Regularized Precision Matrix Estimation via ADMM

Matt Galloway

March 26, 2018

Abstract

`ADMMsigma` is an R package that estimates a penalized precision matrix via the alternating direction method of multipliers (ADMM) algorithm. This report will provide a brief overview of the algorithm and detail how it can be utilized to estimate precision matrices of jointly normal distributions. In addition, examples and simulation results will be provided.

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 $x \in \mathbb{R}^n, z \in \mathbb{R}^m, A \in \mathbb{R}^{p \times m}, B \in \mathbb{R}^{p \times m}, c \in \mathbb{R}^p$ – though we will later consider cases where x and z are matrices. Further, we will assume f and g are convex. 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. The optimal value is

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

Clearly, the minimization under the augmented lagrangian is equivalent to that of the usual lagrangian since any feasible point (x, z) satisfies the constraint $\rho\left\|Ax + Bz - c\right\|_2^2/2 = 0$.

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

$$x^{k+1} := \arg\min_x L_\rho(x, z^k, y^k) \tag{1}$$

$$z^{k+1} := \arg\min_z L_\rho(z^{k+1}, z, y^k) \tag{2}$$

$$y^{k+1} := y^k + \rho(Ax^{k+1} + Bz^{k+1} - c) \tag{3}$$

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

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

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

where $S = \sum_{i=1}^n (X_i - \bar{X})(X_i - \bar{X})^T/n$. It is straight forward to show that when the solution exists, $\hat{\Omega} = S^{-1}$.

We can construct a *penalized* likelihood estimator by adding a penalty term, $P\left(\Omega\right)$, to the likelihood:

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

Throughout the rest of this document we will take $P\left(\Omega\right)$ to be $P\left(\Omega\right) = \lambda\left[\frac{1-\alpha}{2}\left\|\Omega\right|_F^2 + \alpha\left\|\Omega\right\|_1\right]$ so that the full penalized likelihood is as follows:

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

where $0 \leq \alpha \leq 1$, $\lambda > 0$, $\left\|\cdot\right\|_F^2$ is the Frobenius norm and we define $\left\|A\right\|_1 = \sum_{i,j}\left|A_{ij}\right|$. 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 $\alpha = 1$ this 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 regularized precision matrix estimation is

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

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

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

2.1 Condensed-Form ADMM

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

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

The condensed-form can now be written as follows:

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

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

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

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. For instance, 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) – a limitation that could prolong the convergence rate. Because of this, we will only consider the non-condensed form for the remainder of this report.

2.2 Algorithm

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

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

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

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

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

3. Update Λ.

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

2.2.1 Proof of (1):

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

Code snippet:

Note this is not the actual code. The real code is written in c++.

```
# ridge penalized precision matrix
# function
RIDGEsigma = function(S, lam) {

    # dimensions
    p = dim(S)[1]

    # gather eigen values of S (spectral
    # decomposition)
    e.out = eigen(S, symmetric = TRUE)

    # augment eigen values for omega hat
    new.evs = (-e.out$val + sqrt(e.out$val^2 +
        4 * lam))/(2 * lam)

    # compute omega hat for lambda (zero
    # gradient equation)
    omega = tcrossprod(e.out$vec * rep(new.evs,
        each = p), e.out$vec)

    # compute gradient
    grad = S - qr.solve(omega) + lam * omega

    return(list(omega = omega, gradient = grad))
}
```

2.2.2 Proof of (2)

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

Code snippet:

Note this is not the actual code. The real code is written in c++.

```
# ADMMsigma function
ADMMsigma = function(X = NULL, S = NULL,
    lam, alpha = 1, rho = 2, mu = 10, tau1 = 2,
    tau2 = 2, tol1 = 1e-04, tol2 = 1e-04,
    maxit = 1000) {

    # compute sample covariance matrix, if
    # necessary
    if (is.null(S)) {

        # covariance matrix
        n = dim(X)[1]
        S = (n - 1)/n * cov(X)

    }

    # allocate memory
    p = dim(S)[1]
    criterion = TRUE
    iter = lik = s = r = eps1 = eps2 = 0
    new.Z = Y = Omega = matrix(0, nrow = p,
        ncol = p)

    # loop until convergence
    while (criterion && (iter <= maxit)) {

        # ridge equation (1) gather eigen values
        # (spectral decomposition)
        Z = new.Z
        Omega = sigma_ridge(S + Y - rho *
            Z, lam = rho)$omega

        # penalty equation (2) soft-thresholding
        new.Z = soft(Y + rho * Omega, lam *
            alpha)/(lam * (1 - alpha) + rho)

        # update U (3)
        Y = Y + rho * (Omega - new.Z)

        # calculate new rho
        s = sqrt(sum((rho * (new.Z - Z))^2))
        r = sqrt(sum((Omega - new.Z)^2))
```

```
        rho = rho * (tau1 * (r > mu * s) +
            (s > mu * r)/tau2 + (s/mu <=
            r & r <= mu * s))
        iter = iter + 1

        # stopping criterion
        eps1 = p * tol1 + tol2 * max(sqrt(sum(Omega^2)),
            sqrt(sum(new.Z^2)))
        eps2 = p * tol1 + tol2 * sqrt(sum(Y^2))
        criterion = (r >= eps1 || s >= eps2)

    }
    return(list(Iterations = iter, Omega = Omega))
}
```

3 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 for cross validation errors
- `plot.RIDGEsigma()` produces a heat map for cross validation errors

3.2 Usage

```
library(ADMMsigma)

# generate data from a dense matrix first
# compute covariance matrix
S = matrix(0.9, nrow = 5, ncol = 5)
diag(S) = 1

# print oracle precision matrix
(Omega = qr.solve(S))
```

```
##           [,1]      [,2]      [,3]      [,4]      [,5]
## [1,]  8.043478 -1.956522 -1.956522 -1.956522 -1.956522
## [2,] -1.956522  8.043478 -1.956522 -1.956522 -1.956522
## [3,] -1.956522 -1.956522  8.043478 -1.956522 -1.956522
## [4,] -1.956522 -1.956522 -1.956522  8.043478 -1.956522
## [5,] -1.956522 -1.956522 -1.956522 -1.956522  8.043478
```

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


# elastic-net type penalty (set tolerance
# to 1e-8)
ADMMsigma(X, tol1 = 1e-08, tol2 = 1e-08)
```

```
##
## Iterations:
## [1] 64
##
## Tuning parameters:
##       log10(lam)  alpha
## [1,]        -3.5    0.4
##
## Omega:
##          [,1]     [,2]     [,3]     [,4]     [,5]
## [1,]  7.91807 -1.97529 -1.64919 -2.25595 -1.71752
## [2,] -1.97529  8.07187 -1.90879 -2.09746 -1.89548
## [3,] -1.64919 -1.90879  7.69706 -2.33402 -1.61316
## [4,] -2.25595 -2.09746 -2.33402  8.92715 -2.08656
## [5,] -1.71752 -1.89548 -1.61316 -2.08656  7.55797
```

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

```
##
## Iterations:
```

```
## [1] 14
##
## Tuning parameters:
##       log10(lam)  alpha
## [1,]          -5      0
##
## Omega:
##          [,1]     [,2]     [,3]     [,4]     [,5]
## [1,]  8.06721 -2.01562 -1.67416 -2.30933 -1.74603
## [2,] -2.01562  8.22956 -1.94526 -2.14268 -1.93167
## [3,] -1.67416 -1.94526  7.83747 -2.39030 -1.63624
## [4,] -2.30933 -2.14268 -2.39030  9.12550 -2.13158
## [5,] -1.74603 -1.93167 -1.63624 -2.13158  7.69125
```

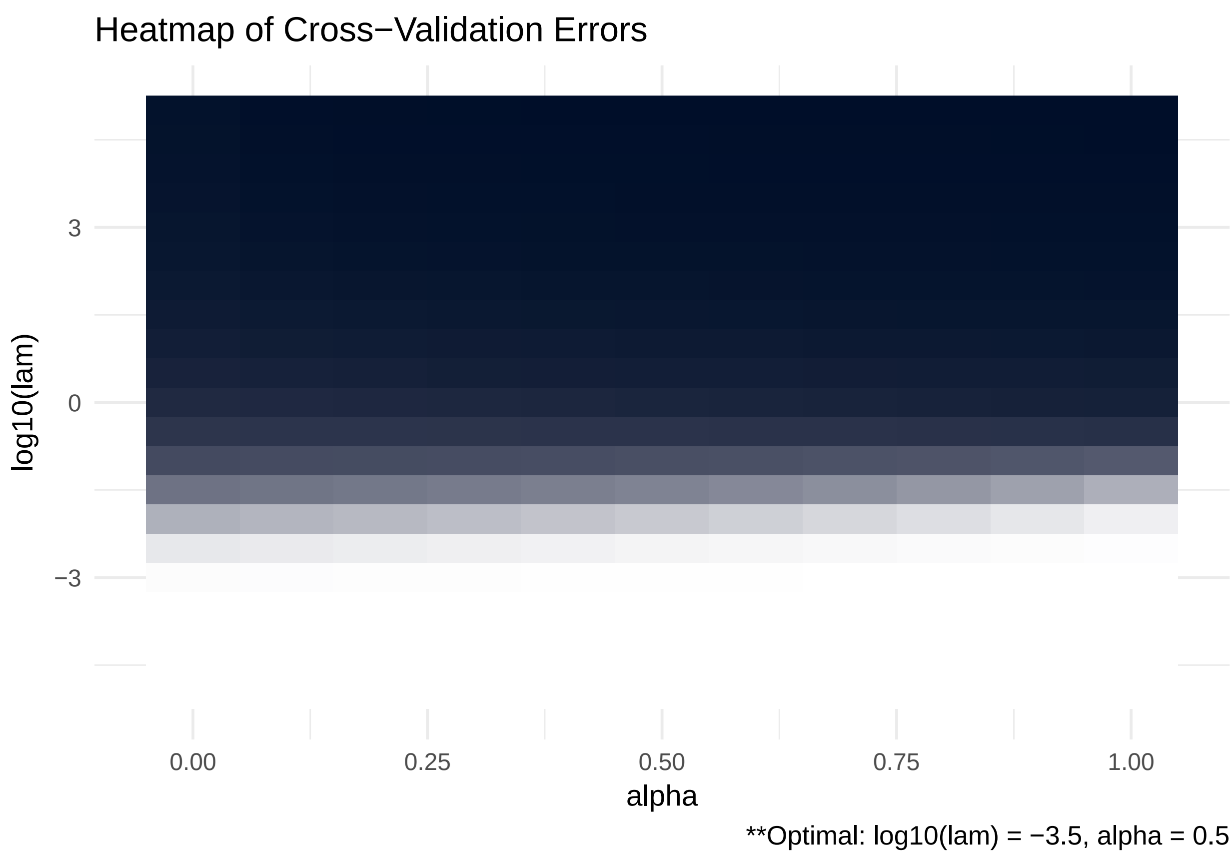
```
# lasso penalty
ADMMsigma(X, alpha = 1)
```

```
##
## Iterations:
## [1] 12
##
## Tuning parameters:
##       log10(lam)  alpha
## [1,]        -3.5      1
##
## Omega:
##          [,1]     [,2]     [,3]     [,4]     [,5]
## [1,]  8.03256 -2.00651 -1.66500 -2.30220 -1.73714
## [2,] -2.00651  8.19426 -1.93613 -2.13439 -1.92274
## [3,] -1.66500 -1.93613  7.80362 -2.38307 -1.62771
## [4,] -2.30220 -2.13439 -2.38307  9.09530 -2.12379
## [5,] -1.73714 -1.92274 -1.62771 -2.12379  7.65707
```

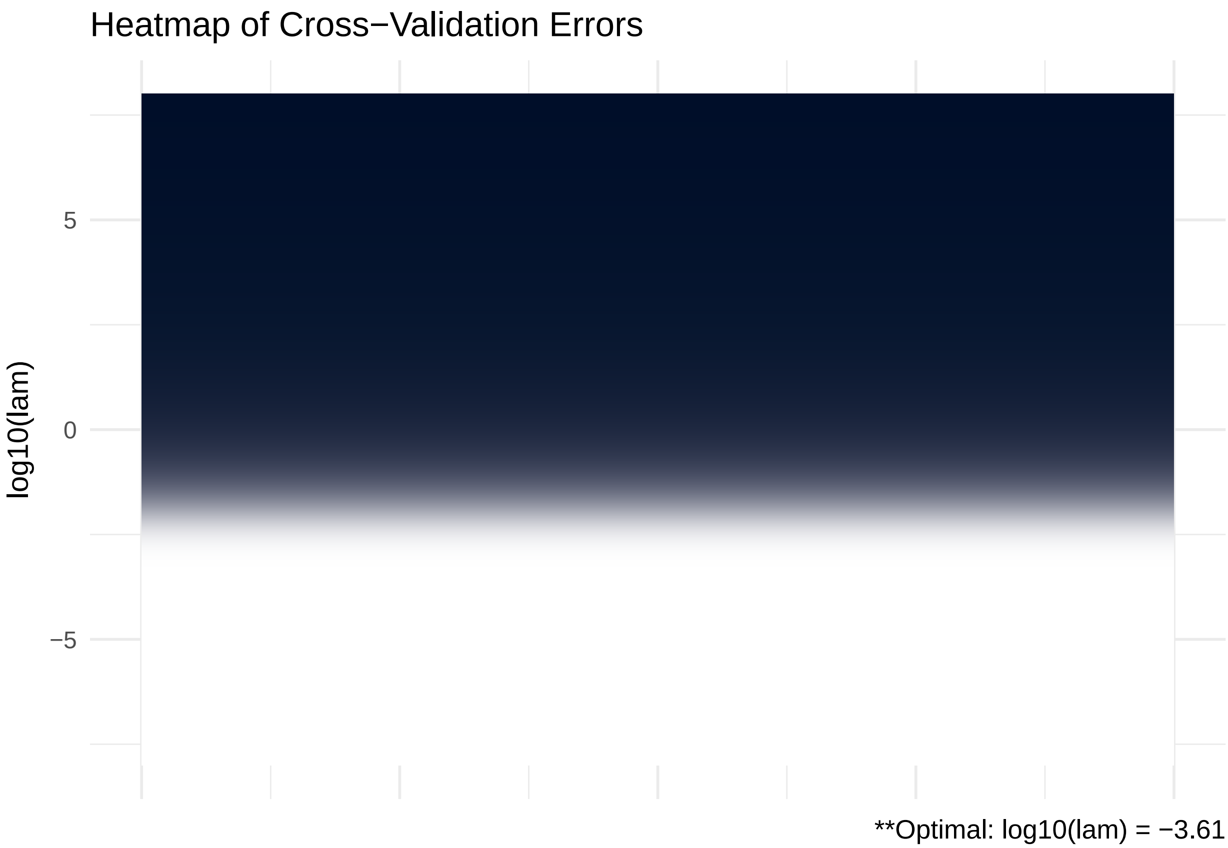
```
# ridge penalty no ADMM
RIDGEsigma(X, lam = 10^seq(-8, 8, 0.01))
```

```
##
## Tuning parameter:
##       log10(lam)  lam
## [1,]       -3.75    0
##
## Omega:
##          [,1]     [,2]     [,3]     [,4]     [,5]
## [1,]  7.94632 -1.98282 -1.65399 -2.26605 -1.72298
## [2,] -1.98282  8.10151 -1.91566 -2.10599 -1.90231
## [3,] -1.65399 -1.91566  7.72381 -2.34465 -1.61771
## [4,] -2.26605 -2.10599 -2.34465  8.96466 -2.09511
## [5,] -1.72298 -1.90231 -1.61771 -2.09511  7.58343
```

```
# produce CV heat map for ADMMsigma
ADMMsigma(X, tol1 = 1e-08, tol2 = 1e-08) %>%
    plot
```




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



3.3 Benchmark

3.3.1 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)
```

```r
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) 49.49746 51.2968 55.21674 53.17891 56.54074
##       max neval
##  94.10694   100
```

```r
# benchmark ADMMsigma - default tolerance
microbenchmark(ADMMsigma(S = S, lam = 0.1,
    alpha = 1, tol1 = 1e-04, tol2 = 1e-04))
```

```
## Unit: milliseconds
##                                                                 expr
##  ADMMsigma(S = S, lam = 0.1, alpha = 1, tol1 = 1e-04, tol2 = 1e-04)
##       min       lq     mean   median       uq      max neval
##  40.48786 41.80211 45.23219 42.58309 44.18543 212.5204   100
```

```r
# benchmark ADMMsigma - tolerance 1e-8
microbenchmark(ADMMsigma(S = S, lam = 0.1,
    alpha = 1, tol1 = 1e-08, tol2 = 1e-08))
```

```
## Unit: milliseconds
##                                                                 expr
##  ADMMsigma(S = S, lam = 0.1, alpha = 1, tol1 = 1e-08, tol2 = 1e-08)
##       min       lq     mean   median       uq      max neval
##  185.7857 190.2324 198.1983 193.4001 198.3315 256.4238   100
```

```r
# benchmark ADMMsigma CV - default
# parameter grid
microbenchmark(ADMMsigma(X), times = 5)
```

```
## Unit: seconds
##          expr      min       lq    mean   median       uq      max neval
##  ADMMsigma(X) 16.30647 16.30802 16.3881 16.40673 16.45475 16.46454     5
```

```r
# benchmark ADMMsigma parallel CV
microbenchmark(ADMMsigma(X, cores = 3), times = 5)
```

```
## Unit: seconds
##                     expr      min       lq     mean   median       uq
##  ADMMsigma(X, cores = 3) 12.95143 13.07855 13.29565 13.29788 13.54759
##       max neval
##  13.60278     5
```

```r
# benchmark ADMMsigma CV - likelihood
# convergence criteria
microbenchmark(ADMMsigma(X, crit = "loglik"),
```

```
    times = 5)
```

```
## Unit: seconds
##                           expr      min       lq     mean   median
##  ADMMsigma(X, crit = "loglik") 52.37604 52.58107 52.74499 52.68818
##        uq      max neval
##  52.77534 53.30433     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.