Regularized Precision Matrix Estimation via ADMM

Matt Galloway

May 30, 2018

Abstract

`ADMMsigma` is an R package that estimates a penalized precision matrix via the alternating direction method of multipliers (ADMM) algorithm. It currently supports a general elastic-net penalty that allows for both ridge and lasso-type penalties as special cases. This report will provide a brief overview of the algorithm, discuss the functions contained in `ADMMsigma`, and provide simulation results.

Contents

1 Introduction

Suppose we want to solve the following optimization problem:

$$\begin{aligned} &\text{minimize } f(x) + g(z) \\ &\text{subject to } Ax + Bz = c \end{aligned}$$

where $x \in \mathbb{R}^n, z \in \mathbb{R}^m, A \in \mathbb{R}^{p\times n}, B \in \mathbb{R}^{p\times m}$, and $c \in \mathbb{R}^p$. Following Boyd et al. (2011), the optimization problem will be introduced in vector form though we will later consider cases where x and z are matrices. We will also assume f and g are convex functions. Optimization problems like this arise naturally in several statistics and machine learning applications – particularly regularization methods. For instance, we could take f to be the squared error loss, g to be the l_2-norm, c to be equal to zero and A and B to be identity matrices to solve the ridge regression optimization problem.

The *augmented lagrangian* is constructed as follows:

$$L_\rho(x, z, y) = f(x) + g(z) + y^T(Ax + Bz - c) + \frac{\rho}{2}\left\|Ax + Bz - c\right\|_2^2$$

where $y \in \mathbb{R}^p$ is the lagrange multiplier and $\rho > 0$ is a scalar. Clearly any minimizer, p^*, under the augmented lagrangian is equivalent to that of the lagrangian since any feasible point (x, z) satisfies the constraint $\rho \left\| Ax + Bz - c \right\|_2^2 / 2 = 0$.

$$p^* = inf \left\{ f(x) + g(z) | Ax + Bz = c \right\}$$

The alternating direction method of multipliers (ADMM) algorithm consists of the following repeated iterations:

$$x^{k+1} := \arg\min_x L_\rho(x, z^k, y^k) \tag{1}$$
$$z^{k+1} := \arg\min_z L_\rho(x^{k+1}, z, y^k) \tag{2}$$
$$y^{k+1} := y^k + \rho(Ax^{k+1} + Bz^{k+1} - c) \tag{3}$$

A more complete introduction to the algorithm – specifically how it arose out of *dual ascent* and *method of multipliers* – can be found in Boyd et al. (2011).

2 Regularized Precision Matrix Estimation

Now consider the case where $X_1, ..., X_n$ are iid $N_p(\mu, \Sigma)$ random variables and we are tasked with estimating the precision matrix, denoted $\Omega \equiv \Sigma^{-1}$. The maximum likelihood estimator for Ω is

$$\hat{\Omega}_{MLE} = \arg\min_{\Omega \in S_+^p} \left\{ Tr\left(S\Omega\right) - \log\det\left(\Omega\right) \right\}$$

where $S = \sum_{i=1}^n (X_i - \bar{X})(X_i - \bar{X})^T / n$ and $\bar{X}$ is the sample mean. By setting the gradient equal to zero, we can show that when the solution exists, $\hat{\Omega}_{MLE} = S^{-1}$.

As in regression settings, we can construct a *penalized* likelihood estimator by adding a penalty term, $P\left(\Omega\right)$, to the likelihood:

$$\hat{\Omega} = \arg\min_{\Omega \in S_+^p} \left\{ Tr\left(S\Omega\right) - \log\det\left(\Omega\right) + P\left(\Omega\right) \right\}$$

$P\left(\Omega\right)$ is often of the form $P\left(\Omega\right) = \lambda\|\Omega\|_F^2$ or $P\left(\Omega\right) = \|\Omega\|_1$ where $\lambda > 0$, $\left\|\cdot\right\|_F^2$ is the Frobenius norm and we define $\left\|A\right\|_1 = \sum_{i,j} \left|A_{ij}\right|$. These penalties are the ridge and lasso, respectively. We will, instead, take $P\left(\Omega\right) = \lambda\left[\frac{1-\alpha}{2}\left\|\Omega\right|_F^2 + \alpha\left\|\Omega\right\|_1\right]$ so that the full penalized likelihood is

$$\hat{\Omega} = \arg\min_{\Omega \in S_+^p} \left\{ Tr\left(S\Omega\right) - \log\det\left(\Omega\right) + \lambda\left[\frac{1-\alpha}{2}\left\|\Omega\right|_F^2 + \alpha\left\|\Omega\right\|_1\right] \right\}$$

where $0 \leq \alpha \leq 1$. This *elastic-net* penalty was explored by Hui Zou and Trevor Hastie (Zou and Hastie 2005) and is identical to the penalty used in the popular penalized regression package `glmnet`. Clearly, when $\alpha = 0$ the elastic-net reduces to a ridge-type penalty and when $\alpha = 1$ it reduces to a lasso-type penalty.

By letting f be equal to the non-penalized likelihood and g equal to $P\left(\Omega\right)$, our goal is to minimize the full augmented lagrangian where the constraint is that $\Omega - Z$ is equal to zero:

$$L_\rho(\Omega, Z, \Lambda) = f\left(\Omega\right) + g\left(Z\right) + Tr\left[\Lambda\left(\Omega - Z\right)\right] + \frac{\rho}{2}\left\|\Omega - Z\right\|_F^2$$

The ADMM algorithm for estimating the penalized precision matrix in this problem is

$$\Omega^{k+1} = \arg\min_{\Omega}\left\{Tr\left(S\Omega\right) - \log\det\left(\Omega\right) + Tr\left[\Lambda^k\left(\Omega - Z^k\right)\right] + \frac{\rho}{2}\left\|\Omega - Z^k\right\|_F^2\right\} \tag{4}$$

$$Z^{k+1} = \arg\min_{Z}\left\{\lambda\left[\frac{1-\alpha}{2}\left\|Z\right\|_F^2 + \alpha\left\|Z\right\|_1\right] + Tr\left[\Lambda^k\left(\Omega^{k+1} - Z\right)\right] + \frac{\rho}{2}\left\|\Omega^{k+1} - Z\right\|_F^2\right\} \tag{5}$$

$$\Lambda^{k+1} = \Lambda^k + \rho\left(\Omega^{k+1} - Z^{k+1}\right) \tag{6}$$

2.1 Scaled-Form ADMM

An alternate form of the ADMM algorithm can constructed by scaling the dual variable (Λ^k). Let us define $R^k = \Omega - Z^k$ and $U^k = \Lambda^k/\rho$.

$$\begin{aligned} Tr\left[\Lambda^k\left(\Omega - Z^k\right)\right] + \frac{\rho}{2}\left\|\Omega - Z^k\right\|_F^2 &= Tr\left[\Lambda^k R^k\right] + \frac{\rho}{2}\left\|R^k\right\|_F^2 \\ &= \frac{\rho}{2}\left\|R^k + \Lambda^k/\rho\right\|_F^2 - \frac{\rho}{2}\left\|\Lambda^k/\rho\right\|_F^2 \\ &= \frac{\rho}{2}\left\|R^k + U^k\right\|_F^2 - \frac{\rho}{2}\left\|U^k\right\|_F^2 \end{aligned}$$

Therefore, the condensed-form ADMM algorithm can now be written as

$$\Omega^{k+1} = \arg\min_{\Omega}\left\{Tr\left(S\Omega\right) - \log\det\left(\Omega\right) + \frac{\rho}{2}\left\|\Omega - Z^k + U^k\right\|_F^2\right\} \tag{7}$$

$$Z^{k+1} = \arg\min_{Z}\left\{\lambda\left[\frac{1-\alpha}{2}\left\|Z\right\|_F^2 + \alpha\left\|Z\right\|_1\right] + \frac{\rho}{2}\left\|\Omega^{k+1} - Z + U^k\right\|_F^2\right\} \tag{8}$$

$$U^{k+1} = U^k + \Omega^{k+1} - Z^{k+1} \tag{9}$$

And more generally (in vector form),

$$x^{k+1} = \arg\min_{x}\left\{f(x) + \frac{\rho}{2}\left\|Ax + Bz^k - c + u^k\right\|_2^2\right\} \tag{10}$$

$$z^{k+1} = \arg\min_{z}\left\{g(z) + \frac{\rho}{2}\left\|Ax^{k+1} + Bz - c + u^k\right\|_2^2\right\} \tag{11}$$

$$u^{k+1} = u^k + Ax^{k+1} + Bz^{k+1} - c \tag{12}$$

Note that there are limitations to using this method. Because the dual variable is scaled by ρ (the step size), this form limits one to using a constant step size without making further adjustments to U^k. It has been shown in the literature that a dynamic step size can significantly reduce the number of iterations required for convergence.

2.2 Algorithm

$$\begin{aligned}
\Omega^{k+1} &= \arg\min_{\Omega} \left\{ Tr\left(S\Omega\right) - \log\det\left(\Omega\right) + Tr\left[\Lambda^k\left(\Omega - Z^k\right)\right] + \frac{\rho}{2}\left\|\Omega - Z^k\right\|_F^2 \right\} \\
Z^{k+1} &= \arg\min_{Z} \left\{ \lambda\left[\frac{1-\alpha}{2}\left\|Z\right\|_F^2 + \alpha\left\|Z\right\|_1\right] + Tr\left[\Lambda^k\left(\Omega^{k+1} - Z\right)\right] + \frac{\rho}{2}\left\|\Omega^{k+1} - Z\right\|_F^2 \right\} \\
\Lambda^{k+1} &= \Lambda^k + \rho\left(\Omega^{k+1} - Z^{k+1}\right)
\end{aligned}$$

Initialize Z^0, Λ^0, and ρ. Iterate the following three steps until convergence:

1. Decompose $S + \Lambda^k - \rho Z^k = VQV^T$ via spectral decomposition.

$$\Omega^{k+1} = \frac{1}{2\rho}V\left[-Q + \left(Q^2 + 4\rho I_p\right)^{1/2}\right]V^T$$

2. Elementwise soft-thresholding for all $i = 1, ..., p$ and $j = 1, ..., p$.

$$\begin{aligned}
Z_{ij}^{k+1} &= \frac{1}{\lambda(1-\alpha) + \rho} Sign\left(\rho\Omega_{ij}^{k+1} + \Lambda_{ij}^k\right)\left(\left|\rho\Omega_{ij}^{k+1} + \Lambda_{ij}^k\right| - \lambda\alpha\right)_+ \\
&= \frac{1}{\lambda(1-\alpha) + \rho} Soft\left(\rho\Omega_{ij}^{k+1} + \Lambda_{ij}^k, \lambda\alpha\right)
\end{aligned}$$

3. Update Λ^{k+1}.

$$\Lambda^{k+1} = \Lambda^k + \rho\left(\Omega^{k+1} - Z^{k+1}\right)$$

2.2.1 Proof of (1):

$$\Omega^{k+1} = \arg\min_{\Omega} \left\{ Tr\left(S\Omega\right) - \log\det\left(\Omega\right) + Tr\left[\Lambda^k\left(\Omega - Z^k\right)\right] + \frac{\rho}{2}\left\|\Omega - Z^k\right\|_F^2 \right\}$$

$$\begin{aligned}
&\nabla_{\Omega} \left\{ Tr\left(S\Omega\right) - \log\det\left(\Omega\right) + Tr\left[\Lambda^k\left(\Omega - Z^k\right)\right] + \frac{\rho}{2}\left\|\Omega - Z^k\right\|_F^2 \right\} \\
&\quad = S - \Omega^{-1} + \Lambda^k + \rho\left(\Omega - Z^k\right)
\end{aligned}$$

Set the gradient equal to zero and decompose $\Omega = VDV^T$ where D is a diagonal matrix with diagonal elements equal to the eigen values of Ω and V is the matrix with corresponding eigen vectors as columns.

$$S + \Lambda^k - \rho Z^k = \Omega^{-1} - \rho\Omega = VD^{-1}V^T - \rho VDV^T = V\left(D^{-1} - \rho D\right)V^T$$

This equivalence implies that

$$\phi_j\left(S + \Lambda^k - \rho Z^k\right) = \frac{1}{\phi_j(\Omega^{k+1})} - \rho\phi_j(\Omega^{k+1})$$

where $\phi_j(\cdot)$ is the jth eigen value.

$$\Rightarrow \rho\phi_j^2(\Omega^{k+1}) + \phi_j\left(S + \Lambda^k - \rho Z^k\right)\phi_j(\Omega^{k+1}) - 1 = 0$$
$$\Rightarrow \phi_j(\Omega^{k+1}) = \frac{-\phi_j(S + \Lambda^k - \rho Z^k) \pm \sqrt{\phi_j^2(S + \Lambda^k - \rho Z^k) + 4\rho}}{2\rho}$$

In summary, if we decompose $S + \Lambda^k - \rho Z^k = VQV^T$ then

$$\Omega^{k+1} = \frac{1}{2\rho} V\left[-Q + (Q^2 + 4\rho I_p)^{1/2}\right]V^T$$

2.2.2 Proof of (2)

$$Z^{k+1} = \arg\min_Z \left\{\lambda\left[\frac{1-\alpha}{2}\|Z\|_F^2 + \alpha\|Z\|_1\right] + Tr\left[\Lambda^k\left(\Omega^{k+1} - Z\right)\right] + \frac{\rho}{2}\left\|\Omega^{k+1} - Z\right\|_F^2\right\}$$

$$\begin{aligned}
&\partial\left\{\lambda\left[\frac{1-\alpha}{2}\|Z\|_F^2 + \alpha\|Z\|_1\right] + Tr\left[\Lambda^k\left(\Omega^{k+1} - Z\right)\right] + \frac{\rho}{2}\left\|\Omega^{k+1} - Z\right\|_F^2\right\} \\
&\quad = \partial\left\{\lambda\left[\frac{1-\alpha}{2}\|Z\|_F^2 + \alpha\|Z\|_1\right] + Tr\left[\Lambda^k\left(\Omega^{k+1} - Z\right)\right]\right\} + \nabla_\Omega\left\{\frac{\rho}{2}\left\|\Omega^{k+1} - Z\right\|_F^2\right\} \\
&\quad = \lambda(1-\alpha)Z + Sign(Z)\lambda\alpha - \Lambda^k - \rho\left(\Omega^{k+1} - Z\right)
\end{aligned}$$

where $Sign(Z)$ is the elementwise Sign operator. By setting the gradient/sub-differential equal to zero, we arrive at the following equivalence:

$$Z_{ij} = \frac{1}{\lambda(1-\alpha) + \rho}\left(\rho\Omega_{ij}^{k+1} + \Lambda_{ij}^k - Sign(Z_{ij})\lambda\alpha\right)$$

for all $i = 1, ..., p$ and $j = 1, ..., p$. We observe two scenarios:

- If $Z_{ij} > 0$ then

$$\rho\Omega_{ij}^{k+1} + \Lambda_{ij}^k > \lambda\alpha$$

- If $Z_{ij} < 0$ then

$$\rho\Omega_{ij}^{k+1} + \Lambda_{ij}^k < -\lambda\alpha$$

This implies that $Sign(Z_{ij}) = Sign(\rho\Omega_{ij}^{k+1} + \Lambda_{ij}^k)$. Putting all the pieces together, we arrive at

$$\begin{aligned}
Z_{ij}^{k+1} &= \frac{1}{\lambda(1-\alpha) + \rho} Sign\left(\rho\Omega_{ij}^{k+1} + \Lambda_{ij}^k\right)\left(\left|\rho\Omega_{ij}^{k+1} + \Lambda_{ij}^k\right| - \lambda\alpha\right)_+ \\
&= \frac{1}{\lambda(1-\alpha) + \rho} Soft\left(\rho\Omega_{ij}^{k+1} + \Lambda_{ij}^k, \lambda\alpha\right)
\end{aligned}$$

where $Soft$ is the soft-thresholding function.

3 ADMMsigma R Package

3.1 Installation

```
# The easiest way to install is from CRAN
install.packages("ADMMsigma")

# You can also install the development
# version from GitHub:
# install.packages('devtools')
devtools::install_github("MGallow/ADMMsigma")
```

If there are any issues/bugs, please let me know: github. You can also contact me via my website. Pull requests are welcome!

A (possibly incomplete) list of functions contained in the package can be found below:

- `ADMMsigma()` computes the estimated precision matrix (ridge, lasso, and elastic-net type regularization optional)
- `RIDGEsigma()` computes the estimated ridge penalized precision matrix via closed-form solution
- `plot.ADMMsigma()` produces a heat map or line graph for cross validation errors
- `plot.RIDGEsigma()` produces a heat map or line graph for cross validation errors

3.2 Usage

We will first generate data from a sparse, tri-diagonal precision matrix and denote it as Omega.

```
library(ADMMsigma)

# generate data from a sparse matrix
# first compute covariance matrix
S = matrix(0.7, nrow = 5, ncol = 5)
for (i in 1:5) {
    for (j in 1:5) {
        S[i, j] = S[i, j]^abs(i - j)
    }
}

# print oracle precision matrix
# (shrinkage might be useful)
(Omega = qr.solve(S) %>% round(3))
```

```
##        [,1]   [,2]   [,3]   [,4]   [,5]
## [1,]  1.961 -1.373  0.000  0.000  0.000
## [2,] -1.373  2.922 -1.373  0.000  0.000
## [3,]  0.000 -1.373  2.922 -1.373  0.000
## [4,]  0.000  0.000 -1.373  2.922 -1.373
## [5,]  0.000  0.000  0.000 -1.373  1.961
```

```
# generate 1000 x 5 matrix with rows
# drawn from iid N_p(0, S)
Z = matrix(rnorm(100 * 5), nrow = 100, ncol = 5)
out = eigen(S, symmetric = TRUE)
S.sqrt = out$vectors %*% diag(out$values^0.5) %*%
    t(out$vectors)
X = Z %*% S.sqrt

# snap shot of data
head(X)
```

```
##             [,1]        [,2]       [,3]       [,4]      [,5]
## [1,]  1.21422314  0.64350108  1.4497582  0.9225252  1.440199
## [2,]  0.58377250  0.42689162  1.3776194  1.5520929  1.180953
## [3,]  0.22860143  0.06369156 -1.2704378 -1.5638194 -1.857888
## [4,] -0.07623948 -0.82504870 -1.8924654 -1.0037209 -1.248488
## [5,]  1.72792718 -0.20342253  0.9573457  2.2965954  1.174462
## [6,] -0.39009415 -0.87759484  0.2451972  0.8410061  1.599513
```

As described earlier in the report, the maximum likelihood estimator (MLE) for Omega is the inverse of the sample precision matrix $S^{-1} = \left[\sum_{i=1}^{n}(X_i - \bar{X})(X_i - \bar{X})^T/n\right]^{-1}$:

```
# print inverse of sample precision
# matrix (perhaps a bad estimate)
(qr.solve(cov(X) * (nrow(X) - 1)/nrow(X)) %>%
    round(5))
```

```
##          [,1]     [,2]     [,3]     [,4]     [,5]
## [1,]  2.00191 -1.41613  0.09304 -0.08652  0.06685
## [2,] -1.41613  2.98375 -1.39224 -0.00180 -0.02669
## [3,]  0.09304 -1.39224  2.93760 -1.57340  0.13505
## [4,] -0.08652 -0.00180 -1.57340  3.29131 -1.50627
## [5,]  0.06685 -0.02669  0.13505 -1.50627  1.95345
```

However, because Omega (known as the *oracle*) is sparse, a shrinkage estimator will perhaps perform better than the sample estimator. Below we construct various penalized estimators:

```
# elastic-net type penalty (set tolerance
# to 1e-8)
ADMMsigma(X, tol.abs = 1e-08, tol.rel = 1e-08)
```

```
##
## Call: ADMMsigma(X = X, tol.abs = 1e-08, tol.rel = 1e-08)
##
## Iterations: 172
##
## Tuning parameters:
##       log10(lam)  alpha
## [1,]      -1.587      1
```

```
##
## Log-likelihood: -110.23594
##
## Omega:
##           [,1]     [,2]     [,3]     [,4]     [,5]
## [1,]  1.74916 -1.21729 -0.11167  0.00000  0.00000
## [2,] -1.21729  2.99130 -1.55691  0.00000  0.00000
## [3,] -0.11167 -1.55691  3.19597 -1.27625 -0.05198
## [4,]  0.00000  0.00000 -1.27625  2.65715 -0.99712
## [5,]  0.00000  0.00000 -0.05198 -0.99712  1.82133
```

LASSO:

```
# lasso penalty (default tolerance)
ADMMsigma(X, alpha = 1)
```

```
##
## Iterations:
## [1] 16
##
## Tuning parameters:
##       log10(lam)  alpha
## [1,]          -3      1
##
## Omega:
##           [,1]     [,2]     [,3]     [,4]     [,5]
## [1,]  1.98897 -1.39520  0.07501 -0.07195  0.05755
## [2,] -1.39520  2.94945 -1.36518 -0.02047 -0.01395
## [3,]  0.07501 -1.36518  2.90753 -1.54178  0.11625
## [4,] -0.07195 -0.02047 -1.54178  3.24849 -1.48219
## [5,]  0.05755 -0.01395  0.11625 -1.48219  1.93985
```

ELASTIC-NET:

```
# elastic-net penalty (alpha = 0.5)
ADMMsigma(X, alpha = 0.5)
```

```
##
## Iterations:
## [1] 16
##
## Tuning parameters:
##       log10(lam)  alpha
## [1,]          -3    0.5
##
## Omega:
##           [,1]     [,2]     [,3]     [,4]     [,5]
## [1,]  1.98401 -1.38850  0.07368 -0.07468  0.06041
## [2,] -1.38850  2.93561 -1.35372 -0.02388 -0.01640
## [3,]  0.07368 -1.35372  2.88711 -1.52619  0.11302
## [4,] -0.07468 -0.02388 -1.52619  3.22819 -1.47294
## [5,]  0.06041 -0.01640  0.11302 -1.47294  1.93396
```

RIDGE:

```
# ridge penalty
ADMMsigma(X, alpha = 0)
```

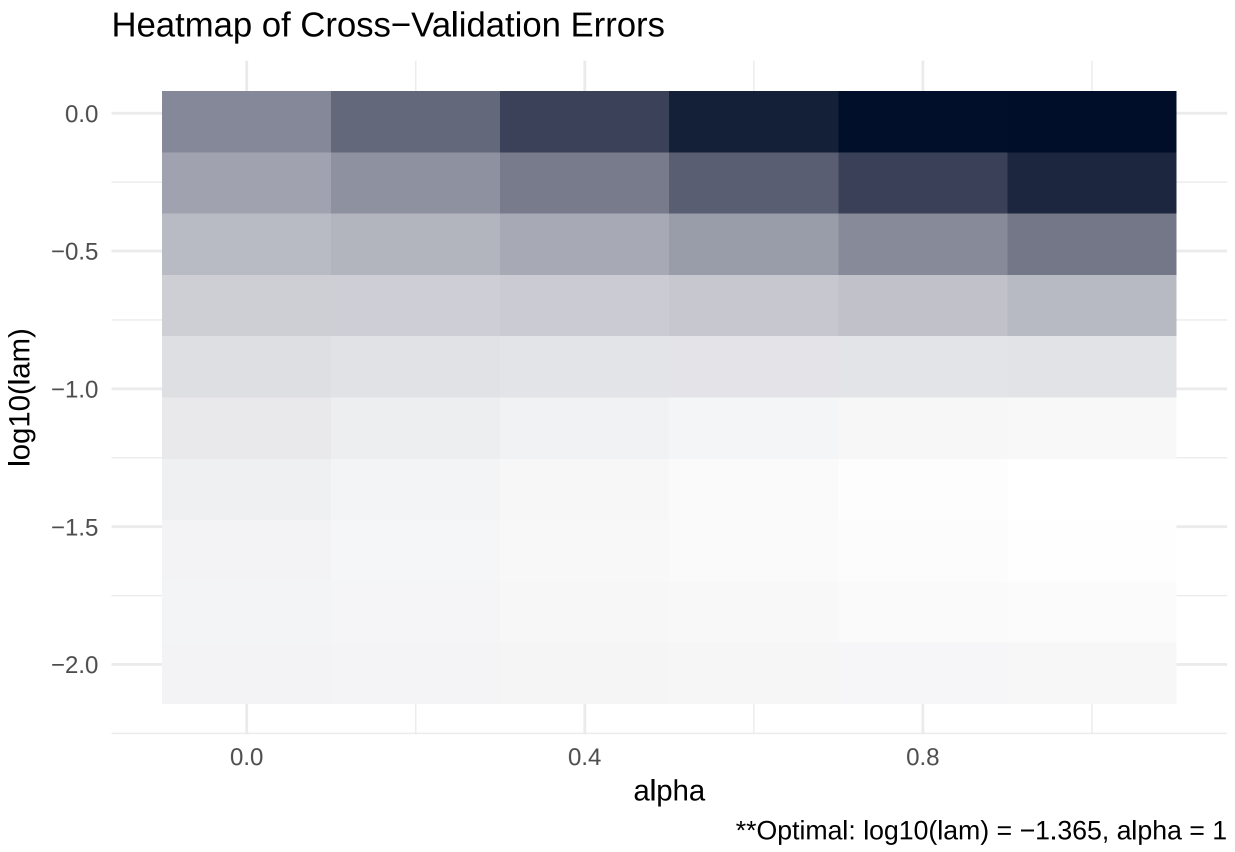
```
##
## Iterations:
## [1] 16
##
## Tuning parameters:
##       log10(lam)  alpha
## [1,]          -3      0
##
## Omega:
##           [,1]     [,2]     [,3]     [,4]     [,5]
## [1,]  1.97957 -1.38283  0.07351 -0.07837  0.06365
## [2,] -1.38283  2.92392 -1.34491 -0.02497 -0.01979
## [3,]  0.07351 -1.34491  2.87050 -1.51430  0.11142
## [4,] -0.07837 -0.02497 -1.51430  3.21177 -1.46548
## [5,]  0.06365 -0.01979  0.11142 -1.46548  1.92892
```

```
# ridge penalty no ADMM
RIDGEsigma(X, lam = 10^seq(-8, 8, 0.01))
```

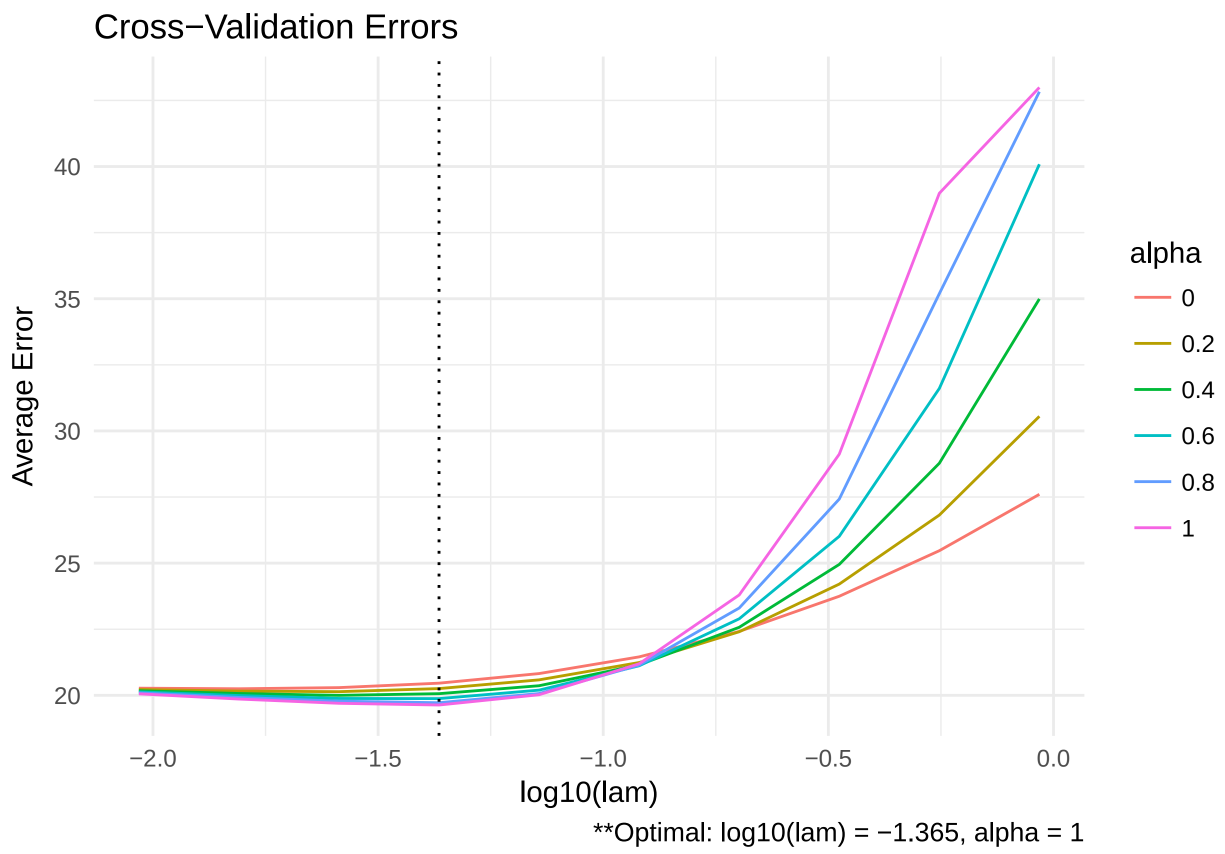
```
##
## Tuning parameter:
##       log10(lam)    lam
## [1,]       -2.82  0.002
##
## Omega:
##           [,1]     [,2]     [,3]     [,4]     [,5]
## [1,]  1.96954 -1.36852  0.06659 -0.07680  0.06314
## [2,] -1.36852  2.89875 -1.32718 -0.03111 -0.01877
## [3,]  0.06659 -1.32718  2.84508 -1.49281  0.10334
## [4,] -0.07680 -0.03111 -1.49281  3.18037 -1.44902
## [5,]  0.06314 -0.01877  0.10334 -1.44902  1.91851
```

This package also has the capability to provide heat maps for the cross validation errors. The more bright (white) areas of the heat map pertain to more optimal tuning parameters.

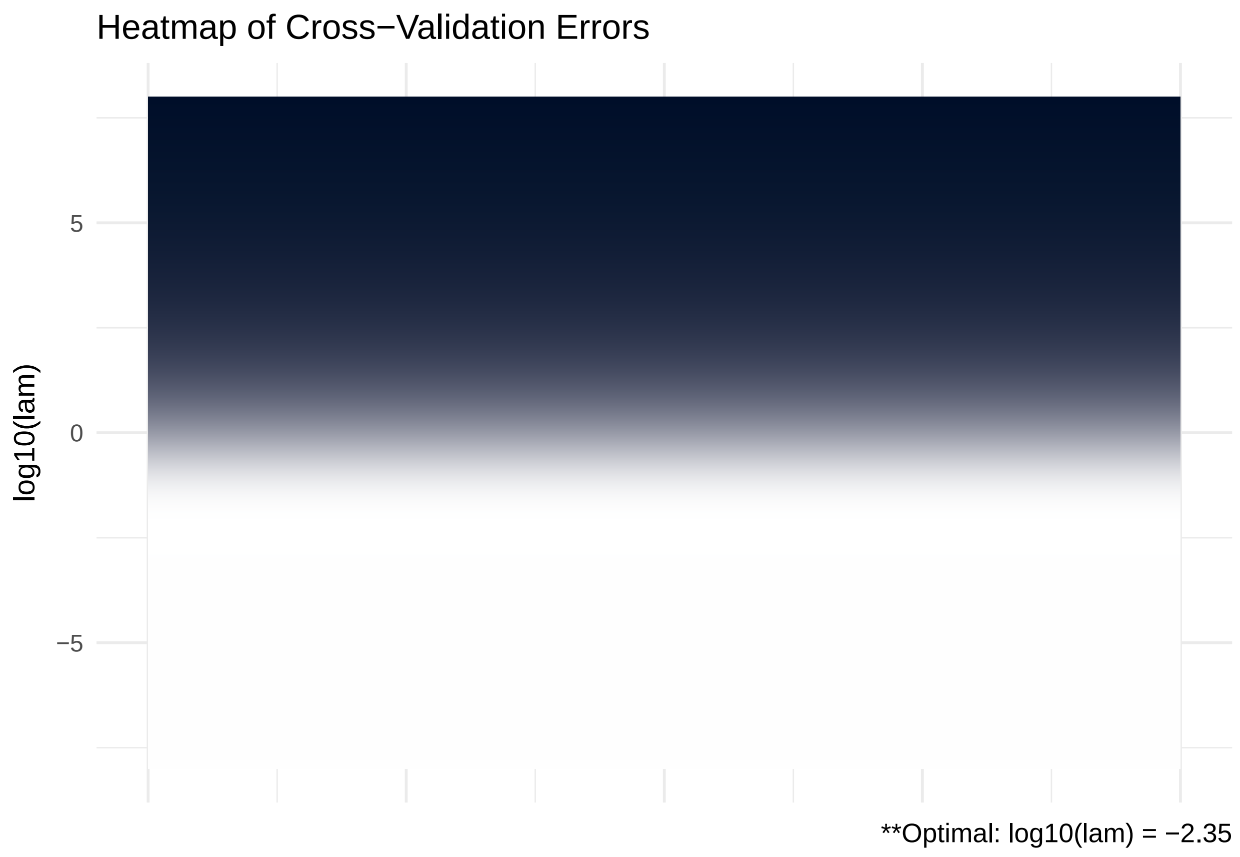
```r
# produce CV heat map for ADMMsigma
ADMM = ADMMsigma(X, tol.abs = 1e-08, tol.rel = 1e-08)
ADMM %>% plot(type = "heatmap")
```



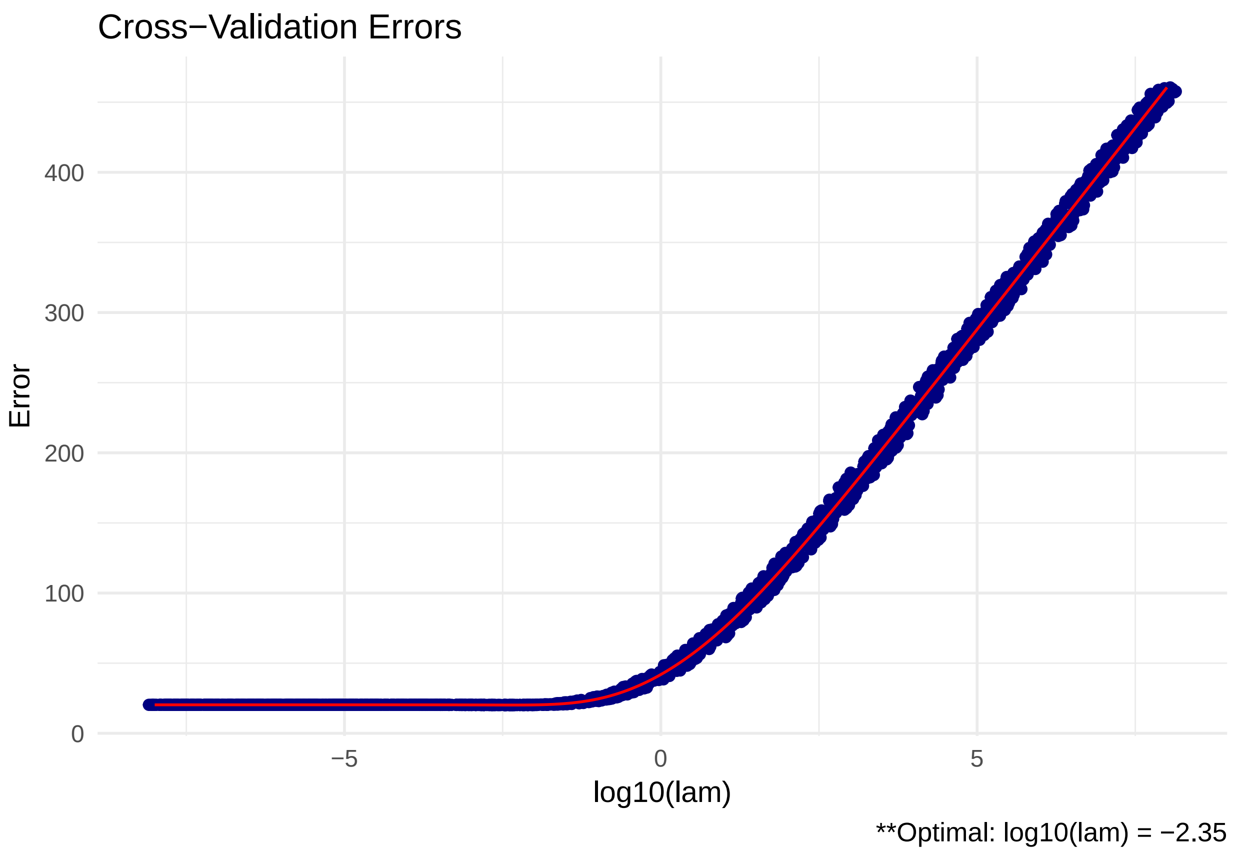
```
# produce line graph for CV errors for
# ADMMsigma
ADMM %>% plot(type = "line")
```



```r
# produce CV heat map for RIDGEsigma
RIDGE = RIDGEsigma(X, lam = 10^seq(-8, 8,
    0.01))
RIDGE %>% plot(type = "heatmap")
```



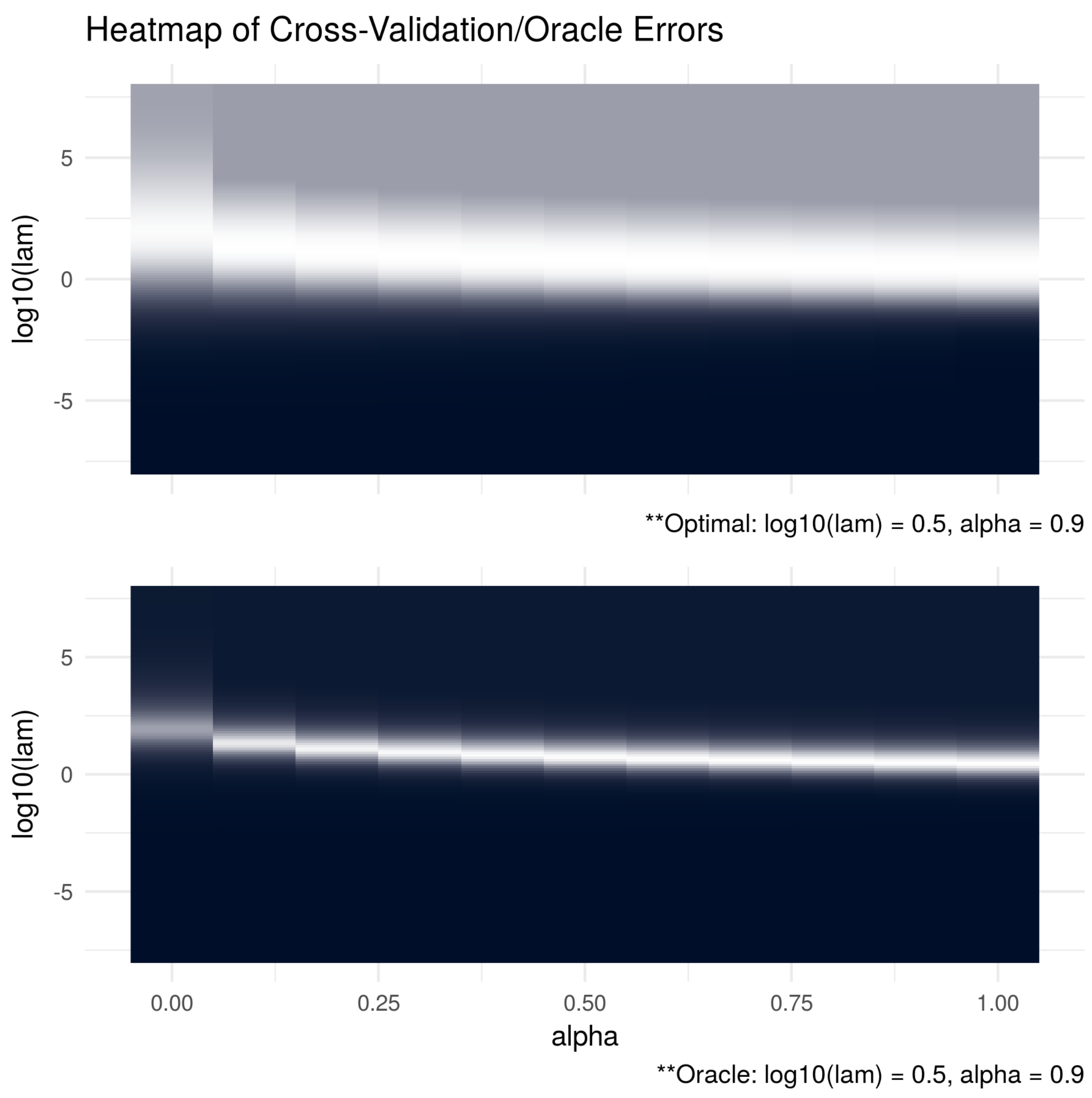
```
# produce line graph for CV errors for
# RIDGEsigma
RIDGE %>% plot(type = "line")
```



3.3 Simulations

In the simulations below we generated data from a number of oracle precision matrices with various structures. For each data-generating procedure, the `ADMMsigma()` function was run using 5-fold cross validation. After 20 replications, the cross validation errors were totalled and the optimal tuning parameters were selected (results in the top figure). These results are compared with the Kullback Leibler (KL) losses between the estimates and the oracle precision matrix (bottom figure). We can see below that our cross validation procedure choosing tuning parameters close to the optimal parameters.

3.3.1 Compound Symmetric: P = 100, N = 50

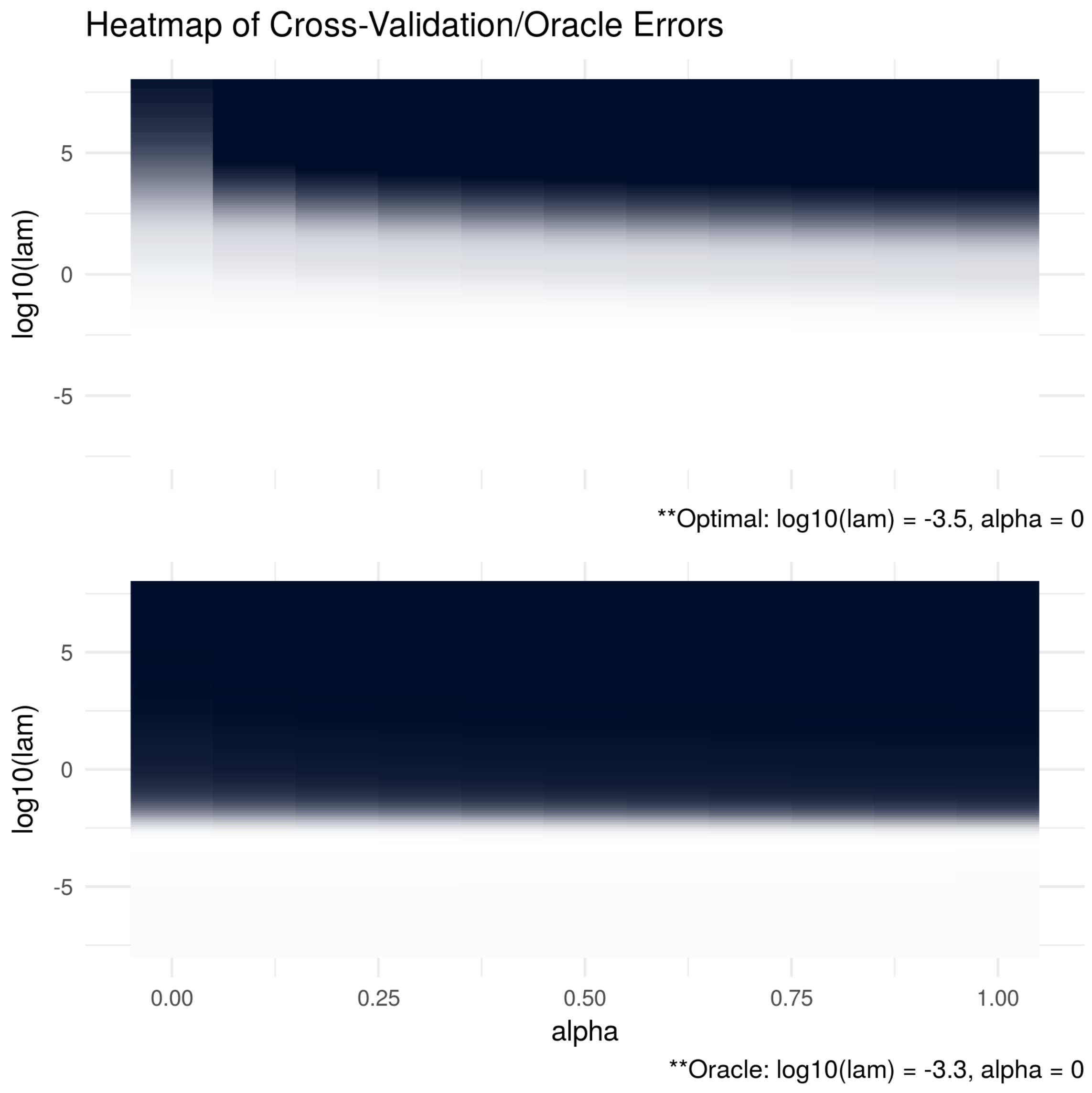


```r
# oracle precision matrix
Omega = matrix(0.9, ncol = 100, nrow = 100)
diag(Omega = 1)

# generate covariance matrix
S = qr.solve(Omega)

# generate data
Z = matrix(rnorm(100 * 50), nrow = 50, ncol = 100)
out = eigen(S, symmetric = TRUE)
S.sqrt = out$vectors %*% diag(out$values^0.5) %*%
    t(out$vectors)
X = Z %*% S.sqrt
```

3.3.2 Compound Symmetric: P = 10, N = 1000

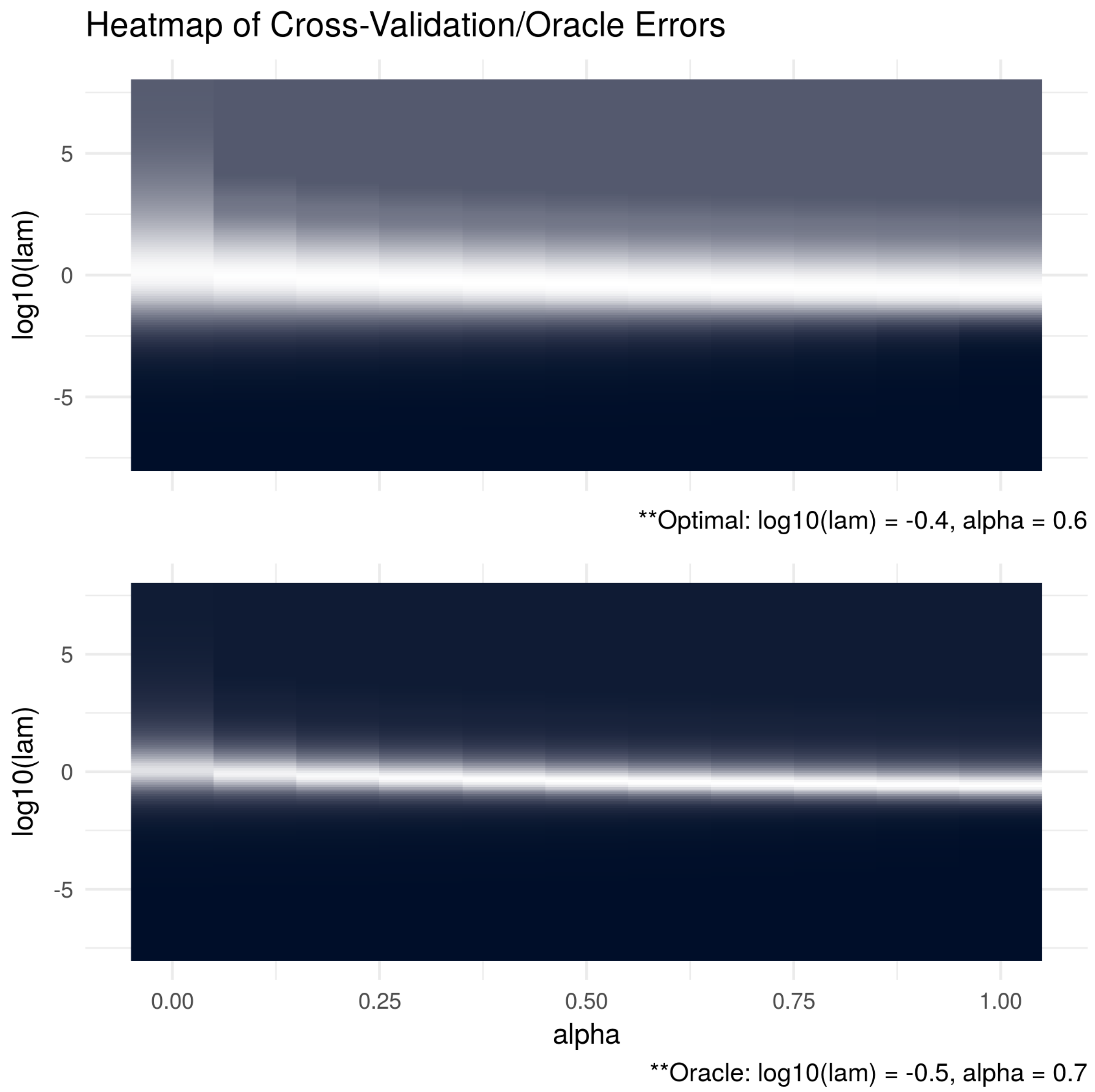


```
# oracle precision matrix
Omega = matrix(0.9, ncol = 10, nrow = 10)
diag(Omega = 1)

# generate covariance matrix
S = qr.solve(Omega)

# generate data
Z = matrix(rnorm(10 * 1000), nrow = 1000,
    ncol = 10)
out = eigen(S, symmetric = TRUE)
S.sqrt = out$vectors %*% diag(out$values^0.5) %*%
    t(out$vectors)
X = Z %*% S.sqrt
```

3.3.3 Dense: $P = 100$, $N = 50$



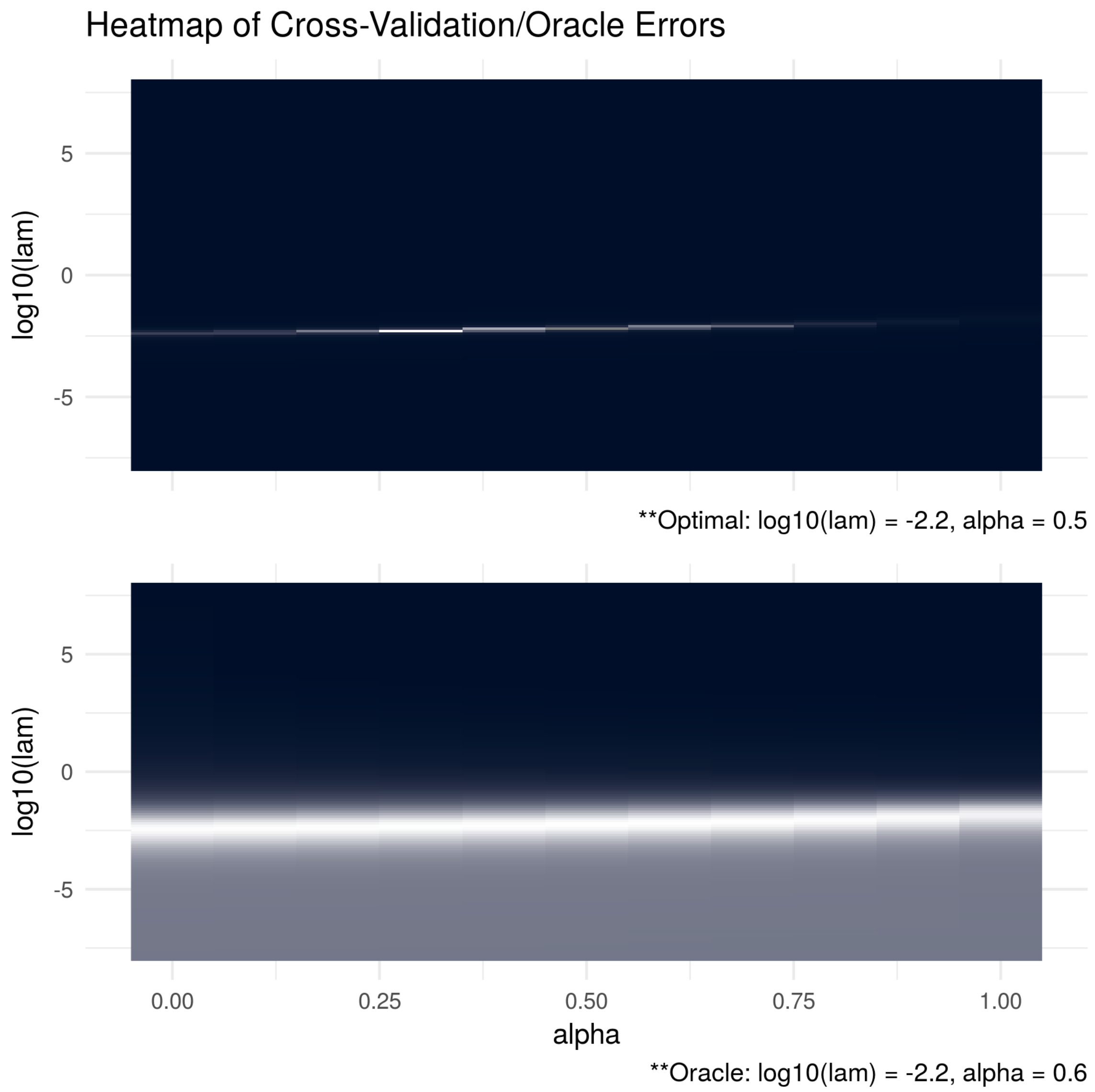
```
# generate eigen values
eigen = c(rep(1000, 5, rep(1, 100 - 5)))

# randomly generate orthogonal basis (via QR)
Q = matrix(rnorm(100*100), nrow = 100, ncol = 100) %>% qr %>% qr.Q

# generate covariance matrix
S = Q %*% diag(eigen) %*% t(Q)

# generate data
Z = matrix(rnorm(100*50), nrow = 50, ncol = 100)
out = eigen(S, symmetric = TRUE)
S.sqrt = out$vectors %*% diag(out$values^0.5) %*% t(out$vectors)
X = Z %*% S.sqrt
```

3.3.4 Dense: $P = 10$, $N = 50$



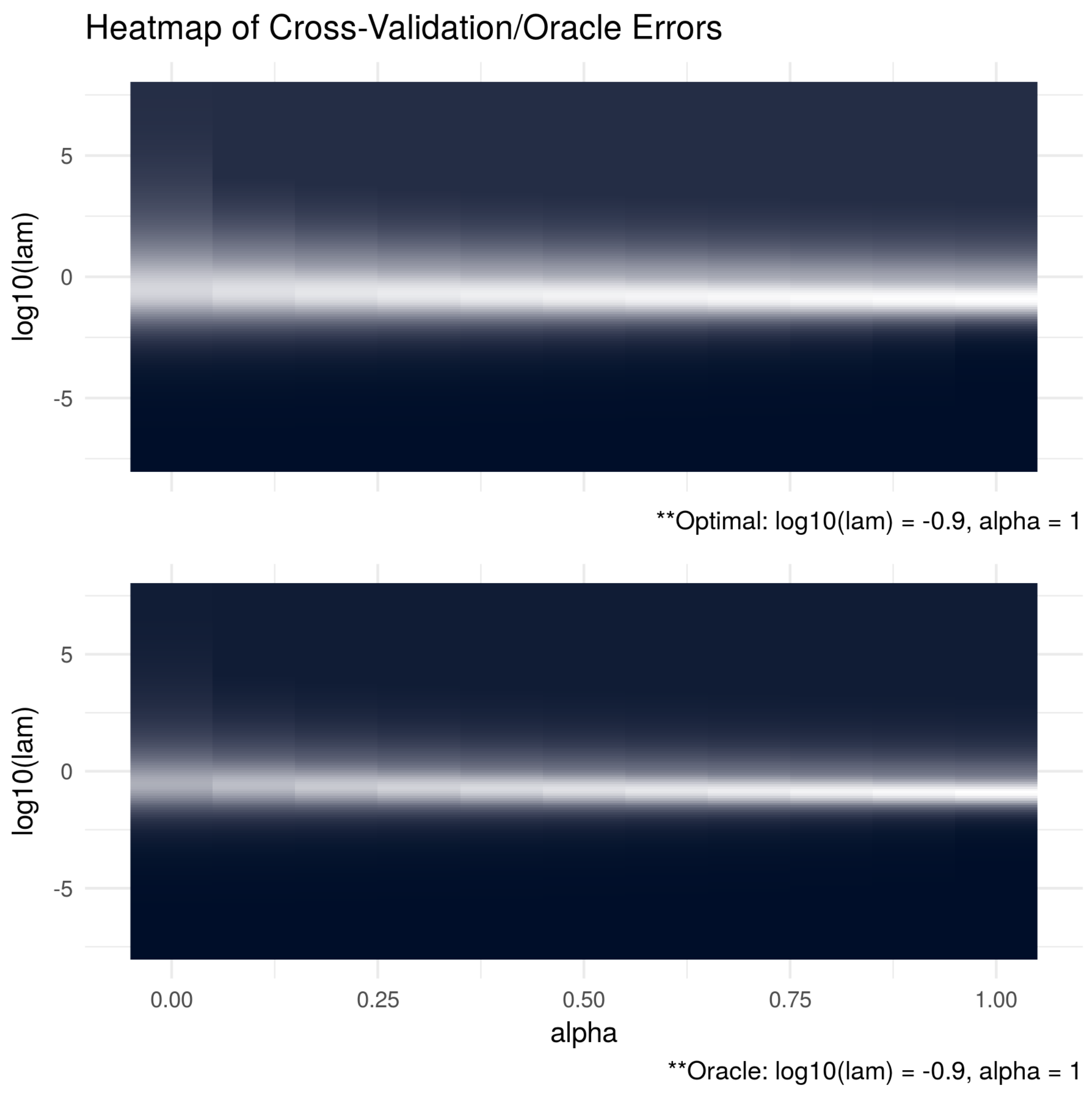
```r
# generate eigen values
eigen = c(rep(1000, 5, rep(1, 10 - 5)))

# randomly generate orthogonal basis (via QR)
Q = matrix(rnorm(10*10), nrow = 10, ncol = 10) %>% qr %>% qr.Q

# generate covariance matrix
S = Q %*% diag(eigen) %*% t(Q)

# generate data
Z = matrix(rnorm(10*50), nrow = 50, ncol = 10)
out = eigen(S, symmetric = TRUE)
S.sqrt = out$vectors %*% diag(out$values^0.5) %*% t(out$vectors)
X = Z %*% S.sqrt
```

3.3.5 Tridiagonal: P = 100, N = 50



```r
# generate covariance matrix
# (can confirm inverse is tri-diagonal)
S = matrix(0, nrow = 100, ncol = 100)
for (i in 1:100){
  for (j in 1:100){
    S[i, j] = 0.7^abs(i - j)
  }
}

# generate data
Z = matrix(rnorm(10*50), nrow = 50, ncol = 10)
out = eigen(S, symmetric = TRUE)
S.sqrt = out$vectors %*% diag(out$values^0.5) %*% t(out$vectors)
X = Z %*% S.sqrt
```

3.4 Benchmark

Below we benchmark the various functions contained in `ADMMsigma`. We can see that `ADMMsigma` (at the default tolerance) offers comparable computation time to the popular `glasso` R package.

3.4.1 Computer Specs:

- MacBook Pro (Late 2016)
- Processor: 2.9 GHz Intel Core i5
- Memory: 8GB 2133 MHz
- Graphics: Intel Iris Graphics 550

```
# generate data from tri-diagonal
# (sparse) matrix compute covariance
# matrix (can confirm inverse is
# tri-diagonal)
S = matrix(0, nrow = 100, ncol = 100)

for (i in 1:100) {
    for (j in 1:100) {
        S[i, j] = 0.7^(abs(i - j))
    }
}

# generate 1000 x 100 matrix with rows
# drawn from iid N_p(0, S)
Z = matrix(rnorm(1000 * 100), nrow = 1000,
    ncol = 100)
out = eigen(S, symmetric = TRUE)
S.sqrt = out$vectors %*% diag(out$values^0.5) %*%
    t(out$vectors)
X = Z %*% S.sqrt


# glasso (for comparison)
microbenchmark(glasso(s = S, rho = 0.1))
```

```
## Unit: milliseconds
##                      expr      min      lq     mean   median      uq
##  glasso(s = S, rho = 0.1) 49.39223 51.1943 55.46338 53.49496 56.3906
##       max neval
##  99.39182   100
```

```
# benchmark ADMMsigma - default tolerance
microbenchmark(ADMMsigma(S = S, lam = 0.1,
    alpha = 1, tol.abs = 1e-04, tol.rel = 1e-04,
    trace = "none"))
```

```
## Unit: milliseconds
##                                                                                            expr
##  ADMMsigma(S = S, lam = 0.1, alpha = 1, tol.abs = 1e-04, tol.rel = 1e-04,      trace = "none")
##       min       lq    mean   median      uq      max neval
##  80.98665 81.81698 84.5567 83.41778 84.4784 115.1402   100
```

```r
# benchmark ADMMsigma - tolerance 1e-8
microbenchmark(ADMMsigma(S = S, lam = 0.1,
    alpha = 1, tol.abs = 1e-08, tol.rel = 1e-08,
    trace = "none"))
```

```
## Unit: milliseconds
##                                                                                                     expr
##  ADMMsigma(S = S, lam = 0.1, alpha = 1, tol.abs = 1e-08, tol.rel = 1e-08,      trace = "none")
##       min       lq     mean   median       uq      max neval
##  257.8969 261.9984 268.8188 265.5823 269.9505 426.4842   100
```

```r
# benchmark ADMMsigma CV - default
# parameter grid
microbenchmark(ADMMsigma(X, trace = "none"),
    times = 5)
```

```
## Unit: seconds
##                          expr      min       lq     mean  median       uq
##  ADMMsigma(X, trace = "none") 7.951453 8.036936 8.044083 8.06635 8.066452
##       max neval
##  8.099225     5
```

```r
# benchmark ADMMsigma parallel CV
microbenchmark(ADMMsigma(X, cores = 3, trace = "none"),
    times = 5)
```

```
## Unit: seconds
##                                     expr      min       lq     mean
##  ADMMsigma(X, cores = 3, trace = "none") 9.145501 9.155234 9.395857
##   median       uq      max neval
##  9.43955 9.468367 9.770631     5
```

```r
# benchmark ADMMsigma CV - likelihood
# convergence criteria
microbenchmark(ADMMsigma(X, crit = "loglik",
    trace = "none"), times = 5)
```

```
## Unit: seconds
##                                           expr      min       lq     mean
##  ADMMsigma(X, crit = "loglik", trace = "none") 6.955905 6.990773 7.002021
##    median       uq      max neval
##  6.993805 7.017449 7.052171     5
```

```r
# benchmark RIDGEsigma CV
microbenchmark(RIDGEsigma(X, lam = 10^seq(-8,
    8, 0.01), trace = "none"), times = 5)
```

```
## Unit: seconds
##                                                      expr      min
##  RIDGEsigma(X, lam = 10^seq(-8, 8, 0.01), trace = "none") 12.01804
##        lq     mean   median       uq      max neval
##  12.02256 12.10764 12.02481 12.16762 12.30518     5
```

References

Boyd, Stephen, Neal Parikh, Eric Chu, Borja Peleato, Jonathan Eckstein, and others. 2011. "Distributed Optimization and Statistical Learning via the Alternating Direction Method of Multipliers." *Foundations and Trends in Machine Learning* 3 (1). Now Publishers, Inc.: 1–122.

Zou, Hui, and Trevor Hastie. 2005. "Regularization and Variable Selection via the Elastic Net." *Journal of the Royal Statistical Society: Series B (Statistical Methodology)* 67 (2). Wiley Online Library: 301–20.