse matrix).[5]
- `W0` [default `NULL`]: Initial value of inverse precision matrix ($p \times p$ sparse matrix).[5]

[3] Setting `max_iter=0` will return the *sparse (thresholded) sample covariance matrix.*

[4] We define $\mathbf{\Lambda}_{ij} = \mathbf{M}_{ij}$ if $\mathbf{M}_{ij} \neq 0$ and $\mathbf{\Lambda}_{ij} = \lambda$, if otherwise. When `M=NULL` (or `None` in Python), we default to the, so called, "scalar-tuning parameter" for which $\mathbf{\Lambda}_{ij} = \lambda$.

[5] If either `X0` or `W0` is set to `NULL` (or `None` in Python), both `X0` and `W0` are set to identity.

**Outputs**

- `out1$X` : Estimated precision matrix ($p \times p$ sparse matrix).
- `out1$W` : Sparse approximate inverse of the precision matrix ($p \times p$ sparse matrix).
- `out1$info_time_total` : Total runtime.
- `out1$info_time_sample_cov` : Runtime of the sample covariance matrix.
- `out1$info_time_optimize` : Runtime of the Newton steps.
- `out1$info_time_factor` : Runtime of matrix factorization.
- `out1$info_time_approximate_inv` : Runtime of approximate matrix inversion.
- `out1$info_time_coordinate_upd` : Runtime of coordinate descent update.
- `out1$info_objective` : Objective value at each Newton iteration.
- `out1$info_logdetX` : Log determinant of the estimated precision matrix.
- `out1$info_trSX` : Trace of sample cov. times the estimated precision matrix.

- `out2$S` : Sparse sample covariance matrix ($p \times p$ sparse matrix).
- `out2$info_time_total` : Total runtime.
- `out2$info_time_sample_cov` : Runtime of the sample covariance matrix.

## 3.2 Usage

In the basic example outlined below, we will use SQUIC to estimate the precision matrix of a synthetically generated dataset with correlated random variables, where the true precision matrix is tridiagonal. [6]

```
library(Matrix)
library(SQUIC)

# basic test parameters
p       <- 1024
n       <- 100
lambda  <- .4

# generate a tridiagonal matrix &  dataset
set.seed(1)
iC_star <- Matrix::bandSparse(p, p, (-1):1, list(rep(-.5, p-1), rep(1.25, p),
 ↪  rep(-.5, p-1)));
z       <- replicate(n,rnorm(p))
iC_L    <- chol(iC_star)
Y       <- matrix(solve(iC_L,z),p,n)

# run SQUIC
out <- SQUIC(Y,lambda)
```

[6] There may be slight variations in the results across different machines due to *round-off errors* These differences are visible in the sparse approximation of the estimated inverse of the precision matrix `W` and are associated with the specific permutation used by the underlying matrix factorization routine, which are not related to the random seed for data generation.

With the default verbosity level the output is as shown below.[7]

```
------------------------------------------------------------
                    SQUIC Version 1.0
------------------------------------------------------------
Input Matrices
 nnz(X0)/p:   1.000000e+00
 nnz(W0)/p:   1.000000e+00
 nnz(M)/p:    ignored
 Y:           1024 x 100
Runtime Configs
 Sample Cov.:   Deterministic
 CordDec Vers:  Collective coordinate descent update
 Inversion:     Approx. block Neumann series
 Fact. Routine: CHOLMOD
Parameters
 verbose:     1
 lambda:      4.000000e-01
 max_iter:    100
 term_tol:    1.000000e-03
 inv_tol:     1.000000e-04
 threads:     12
#SQUIC Started
* sample covariance matrix S: time=2.36e-03 nnz(S)/p=6.91e+00
* iter=1 time=4.89e-03 obj=1.60e+03 |delta(obj)|/obj=1.56e-01 nnz(X,L,W)/p=
    [6.91e+00 7.54e+01 1.69e+00] lns_iter=3
* iter=2 time=7.23e-03 obj=1.56e+03 |delta(obj)|/obj=3.13e-02 nnz(X,L,W)/p=
    [6.91e+00 7.54e+01 5.75e+01] lns_iter=2
* iter=3 time=3.19e-02 obj=1.55e+03 |delta(obj)|/obj=6.62e-03 nnz(X,L,W)/p=
    [6.91e+00 7.54e+01 5.78e+01] lns_iter=1
* iter=4 time=3.06e-02 obj=1.54e+03 |delta(obj)|/obj=2.71e-04 nnz(X,L,W)/p=
    [6.91e+00 7.54e+01 9.01e+01] lns_iter=1
#SQUIC Finished: time=8.60e-02 nnz(X,W)/p=[6.91e+00 9.01e+01]
```

Input details are provided under "Input Matrices", where `X0` and `W0` denote the initial values of the precision matrix and its inverse (defaulting to identity matrices). The absence of `M` is noted, and the size of the input data `Y` is specified as $p = 1024$, $n = 100$. In the "Runtime Configs" section, key computational operations are summarized. The "Parameters" section lists parameters used in the overall optimization procedure, where the maximum number of threads available on the system is automatically utilized. Here `iter`, `nnz`, and `lns_iter` represent the Newton iteration index, number of nonzeros, and the number of line-search iterations, respectively. For instance, `nnz(X,L,W)/p` indicates the number of nonzeros per row for the precision matrix, its Cholesky factor, and the sparse approximate inverse, respectively.

[7] In R notebooks the verbose output may not be visible from within the notebook (see, e.g., here). This issue is not present when using the terminal or RStudio, and it does not affect the results.

# 4 SQUIC for Python

The Python (V3) interface for SQUIC is located at `Src/python`. The installation of `libSQUIC` detailed in Section 2, automatically installs the Python interface package and required dependencies using the `pip3` package manager, which is bundled with Python. The user can manually install or reinstall the interface package by following these steps:

❶ Install `libSQUIC` in the recommended home directory, i.e., `~/`.

❷ Install dependencies `NumPy`,and `SciPy` using `pip3` form the command-line

```
pip3 install numpy scipy --user
```

❸ Install the interface package using `pip3` form the command-line.

```
pip3 install Src/python --user
```

Use `export SQUIC_LIB_PATH=/path/to/squic/` , replacing `/path/to/squic/` with the path of where `libSQUIC` can be found. If this variable is not set, the default location of `libSQUIC` is `~/`. Use `pip3 install Src/python` for a global installation of the Python interface.

## 4.1 List of variables

The `squic.run()` function call serves as the Python interface function for the `libSQUIC`, invoking the SQUIC algorithm for sparse precision matrix estimation. The list of input and output variables is shown below and can also be viewed via the help command `help(squic)` . The description of the function here is exactly the same as the R interface discussed in Section 3. The sole difference between the input variable of the R and Python interface is that `lambda` is a reserved keyword in Python, so `l` is used instead. For completeness, the input and output variable descriptions are provided below.

```
out1 = squic.run(Y,l,max_iter,tol,tol_inv,verbose,M,X0,W0);
out2 = squic.run(Y,l,max_iter = 0,tol,verbose,M);
```

All sparse matrix formats utilize the `scipy` package.

### Inputs

- `Y` :Input data matrix ($p \times n$ dense matrix).
- `l` : Scalar tuning parameter a nonzero positive number.
- `max_iter` [default 100]: Maximum number of Newton iterations.[3]
- `tol` [default $10^{-3}$]: Tolerance for convergence.
- `tol_inv` [default $10^{-4}$]: Tolerance for approximate matrix inversion.
- `verbose` [default 1]: Verbosity level as 0 or 1.
- `M` [default `None` ]: Symmetric sparse matrix ($p \times p$ sparse matrix)..[4]
- `X0` [default `None` ]: Initial value of precision matrix ($p \times p$ sparse matrix).[5]
- `W0` [default `None` ]: Initial value of inverse precision matrix ($p \times p$ sparse matrix).[5]

**Outputs**

- `out1[0]` : Estimated precision matrix ($p \times p$ sparse matrix) .
- `out1[1]` : Sparse approximate inverse of the precision matrix ($p \times p$ sparse matrix).
- `out1[2]` : List of different component runtimes.
    - `out1[2][0]` : Total runtime.
    - `out1[2][1]` : Runtime of the sample covariance matrix.
    - `out1[2][2]` : Runtime of the Newton steps.
    - `out1[2][3]` : Runtime of matrix factorization.
    - `out1[2][4]` : Runtime of approximate matrix inversion.
    - `out1[2][5]` : Runtime of coordinate descent update.
- `out1[3]` : Objective value at each Newton iteration.
- `out1[4]` : Log determinant of the estimated precision matrix..
- `out1[5]` : Trace of sample covariance matrix times the estimated precision matrix.

- `out2[1]` : Sparse sample covariance matrix ($p \times p$ sparse matrix) .
- `out2[2]` : List of different compute times.
    - `out2[2][0]` : Total runtime.
    - `out2[2][1]` : Runtime of the sample covariance matrix.

## 4.2 Usage

In the basic example outlined below, we will use SQUIC to estimate the precision matrix of a synthetically generated dataset with correlated random variables, where the true precision matrix is tridiagonal.[6]

```
import numpy as np
import squic

# basic test parameters
p = 1024
n = 100
l = .4
# generate a tridiagonal matrix &  dataset
np.random.seed(1)
a = -0.5 * np.ones(p-1)
b = 1.25 * np.ones(p)
iC_star = np.diag(a,-1) + np.diag(b,0) + np.diag(a,1)
L = np.linalg.cholesky(iC_star)
Y = np.linalg.solve(L.T,np.random.randn(p,n))
# run SQUIC
[X,W,info_times,info_objective,info_logdetX,info_trSX] = squic.run(Y,l)
```

Using the default verbosity level the output is as shown below. This output differs compared to the R example in Section 3, as the generated random data differs. With this said, reruns of the same script, using the same random seed, will result in the same results.

```
------------------------------------------------------------
                    SQUIC Version 1.0
------------------------------------------------------------
Input Matrices
 nnz(X0)/p:   1.000000e+00
 nnz(W0)/p:   1.000000e+00
 nnz(M)/p:    ignored
 Y:           1024 x 100
Runtime Configs
 Sample Cov.:   Deterministic
 CordDec Vers:  Collective coordinate descent update
 Inversion:     Approx. block Neumann series
 Fact. Routine: CHOLMOD
Parameters
 verbose:     1
 lambda:      4.000000e-01
 max_iter:    100
 term_tol:    1.000000e-03
 inv_tol:     1.000000e-04
 threads:     12

#SQUIC Started
* sample covariance matrix S: time=1.91e-02 nnz(S)/p=6.49e+00
* iter=1 time=4.34e-02 obj=1.60e+03 |delta(obj)|/obj=1.60e-01 nnz(X,L,W)/p=
    [6.49e+00 6.61e+01 1.67e+00] lns_iter=3
* iter=2 time=6.63e-02 obj=1.55e+03 |delta(obj)|/obj=3.11e-02 nnz(X,L,W)/p=
    [6.49e+00 6.61e+01 5.30e+01] lns_iter=2
* iter=3 time=4.80e-02 obj=1.54e+03 |delta(obj)|/obj=6.42e-03 nnz(X,L,W)/p=
    [6.49e+00 6.61e+01 5.29e+01] lns_iter=1
* iter=4 time=4.44e-02 obj=1.54e+03 |delta(obj)|/obj=2.53e-04 nnz(X,L,W)/p=
    [6.49e+00 6.61e+01 8.24e+01] lns_iter=1
#SQUIC Finished: time=2.39e-01 nnz(X,W)/p=[6.49e+00 8.24e+01]
```

Note that the verbose output of `libSQUIC` is not visible in notebooks; but it does not affect the results.[8] This issue does not occur when using Python in a non-notebook environment. By using the `wurlitzer` package, we can sidestep this issue with the following lines of code before the `squic.run` notebook function call.

```
!pip install wurlitzer
%load_ext wurlitzer
```

[8] This is a known issue; see, e.g., pypi.org/project/wurlitzer for details.

# 5 Numerical Tests

Experiments validating the performance and accuracy of the SQUIC algorithm, as well as its comparison with similar packages, are located in `/Replication`. The five experiments include assessments of the sparse approximate matrix inversion routine (`test_inv.R`), evaluations of accuracy using the scalar tuning parameter (`test_scal_acc.R`), matrix tuning parameter (`test_mat_acc.R`), performance analyses (`test_perf.R`), and a microarray classification case study (`test_lda.R`). The experiments, implemented in R, can be run by following the steps below:

❶ Download the synthetic datasets (total size 20GB) by executing the following command in the terminal within the `/Replication` directory.

```
curl -o temp.zip https://drive.switch.ch/index.php/s/Avxc8ugzHXvW5jN/download
unzip temp.zip -d squic_test_db/
```

Note that the default location of the datasets should be `Replication/squic_test_db/`. If this is changed the file `Replication/load_data.R` must be modified accordingly.

❷ Install the R libraries required for the experiments. These packages are not dependencies of SQUIC, but are used in the numerical experiments.

```
Rscript install_dependencies.R
```

❸ The test scripts for the five experiments can now be run individually. They all start with "`test_`" and can be run from the terminal; for example, to execute the test concerning the matrix inversion experiments, use the following command:

```
Rscript test_inv.R
```

MATLAB scripts, as opposed to R, are utilized for the testing of the `SNETCH` algorithm. Experiments evaluating the accuracy of the scalar tuning parameter and the performance of the `SNETCH` package can be found in `/Replication/test_SNETCH` and can be executed using the scripts `test_scal_acc_SNETCH.m` and `test_perf_SNETCH.m`, respectively.

## 5.1 Microarray Classification Case Study

This section describes the microarray classification test (`test_lda.R`), using SQUIC for high-dimensional gene expression classification via inear discriminant analysis (LDA). This complements Section 4.2 in [3], which discusses results and test timings. The collection of datasets used for our test are available from the `datamicroarray` package.[9] We provide R code snippets for key steps and use the transcription of the variable name like $\eta$ and $\mathbf{M}$ as `eta` and `M` in the code.

[9] The `datamicroarray` package is available at github.com/ramhiser/datamicroarray.

We begin by loading the datasets and employ a stratified approach to split them into training and testing sets. Using the `caret` package,[10] we select 70% of the samples, comprising of the feature values and their corresponding labels from each class, to form the training set. This data is referred to as `data_train` and `labels_train`. The remainder of the dataset is used for testing, for which we follow the same naming standard. To simplify the selection of the tuning parameter, matrix or its scalar form (see, e.g., Section 1 for further details), we normalized the data using the diagonal matrix `D` as shown below. We later undo this scaling once the precision matrix estimation procedure is completed.

```
n_train <- dim(data_train)[2]
a <- (apply(data_train,1,var))*(n_train-1)/n_train
D <- Diagonal(x=1/sqrt(a))
data_train_scaled <- D%*%data_train
```

In this study, we define two versions of the tests: one using the matrix tuning parameter $\mathbf{\Lambda}$, and the other using just the scalar tuning parameter $\lambda$. To define $\mathbf{\Lambda}$, we need to first specify $\mathbf{M}$, which we now describe. In the first step, we generate a complete graph with edge weights represented by the Euclidean distance. Subsequently, we compute the adjacency matrix, denote here as `G`, corresponding to the Minimal Spanning Tree (MST). The MST is a sparsely connected graph with the minimum sum of edge weights required for connectivity. The nonzero pattern of `G` corresponds to pairs of nodes with low Euclidean distances between them, indicating high similarity. By using the `emstreeR` package,[11] `G` can be efficiently computed as follows.

```
mst <- ComputeMST(as.matrix(data_train))
G <- sparseMatrix(dims=c(p,p), i=mst$to, j=mst$from, x=rep(1, p))
G <- forceSymmetric(G,uplo="L") + Diagonal(nrow(G))
```

The last operation enforces symmetry and adds diagonal entries to `G`, as the adjacency matrix of the MST will not have diagonal entries or self-links. Note that the diagonal entries of `G` are added simply so that they are reflected in `M`—the adjacency matrix of a MST will have zero for its diagonal entries. We can now define `M<-G*eta`, where for this study we fix `eta <- 0.1`.

Using SQUIC, we now compute the required precision matrices, for which an appropriate $\lambda$ must be selected. In this example, we set $\lambda = 0.8$, but this choice depends on the dataset. For further details on the selection of $\lambda$ for the different datasets, please refer to [3] or the replication script. For the estimation procedure using the matrix tuning parameter, we fix $\lambda = 0.95$ across all dataset tests. The respective function calls for using just the scalar tuning parameter, and the case where the matrix tuning parameter is used, are as follows:

[10] The `caret` package is a collection of functions for training and plotting classification and regression models, and is available at cran.r-project.org/package=caret.

[11] The `emstreeR` package computes an Euclidean MST from data, and is available at cran.r-project.org/package=emstreeR.

```
squic_out   <- SQUIC(Y=data_train,lambda=0.80,M=NULL,tol=1e-3,tol_inv=1e-3)
squic_out_M <- SQUIC(Y=data_train,lambda=0.95,M=eta*G,tol=1e-3,tol_inv=1e-3)
```

After computing $\widehat{\boldsymbol{\Theta}}$, we *undo* the normalization operation; for example, for the scalar tuning estimate, we would rescale the matrix as follows: `D%*%squic_out$X%*%D`.

The LDA method is subsequently applied to classify the samples in the testing dataset. For each class $k = 1,\ldots,K$, where $K$ is the number of classes in the given dataset (i.e., the labels), we compute the sample means using `data_train`. Denoting the estimated precision matrices as `theta_est`, we compute the LDA score, `lda_score`, for each class and assign a predicted class based on the one with the highest LDA score.

```
for (k in 1:K) {
  # prior probability for class k
  prior <- length(which(labels_train == k)) / n_train
  # sample mean for class k
  mu <- rowMeans(data_train[, which(labels_train == k)])
  mu_matrix <- matrix(mu, nrow = nrow(data_test), ncol = ncol(data_test), byrow =
  ↪ FALSE)
  # x - 0.5*\mu
  adjusted_data <- data_test - 0.5 * mu_matrix
  # Linear Discriminant function
  lda_score <- diag(t(adjusted_data) %*% theta_est %*% mu_matrix + log(prior))
  rho[, k] <- lda_score
}
predicted <- apply(rho, 1, which.max)
```

Notice that, `prior` is a scalar value, `mu` is a one dimensional array of length $p$, and `mu_matrix` is an array of size $p$ by $n$ which contains repeated columns of `mu`. In addition, `rho` is an array containing the LDA scores for all samples for each class. For comparative purposes, this computation will be performed twice: once for each estimate of the precision matrix. Further details on the parameters used and the results attained in this experiment see [3].

# References


[1] BOLLHÖFER, M., EFTEKHARI, A., SCHEIDEGGER, S., AND SCHENK, O. Large-scale Sparse Inverse Covariance Matrix Estimation. *SIAM Journal on Scientific Computing 41*, 1 (2019), A380–A401. URL: https://doi.org/10.1137/17M1147615.

[2] EFTEKHARI, A., BOLLHÖFER, M., AND SCHENK, O. Distributed Memory Sparse Inverse Covariance Matrix Estimation on High-Performance Computing Architectures. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis* (2018), SC '18, IEEE Press. URL: http://doi.org/10.1109/SC.2018.00023.

[3] EFTEKHARI, A., GAEDKE-MERZHÄUSER, L., PASADAKIS, D., BOLLHÖFER, M., SCHEIDEGGER, S., AND SCHENK, O. Sparse precision matrix estimation with SQUIC. *Under Review* (2023).

[4] EFTEKHARI, A., PASADAKIS, D., BOLLHÖFER, M., SCHEIDEGGER, S., AND SCHENK, O. Block-enhanced precision matrix estimation for large-scale datasets. *Journal of Computational Science* (2021), 101389. URL: https://doi.org/10.1016/j.jocs.2021.101389.

[5] HSIEH, C.-J., DHILLON, I., RAVIKUMAR, P., AND SUSTIK, M. Sparse inverse covariance matrix estimation using quadratic approximation. In *Advances in Neural Information Processing Systems* (2011), J. Shawe-Taylor, R. Zemel, P. Bartlett, F. Pereira, and K. Q. Weinberger, Eds., vol. 24, Curran Associates, Inc. URL: https://proceedings.neurips.cc/paper/2011/hash/2ba8698b79439589fdd2b0f7218d8b07-Abstract.html.