Journal of Statistical Software

August 2020, Volume 94, Issue 14. doi: 10.18637/jss.v094.i14

CVXR: An R Package for Disciplined Convex Optimization

Anqi Fu Stanford University

Balasubramanian Narasimhan Stanford University

Stephen Boyd Stanford University

Abstract

CVXR is an R package that provides an object-oriented modeling language for convex optimization, similar to **CVX**, **CVXPY**, **YALMIP**, and **Convex.jl**. It allows the user to formulate convex optimization problems in a natural mathematical syntax rather than the restrictive form required by most solvers. The user specifies an objective and set of constraints by combining constants, variables, and parameters using a library of functions with known mathematical properties. **CVXR** then applies signed disciplined convex programming (DCP) to verify the problem's convexity. Once verified, the problem is converted into standard conic form using graph implementations and passed to a cone solver such as **ECOS** or **SCS**. We demonstrate **CVXR**'s modeling framework with several applications.

Keywords: convex optimization, disciplined convex optimization, optimization, regression, penalized regression, isotonic regression, R package **CVXR**.

1. Introduction

Optimization plays an important role in fitting many statistical models. Some examples include least squares, ridge and lasso regression, isotonic regression, Huber regression, support vector machines, and sparse inverse covariance estimation. Koenker and Mizera (2014) discuss the role of convex optimization in statistics and provide a survey of packages for solving such problems in R (R Core Team 2020). Our package, **CVXR** (Fu, Narasimhan, Kang, Diamond, and Miller 2020), solves a broad class of convex optimization problems, which includes those noted above as well as many other models and methods in statistics. Similar systems already exist, such as **CVX** (Grant and Boyd 2014) and **YALMIP** (Lofberg 2004) in MATLAB (The MathWorks Inc. 2019), **CVXPY** (Diamond and Boyd 2016) in Python (Van Rossum *et al.* 2011), and **Convex.jl** (Udell, Mohan, Zeng, Hong, Diamond, and Boyd 2014) in Ju-

lia (Bezanson, Karpinski, Shah, and Edelman 2012). **CVXR** brings these capabilities to R, providing a domain-specific language (DSL) that allows users to easily formulate and solve new problems for which custom code does not exist. As an illustration, suppose we are given $X \in \mathbf{R}^{m \times n}$ and $y \in \mathbf{R}^m$, and we want to solve the ordinary least squares (OLS) problem

$$\underset{\beta}{\text{minimize}} \quad \|y - X\beta\|_2^2$$

with optimization variable $\beta \in \mathbf{R}^n$. This problem has a well-known analytical solution, which can be determined using `lm` in the default **stats** package. In **CVXR**, we can solve for β using the code

```
R> beta <- Variable(n)
R> obj <- sum((y - X %*% beta)^2)
R> prob <- Problem(Minimize(obj))
R> result <- solve(prob)
```

The first line declares our variable, the second line forms our objective function, the third line defines the optimization problem, and the last line solves this problem by converting it into a second-order cone program and sending it to one of **CVXR**'s solvers. The results for the optimal objective, the optimal variables, and the solver runtime, respectively, are retrieved with

```
R> result$value
R> result$getValue(beta)
R> result$solve_time
```

This code runs slower and requires additional set-up at the beginning. So far, it does not look like an improvement on `stats::lm`. However, suppose we add a constraint to our problem:

$$\begin{aligned} \underset{\beta}{\text{minimize}} \quad & \|y - X\beta\|_2^2 \\ \text{subject to} \quad & \beta_j \leq \beta_{j+1}, \quad j = 1, \ldots, n-1. \end{aligned}$$

This is a special case of isotonic regression. Now, we can no longer use `stats::lm` for the optimization. We would need to find another R package tailored to this type of problem such as **nnls** (Mullen and Van Stokkum 2012) or write our own custom solver. With **CVXR** though, we need only add the constraint as a second argument to the problem:

```
R> prob <- Problem(Minimize(obj), list(diff(beta) >= 0))
```

Our new problem definition includes the coefficient constraint, and a call to `solve` will produce its solution. In addition to the usual results, we can get the dual variables with

```
R> result$getDualValue(constraints(prob)[[1]])
```

This example demonstrates **CVXR**'s chief advantage: flexibility. Users can quickly modify and re-solve a problem, making our package ideal for prototyping new statistical methods. Its syntax is simple and mathematically intuitive. Furthermore, **CVXR** combines seamlessly with native R code as well as several popular packages, allowing it to be incorporated easily

into a larger analytical framework. The user can, for instance, apply resampling techniques like the bootstrap to estimate variability, as we show in Section 3.2.

DSLs for convex optimization are already widespread on other application platforms. In R, users have access to the packages listed in the CRAN Task View for *Optimization and Mathematical Programming* (Theußl, Schwendinger, and Borchers 2020a). Packages like **optimx** (Nash and Varadhan 2011) and **nloptr** (Johnson 2008) provide access to a variety of general algorithms, which can handle nonlinear and certain classes of nonconvex problems. **CVXR**, on the other hand, offers a language to express convex optimization problems using R syntax, along with a tool for analyzing and restructuring them for the solver best suited to their type. **ROI** (Theußl, Schwendinger, and Hornik 2020b) is perhaps the package closest to ours in spirit. It offers an object-oriented framework for defining optimization problems, but still requires users to explicitly identify the type of every objective and constraint, whereas **CVXR** manages this process automatically.

In the next section, we provide a brief mathematical overview of convex optimization. Interested readers can find a full treatment in Boyd and Vandenberghe (2004). Then we give a series of examples ranging from basic regression models to semidefinite programming, which demonstrate the simplicity of problem construction in **CVXR**. Finally, we describe the implementation details before concluding. Our package and the example code for this paper are available on the Comprehensive R Archive Network (CRAN) at `https://CRAN.R-project.org/package=CVXR` and the official **CVXR** site at `https://cvxr.rbind.io`.

2. Disciplined convex optimization

The general convex optimization problem is of the form

$$\begin{array}{ll} \underset{v}{\text{minimize}} & f_0(v) \\ \text{subject to} & f_i(v) \leq 0, \quad i = 1, \ldots, M \\ & Av = b, \end{array}$$

where $v \in \mathbf{R}^n$ is our variable of interest, and $A \in \mathbf{R}^{m \times n}$ and $b \in \mathbf{R}^n$ are constants describing our linear equality constraints. The objective and inequality constraint functions $f_0, \ldots, f_M$ are convex, i.e., they are functions $f_i : \mathbf{R}^n \to \mathbf{R}$ that satisfy

$$f_i(\theta u + (1-\theta)v) \leq \theta f_i(u) + (1-\theta) f_i(v)$$

for all $u, v \in \mathbf{R}^n$ and $\theta \in [0, 1]$. This class of problems arises in a variety of fields, including machine learning and statistics.

A number of efficient algorithms exist for solving convex problems (Wright 1997; Boyd, Parikh, Chu, Peleato, and Eckstein 2011; Andersen, Dahl, Liu, and Vandenberghe 2011; Skajaa and Ye 2015). However, it is unnecessary for the **CVXR** user to know the operational details of these algorithms. **CVXR** provides a DSL that allows the user to specify the problem in a natural mathematical syntax. This specification is automatically converted into the standard form ingested by a generic convex solver. See Section 4 for more on this process.

In general, it can be difficult to determine whether an optimization problem is convex. We follow an approach called disciplined convex programming (DCP; Grant, Boyd, and Ye 2006) to define problems using a library of basic functions (atoms), whose properties like curvature, monotonicity, and sign are known. Adhering to the DCP rule,

$f(g_1, \ldots, g_k)$ is convex if f is convex and for each $i = 1, \ldots, k$, either

- g_i is affine,
- g_i is convex and f is increasing in argument i, or
- g_i is concave and f is decreasing in argument i,

we combine these atoms such that the resulting problem is convex by construction. Users will need to become familiar with this rule if they wish to define complex problems.

The library of available atoms is provided in the documentation. It covers an extensive array of functions, enabling any user to model and solve a wide variety of sophisticated optimization problems. In the next section, we provide sample code for just a few of these problems, many of which are cumbersome to prototype or solve with other R packages.

3. Examples

In the following examples, we are given a dataset (x_i, y_i) for $i = 1, \ldots, m$, where $x_i \in \mathbf{R}^n$ and $y_i \in \mathbf{R}$. We represent these observations in matrix form as $X \in \mathbf{R}^{m \times n}$ with stacked rows $x_i^\top$ and $y \in \mathbf{R}^m$. Generally, we assume that $m > n$.

3.1. Regression

Robust (Huber) regression

In Section 1, we saw an example of OLS in **CVXR**. While least squares is a popular regression model, one of its flaws is its high sensitivity to outliers. A single outlier that falls outside the tails of the normal distribution can drastically alter the resulting coefficients, skewing the fit on the other data points. For a more robust model, we can fit a Huber regression (Huber 1964) instead by solving

$$\underset{\beta}{\text{minimize}} \quad \sum_{i=1}^m \phi(y_i - x_i^\top \beta)$$

for variable $\beta \in \mathbf{R}^n$, where the loss is the Huber function with threshold $M > 0$,

$$\phi(u) = \begin{cases} \frac{1}{2}u^2 & \text{if } |u| \leq M \\ M|u| - \frac{1}{2}M^2 & \text{if } |u| > M. \end{cases}$$

This function is identical to the least squares penalty for small residuals, but on large residuals, its penalty is lower and increases linearly rather than quadratically. It is thus more forgiving of outliers.

In **CVXR**, the code for this problem is

```
R> beta <- Variable(n)
R> obj <- sum(huber(y - X %*% beta, M))
R> prob <- Problem(Minimize(obj))
R> result <- solve(prob)
```

Note the similarity to the OLS code. As before, the first line instantiates the n-dimensional optimization variable, and the second line defines the objective function by combining this variable with our data using **CVXR**'s library of atoms. The only difference this time is we call the `huber` atom on the residuals with threshold `M`, which we assume has been set to a positive scalar constant. Our package provides many such atoms to simplify problem definition for the user.

Quantile regression

Another variation on least squares is quantile regression (Koenker 2005). The loss is the tilted l_1 function,

$$\phi(u) = \tau \max(u, 0) - (1 - \tau) \max(-u, 0) = \frac{1}{2}|u| + \left(\tau - \frac{1}{2}\right) u,$$

where $\tau \in (0, 1)$ specifies the quantile. The problem as before is to minimize the total residual loss. This model is commonly used in ecology, healthcare, and other fields where the mean alone is not enough to capture complex relationships between variables. **CVXR** allows us to create a function to represent the loss and integrate it seamlessly into the problem definition, as illustrated below.

```
R> quant_loss <- function(u, tau) 0.5 * abs(u) + (tau - 0.5) * u
R> obj <- sum(quant_loss(y - X %*% beta, t))
R> prob <- Problem(Minimize(obj))
R> result <- solve(prob)
```

Here `t` is the user-defined quantile parameter. We do not need to create a new '`Variable`' object, since we can reuse `beta` from the previous example.

By default, the `solve` method automatically selects the **CVXR** solver most specialized to the given problem's type. This solver may be changed by passing in an additional `solver` argument. For instance, the following line fits our quantile regression with **SCS** (O'Donoghue, Chu, Parikh, and Boyd 2016).

```
R> result <- solve(prob, solver = "SCS")
```

Elastic net regularization

Often in applications, we encounter problems that require regularization to prevent overfitting, introduce sparsity, facilitate variable selection, or impose prior distributions on parameters. Two of the most common regularization functions are the l_1-norm and squared l_2-norm, combined in the elastic net regression model (Hastie and Zou 2005; Friedman, Hastie, and Tibshirani 2010),

$$\underset{\beta}{\text{minimize}} \quad \tfrac{1}{2m}\|y - X\beta\|_2^2 + \lambda(\tfrac{1-\alpha}{2}\|\beta\|_2^2 + \alpha\|\beta\|_1).$$

Here $\lambda \geq 0$ is the overall regularization weight and $\alpha \in [0, 1]$ controls the relative l_1 versus squared l_2 penalty. Thus, this model encompasses both ridge ($\alpha = 0$) and lasso ($\alpha = 1$) regression.

To solve this problem in **CVXR**, we first define a function that calculates the regularization term given the variable and penalty weights.

```
R> elastic_reg <- function(beta, lambda = 0, alpha = 0) {
+    ridge <- (1 - alpha) * sum(beta^2)
+    lasso <- alpha * p_norm(beta, 1)
+    lambda * (lasso + ridge)
+  }
```

Then, we add it to the scaled least squares loss.

```
R> loss <- sum((y - X %*% beta)^2) / (2 * m)
R> obj <- loss + elastic_reg(beta, lambda, alpha)
R> prob <- Problem(Minimize(obj))
R> result <- solve(prob)
```

The advantage of this modular approach is that we can easily incorporate elastic net regularization into other regression models. For instance, if we wanted to run regularized Huber regression, **CVXR** allows us to reuse the above code with just a single changed line,

```
R> loss <- sum(huber(y - X %*% beta, M))
```

Logistic regression

Suppose now that $y_i \in \{0, 1\}$ is a binary class indicator. One of the most popular methods for binary classification is logistic regression (Cox 1958; Freedman 2009). We model the conditional response as $y|x \sim \text{Bernoulli}(g_\beta(x))$, where $g_\beta(x) = \frac{1}{1+e^{-x^\top \beta}}$ is the logistic function, and maximize the log-likelihood function, yielding the optimization problem

$$\underset{\beta}{\text{maximize}} \quad \sum_{i=1}^{m} \{y_i \log(g_\beta(x_i)) + (1 - y_i) \log(1 - g_\beta(x_i))\}.$$

CVXR provides the `logistic` atom as a shortcut for $f(z) = \log(1 + e^z)$, so our problem is succinctly expressed as

```
R> obj <- -sum(X[y == 0, ] %*% beta) - sum(logistic(-X %*% beta))
R> prob <- Problem(Maximize(obj))
R> result <- solve(prob)
```

The user may be tempted to type `log(1 + exp(X %*% beta))` as in conventional R syntax. However, this representation of $f(z)$ violates the DCP composition rule, so the **CVXR** parser will reject the problem even though the objective is convex. Users who wish to employ a function that is convex, but not DCP compliant should check the documentation for a custom atom or consider a different formulation.

We can retrieve the optimal objective and variables just like in OLS. More interestingly, we can evaluate various functions of these variables as well by passing them directly into `result$getValue`. For instance, the log-odds are

```
R> log_odds <- result$getValue(X %*% beta)
```

This will coincide with the ratio we get from computing the probabilities directly:

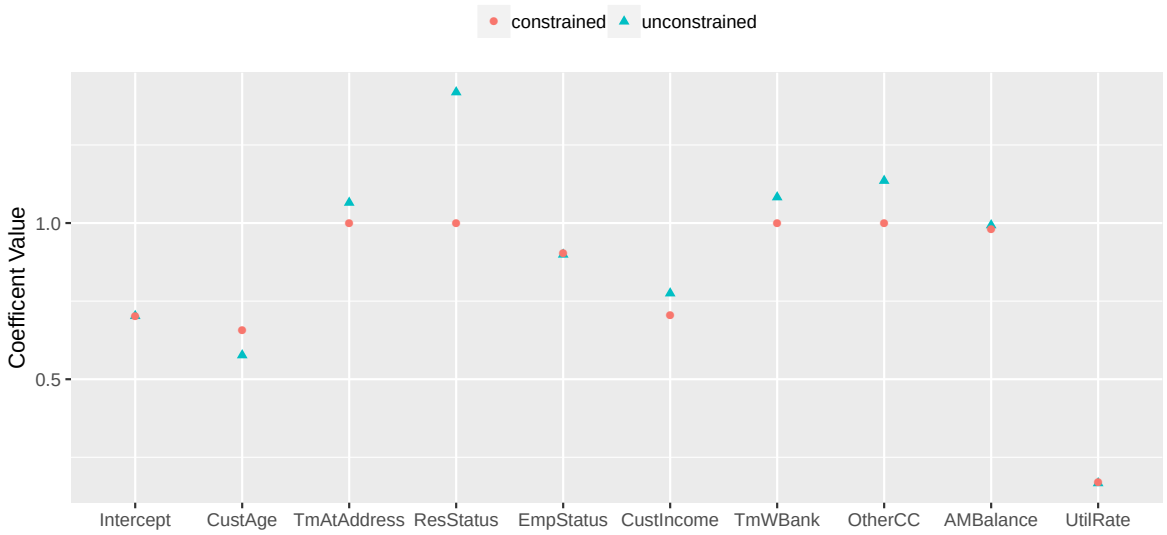


Figure 1: Logistic regression with constraints using data from The MathWorks Inc. (2018). The addition of constraint 1 moves the coefficients for customer age and customer income closer to each other.

```
R> beta_res <- result$getValue(beta)
R> y_probs <- 1 / (1 + exp(-X %*% beta_res))
R> log(y_probs / (1 - y_probs))
```

We illustrate with a logistic regression fit from a credit scoring example (The MathWorks Inc. 2018). The nine regression coefficients other than the intercept are constrained to be in the unit interval. To reflect the correlation between two of the covariates, customer age (x_2) and customer income (x_6), an additional constraint is placed on the respective coefficients β_2 and β_6:

$$|\beta_2 - \beta_6| \leq 0.5. \tag{1}$$

The code below demonstrates how the latter constraint can be specified by seamlessly combining familiar R functions such as `abs` with standard indexing constructs.

```
R> constr <- list(beta[2:10] >= 0, beta[2:10] <= 1,
+    abs(beta[2] - beta[6]) <= 0.05)
R> prob <- Problem(Maximize(obj), constr)
R> result <- solve(prob)
R> beta_res_con <- result$getValue(beta)
```

Figure 1 compares the unconstrained and constrained fits and shows that the addition of constraint 1 pulls the coefficient estimates for customer age and customer income towards each other.

Many other classification methods belong to the convex framework. For example, the support vector classifier is the solution of a l_2-norm minimization problem with linear constraints, which we have already shown how to model. Support vector machines are a straightforward

extension. The multinomial distribution can be used to predict multiple classes, and estimation via maximum likelihood produces a convex problem. To each of these methods, we can easily add new penalties, variables, and constraints in **CVXR**, allowing us to adapt to a specific dataset or environment.

Sparse inverse covariance estimation

Assume we are given i.i.d. observations $x_i \sim N(0, \Sigma)$ for $i = 1, \ldots, m$, and the covariance matrix $\Sigma \in \mathbf{S}_+^n$, the set of symmetric positive semidefinite matrices, has a sparse inverse $S = \Sigma^{-1}$. Let $Q = \frac{1}{m-1}\sum_{i=1}^m (x_i - \bar{x})(x_i - \bar{x})^\top$ be our sample covariance. One way to estimate Σ is to maximize the log-likelihood with an l_1-norm constraint (Yuan and Lin 2007; Banerjee, Ghaoui, and d'Aspremont 2008; Friedman, Hastie, and Tibshirani 2008), which amounts to the optimization problem

$$\begin{array}{ll} \underset{S}{\text{maximize}} & \log\det(S) - \text{tr}(SQ) \\ \text{subject to} & S \in \mathbf{S}_+^n, \quad \sum_{i=1}^n \sum_{j=1}^n |S_{ij}| \leq \alpha. \end{array}$$

The parameter $\alpha \geq 0$ controls the degree of sparsity. Our problem is convex, so we can solve it with

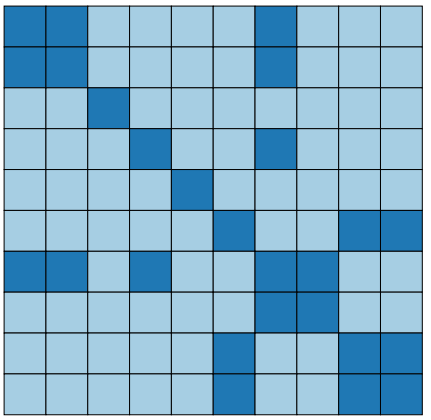
```
R> S <- Variable(n, n, PSD = TRUE)
R> obj <- log_det(S) - matrix_trace(S %*% Q)
R> constr <- list(sum(abs(S)) <= alpha)
R> prob <- Problem(Maximize(obj), constr)
R> result <- solve(prob, solver = "SCS")
```

The `PSD = TRUE` argument to the `Variable` constructor restricts `S` to the positive semidefinite cone. In our objective, we use **CVXR** functions for the log-determinant and trace. The expression `matrix_trace(S %*% Q)` is equivalent to `sum(diag(S %*% Q))`, but the former is preferred because it is more efficient than making nested function calls. However, a standalone atom does not exist for the determinant, so we cannot replace `log_det(S)` with `log(det(S))` since `det` is undefined for a '`Variable`' object.

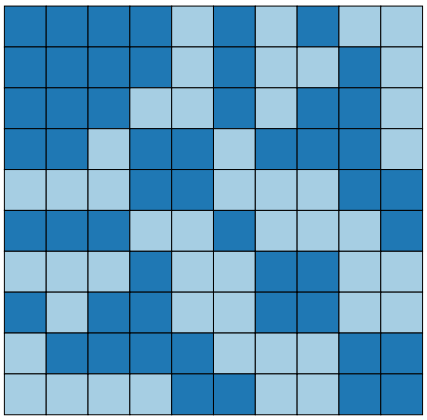
Figure 2 depicts the solutions for a particular dataset with $m = 1000, n = 10$, and S containing 26% non-zero entries represented by the black squares in the top left image. The sparsity of our inverse covariance estimate decreases for higher α, so that when $\alpha = 1$, most of the off-diagonal entries are zero, while if $\alpha = 10$, over half the matrix is dense. At $\alpha = 4$, we achieve the true percentage of non-zeros.

Saturating hinges

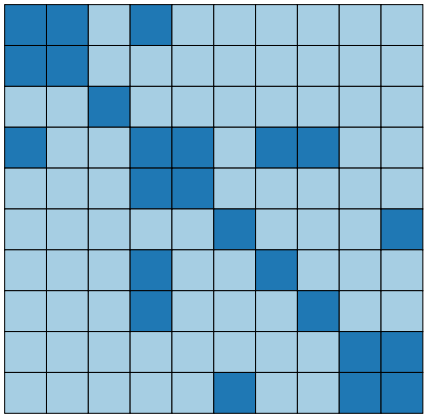
The following example comes from work on saturating splines in Boyd, Hastie, Boyd, Recht, and Jordan (2018). Adaptive regression splines are commonly used in statistical modeling, but the instability they exhibit beyond their boundary knots makes extrapolation dangerous. One way to correct this issue for linear splines is to require they *saturate*: remain constant outside their boundary. This problem can be solved using a heuristic that is an extension of lasso regression, producing a weighted sum of hinge functions, which we call a *saturating hinge*.



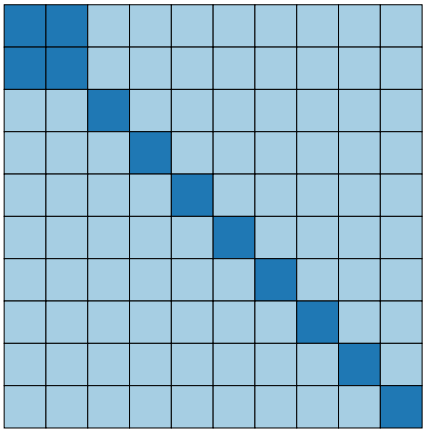
(a) True inverse.



(b) $\alpha = 10$.



(c) $\alpha = 4$.



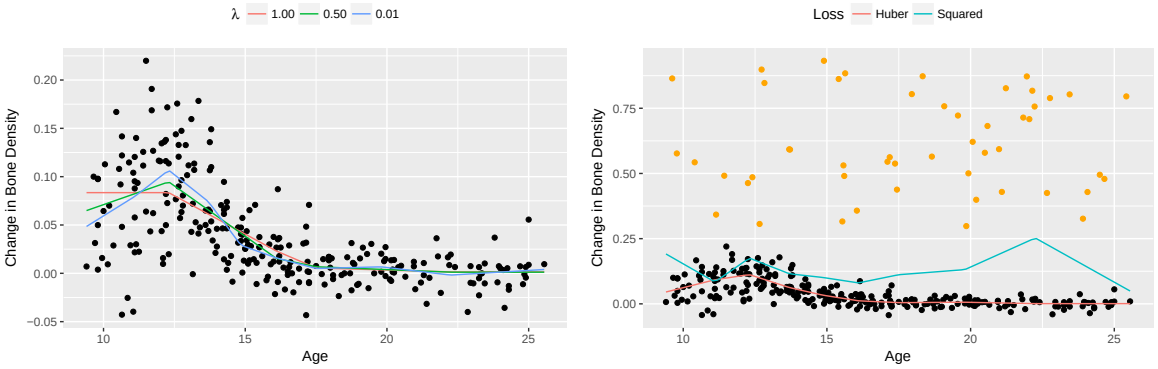
(d) $\alpha = 1$.

Figure 2: Sparsity patterns for (a) inverse of true covariance matrix, and estimated inverse covariance matrices with (b) $\alpha = 10$, (c) $\alpha = 4$, and (d) $\alpha = 1$. The light blue regions indicate where $S_{ij} = 0$.

For simplicity, consider the univariate case with $n = 1$. Assume we are given knots $t_1 < t_2 < \cdots < t_k$ where each $t_j \in \mathbf{R}$. Let h_j be a hinge function at knot t_j, i.e., $h_j(x) = \max(x - t_j, 0)$, and define $f(x) = w_0 + \sum_{j=1}^{k} w_j h_j(x)$. We want to solve

$$\begin{array}{ll} \underset{w_0, w}{\text{minimize}} & \sum_{i=1}^{m} \ell(y_i, f(x_i)) + \lambda \|w\|_1 \\ \text{subject to} & \sum_{j=1}^{k} w_j = 0 \end{array}$$

for variables $(w_0, w) \in \mathbf{R} \times \mathbf{R}^k$. The function $\ell : \mathbf{R} \times \mathbf{R} \to \mathbf{R}$ is the loss associated with every observation, and $\lambda \geq 0$ is the penalty weight. In choosing our knots, we set $t_1 = \min(x_i)$ and $t_k = \max(x_i)$ so that by construction, the estimate $\hat{f}$ will be constant outside $[t_1, t_k]$.



(a) Squared error loss.

(b) Saturating hinges with outliers.

Figure 3: (a) Saturating hinges fit to the change in bone density for female patients with $\lambda = 0.01$ (blue), $\lambda = 0.5$ (green), and $\lambda = 1$ (red). (b) Hinges refit to the previous data with additional outliers (orange) using squared error (blue) and Huber loss (red).

We demonstrate this technique on the bone density data for female patients from Hastie, Tibshirani, and Friedman (2001, Section 5.4). There are a total of $m = 259$ observations. Our response y_i is the change in spinal bone density between two visits, and our predictor x_i is the patient's age. We select $k = 10$ knots about evenly spaced across the range of X and fit a saturating hinge with squared error loss $\ell(y_i, f(x_i)) = (y_i - f(x_i))^2$.

In R, we first define the estimation and loss functions:

```
R> f_est <- function(x, knots, w0, w) {
+    hinges <- sapply(knots, function(t) pmax(x - t, 0))
+    w0 + hinges %*% w
+  }
R> loss_obs <- function(y, f) (y - f)^2
```

This allows us to easily test different losses and knot locations later. The rest of the set-up is similar to previous examples. We assume that `knots` is a R vector representing $(t_1, \ldots, t_k)$.

```
R> w0 <- Variable(1)
R> w <- Variable(k)
R> loss <- sum(loss_obs(y, f_est(X, knots, w0, w)))
R> reg <- lambda * p_norm(w, 1)
R> obj <- loss + reg
R> constr <- list(sum(w) == 0)
R> prob <- Problem(Minimize(obj), constr)
R> result <- solve(prob)
```

The optimal weights are retrieved using separate calls, as shown below.

```
R> w0s <- result$getValue(w0)
R> ws <- result$getValue(w)
```

We plot the fitted saturating hinges in Figure 3a. As expected, when λ increases, the spline exhibits less variation and grows flatter outside its boundaries. The squared error loss works well in this case, but as we saw previously in this section, the Huber loss is preferred when the dataset contains large outliers. We can change the loss function by simply redefining

```
R> loss_obs <- function(y, f, M) huber(y - f, M)
```

and passing an extra threshold parameter in when initializing `loss`. In Figure 3b, we have added 50 randomly generated outliers to the bone density data and plotted the re-fitted saturating hinges. For a Huber loss with $M = 0.01$, the resulting spline is fairly smooth and follows the shape of the original data, as opposed to the spline using squared error loss, which is biased upwards by a significant amount.

3.2. Nonparametric estimation

Log-concave distribution estimation

Let $n = 1$ and suppose x_i are i.i.d. samples from a log-concave discrete distribution on $\{0, \ldots, K\}$ for some $K \in \mathbf{Z}_+$. Define $p_k := \mathsf{P}(X = k)$ to be the probability mass function. One method for estimating $(p_0, \ldots, p_K)$ is to maximize the log-likelihood function subject to a log-concavity constraint (Dümbgen and Rufibach 2009), i.e.,

$$\begin{array}{ll} \underset{p}{\text{maximize}} & \sum_{k=0}^{K} M_k \log p_k \\ \text{subject to} & p \geq 0, \quad \sum_{k=0}^{K} p_k = 1, \\ & p_k \geq \sqrt{p_{k-1} p_{k+1}}, \quad k = 1, \ldots, K-1, \end{array}$$

where $p \in \mathbf{R}^{K+1}$ is our variable of interest and M_k represents the number of observations equal to k, so that $\sum_{k=0}^{K} M_k = m$. The problem as posed above is not convex. However, we can transform it into a convex optimization problem by defining new variables $u_k = \log p_k$ and relaxing the equality constraint to $\sum_{k=0}^{K} p_k \leq 1$, since the latter always holds tightly at an optimal solution. The result is

$$\begin{array}{ll} \underset{u}{\text{maximize}} & \sum_{k=0}^{K} M_k u_k \\ \text{subject to} & \sum_{k=0}^{K} e^{u_k} \leq 1, \\ & u_k - u_{k-1} \geq u_{k+1} - u_k, \quad k = 1, \ldots, K-1. \end{array}$$

If `counts` is the R vector of $(M_0, \ldots, M_K)$, the code for our convex problem is

```
R> u <- Variable(K+1)
R> obj <- t(counts) %*% u
R> constr <- list(sum(exp(u)) <= 1, diff(u[1:K)]) >= diff(u[2:(K+1)]))
R> prob <- solve(Maximize(obj), constr)
R> result <- solve(prob)
```

Once the solver is finished, we can retrieve the probabilities directly with

```
R> pmf <- result$getValue(exp(u))
```

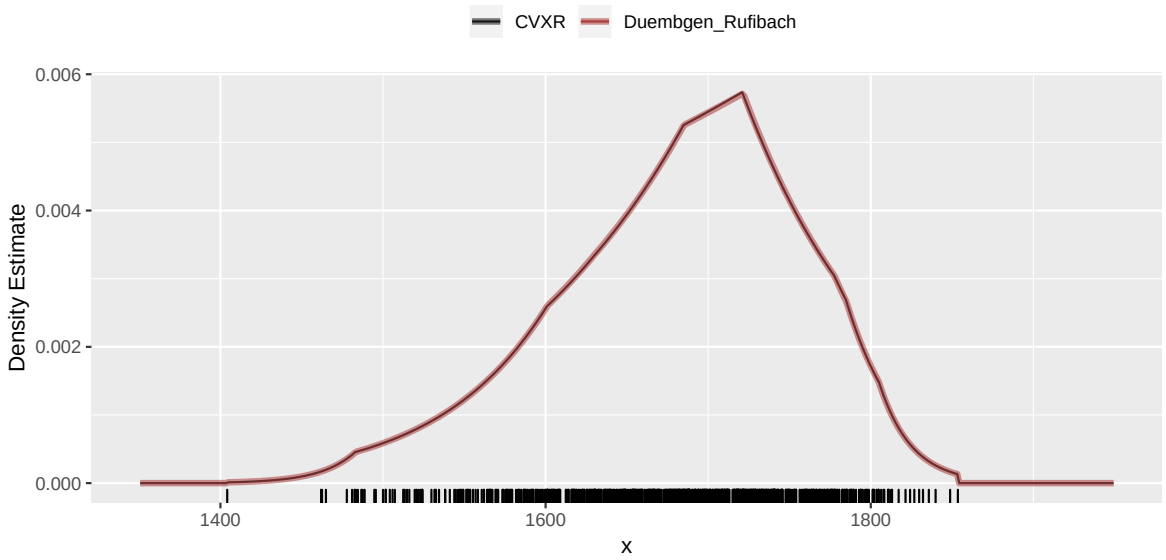


Figure 4: Log-concave estimation using the approach of Dümbgen and Rufibach (2011) and **CVXR**.

The above line transforms the variables u_k to e^{u_k} before calculating their resulting values. This is possible because `exp` is a member of **CVXR**'s library of atoms, so it can operate directly on a '`Variable`' object such as `u`.

As an example, we consider the reliability data from Dümbgen and Rufibach (2011) that was collected as part of a consulting project at the Institute for Mathematical Statistics and Actuarial Science, University of Bern (Dümbgen and Rufibach 2009). The dataset consists of $n = 786$ observations, and the goal is to fit a suitable distribution to this sample that can be used for simulations. For various reasons detailed in the paper, the authors chose a log-concave estimator, which they implemented in the R package **logcondens** (Dümbgen and Rufibach 2011). Figure 4 shows that the curve obtained from the **CVXR** code above matches their results exactly.

Survey calibration

Calibration is a widely used technique in survey sampling. Suppose m sampling units in a survey have been assigned initial weights d_i for $i = 1, \ldots, m$, and furthermore, there are n auxiliary variables whose values in the sample are known. Calibration seeks to improve the initial weights d_i by finding new weights w_i that incorporate this auxiliary information while perturbing the initial weights as little as possible, i.e., the ratio $g_i = w_i/d_i$ must be close to one. Such reweighting improves precision of estimates (Lumley 2010, Chapter 7).

Let $X \in \mathbf{R}^{m \times n}$ be the matrix of survey samples, with each column corresponding to an auxiliary variable. Reweighting can be expressed as the optimization problem

$$\begin{array}{ll} \text{minimize} & \sum_{i=1}^{m} d_i \phi(g_i) \\ \text{subject to} & A^\top g = r \end{array}$$

with respect to $g \in \mathbf{R}^m$, where $\phi : \mathbf{R} \to \mathbf{R}$ is a strictly convex function with $\phi(1) = 0$, $r \in \mathbf{R}^n$

		survey		**CVXR**	
School type	Target met?	Weight	Frequency	Weight	Frequency
E	Yes	29.00	15	29.00	15
H	No	31.40	13	31.40	13
M	Yes	29.03	9	29.03	9
E	No	28.91	127	28.91	127
H	Yes	31.50	12	31.50	12
M	No	31.53	24	31.53	24

Table 1: Raking weight estimates with **survey** package and **CVXR** for California Academic Performance Index data.

are the known population totals of the auxiliary variables, and $A \in \mathbf{R}^{m \times n}$ is related to X by $A_{ij} = d_i X_{ij}$ for $i = 1, \ldots, m$ and $j = 1, \ldots, n$. A common technique is raking, which uses the penalty function $\phi(g_i) = g_i \log(g_i) - g_i + 1$.

We illustrate with the California Academic Performance Index data in the **survey** package (Lumley 2004, 2020), which also supplies facilities for calibration via the function `calibrate`. Both the population dataset (`apipop`) and a simple random sample of $m = 200$ (`apisrs`) are provided. Suppose that we wish to reweight the observations in the sample using known totals for two variables from the population: `stype`, the school type (elementary, middle or high) and `sch.wide`, whether the school met the yearly target or not. This reweighting would make the sample more representative of the general population.

The code below solves the problem in **CVXR**, where we have used a model matrix to generate the appropriate dummy variables for the two factor variables.

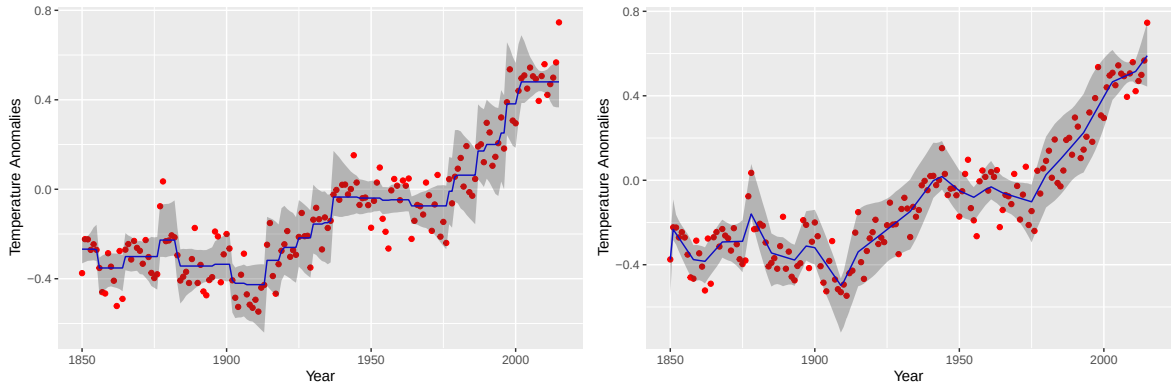
```
R> m <- nrow(apisrs)
R> di <- apisrs$pw
R> formula <- ~ stype + sch.wide
R> r <- apply(model.matrix(object = formula, data = apipop), 2, sum)
R> X <- model.matrix(object = formula, data = apisrs)
R> A <- di * X
R> g <- Variable(m)
R> obj <- sum(di * (-entr(g) - g + 1))
R> constr <- list(t(A) %*% g == r)
R> prob <- Problem(Minimize(obj), constr)
R> result <- solve(prob)
R> w_cvxr <- di * result$getValue(g)
```

Table 1 shows that the results are identical to those obtained from **survey**. **CVXR** can also accommodate other penalty functions common in the survey literature, as well as additional constraints.

Nearly-isotonic and nearly-convex fits

Given a set of data points $y \in \mathbf{R}^m$, Tibshirani, Hoefling, and Tibshirani (2011) fit a nearly-isotonic approximation $\beta \in \mathbf{R}^m$ by solving

$$\underset{\beta}{\text{minimize}} \quad \frac{1}{2}\sum_{i=1}^{m}(y_i - \beta_i)^2 + \lambda \sum_{i=1}^{m-1}(\beta_i - \beta_{i+1})_+,$$



(a) Nearly-isotonic. (b) Nearly-convex.

Figure 5: (a) A nearly-isotonic fit and (b) nearly-convex fit to global warming data on temperature anomalies for $\lambda = 0.44$. The 95% normal confidence intervals are shown in gray using $R = 400$ and $R = 200$ bootstrap samples, respectively.

where $\lambda \geq 0$ is a penalty parameter and $x_+ = \max(x, 0)$. Our **CVXR** formulation follows directly as shown below. The `pos` atom evaluates x_+ elementwise on the input expression.

```
R> near_fit <- function(y, lambda) {
+    m <- length(y)
+    beta <- Variable(m)
+    penalty <- sum(pos(diff(beta)))
+    obj <- 0.5 * sum((y - beta)^2) + lambda * penalty
+    prob <- Problem(Minimize(obj))
+    result <- solve(prob)
+    result$getValue(beta)
+  }
```

We demonstrate this technique on the global warming data provided by the Carbon Dioxide Information Analysis Center (CDIAC). Our data points are the annual temperature anomalies relative to the 1961–1990 mean. Combining `near_fit` with the **boot** package (Canty and Ripley 2020), we can obtain the standard errors and confidence intervals for our estimate in just a few lines of code. The `near_fit_stat` function first obtains a bootstrap `sample` of rows, orders them by ascending `year`, and then calls `near_fit`.

```
R> near_fit_stat <- function(data, index, lambda) {
+    sample <- data[index, ]
+    sample <- sample[order(sample$year), ]
+    near_fit(sample$annual, lambda)
+  }
R> boot.out <- boot(CDIAC, near_fit_stat, R = 400, lambda = 0.44)
```

Figure 5a shows a nearly-isotonic fit with $\lambda = 0.44$ and 95% normal confidence bands, which were generated using $R = 400$ bootstrap samples. The curve follows the data well, but exhibits choppiness in regions with a steep trend.

For a smoother curve, we can solve for the nearly-convex fit described in the same paper:

$$\underset{\beta}{\text{minimize}} \quad \tfrac{1}{2}\sum_{i=1}^{m}(y_i - \beta_i)^2 + \lambda \sum_{i=1}^{m-2}(\beta_i - 2\beta_{i+1} + \beta_{i+2})_+$$

This replaces the first difference term with an approximation to the second derivative at β_{i+1}. In **CVXR**, the only change necessary is the penalty line in `near_fit`,

```
R> penalty <- sum(pos(diff(beta, differences = 2)))
```

The resulting curve is depicted in Figure 5b with 95% confidence bands generated from $R = 200$ samples. Note the jagged staircase pattern has been smoothed out. We can easily extend this example to higher-order differences or lags by modifying the arguments to `diff`.

3.3. Miscellaneous applications

Worst case covariance

Suppose we have i.i.d. samples $x_i \sim N(0, \Sigma)$ for $i = 1, \ldots, m$ and want to determine the maximum covariance of $y = w^\top x = \sum_{i=1}^m w_i x_i$, where $w \in \mathbf{R}^m$ is a given vector of weights. We are provided limited information on the elements of Σ. For example, we may know the specific value or sign of certain Σ_{jk}, which are represented by upper and lower bound matrices L and $U \in \mathbf{R}^{n\times n}$, respectively (Boyd and Vandenberghe 2004, pp. 171–172). This situation can arise when calculating the worst-case risk of an investment portfolio (Lobo and Boyd 2000). Formally, our optimization problem is

$$\begin{array}{ll} \underset{\Sigma}{\text{maximize}} & w^\top \Sigma w \\ \text{subject to} & \Sigma \in \mathbf{S}_+^n, \quad L_{jk} \le \Sigma_{jk} \le U_{jk}, \quad j,k = 1,\ldots,n. \end{array}$$

Consider the specific case

$$w = \begin{bmatrix} 0.1 \\ 0.2 \\ -0.05 \\ 0.1 \end{bmatrix}, \quad \Sigma = \begin{bmatrix} 0.2 & + & + & \pm \\ + & 0.1 & - & - \\ + & - & 0.3 & + \\ \pm & - & + & 0.1 \end{bmatrix},$$

where a $+$ means the element is non-negative, a $-$ means the element is non-positive, and a $\pm$ means the element can be any real number. In **CVXR**, this semidefinite program is

```
R> Sigma <- Variable(n, n, PSD = TRUE)
R> obj <- t(w) %*% Sigma %*% w
R> constr <- list(Sigma[1, 1] == 0.2, Sigma[1, 2] >= 0, Sigma[1, 3] >= 0,
+    Sigma[2, 2] == 0.1, Sigma[2, 3] <= 0, Sigma[2, 4] <= 0,
+    Sigma[3, 3] == 0.3, Sigma[3, 4] >= 0, Sigma[4, 4] == 0.1)
R> prob <- Problem(Maximize(obj), constr)
R> result <- solve(prob, solver = "SCS")
```

Our result for this numerical case is

$$\Sigma = \begin{bmatrix} 0.2000 & 0.0967 & 0.0000 & 0.0762 \\ 0.0967 & 0.1000 & -0.1032 & 0.0000 \\ 0.0000 & -0.1032 & 0.3000 & 0.0041 \\ 0.0762 & 0.0000 & 0.0041 & 0.1000 \end{bmatrix}$$

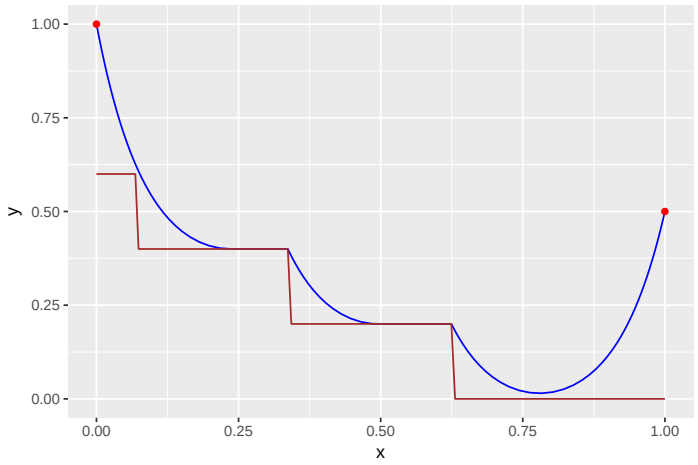


Figure 6: Solution of the catenary problem (blue) with a ground constraint (brown).

This example can be generalized to include arbitrary convex constraints on Σ. Furthermore, if we have a target estimate for the covariance, we can bound deviations from the target by incorporating penalized slack variables into our optimization problem.

Catenary problem

We consider a discretized version of the catenary problem in Griva and Vanderbei (2005). A chain with uniformly distributed mass hangs from the endpoints $(0, 1)$ and $(1, 1)$ on a 2-D plane. Gravitational force acts in the negative y direction. Our goal is to find the shape of the chain in equilibrium, which is equivalent to determining the (x, y) coordinates of every point along its curve when its potential energy is minimized.

To formulate this as an optimization problem, we parameterize the chain by its arclength and divide it into m discrete links. The length of each link must be no more than $h > 0$. Since mass is uniform, the total potential energy is simply the sum of the y-coordinates. Therefore, our problem is

$$\begin{array}{ll} \underset{x,y}{\text{minimize}} & \sum_{i=1}^{m} y_i \\ \text{subject to} & x_1 = 0, \quad y_1 = 1, \quad x_m = 1, \quad y_m = 1 \\ & (x_{i+1} - x_i)^2 + (y_{i+1} - y_i)^2 \leq h^2, \quad i = 1, \ldots, m-1 \end{array}$$

with variables $x \in \mathbf{R}^m$ and $y \in \mathbf{R}^m$. This basic catenary problem has a well-known analytical solution (Gelfand and Fomin 1963), which we can easily verify with **CVXR**.

```
R> x <- Variable(m)
R> y <- Variable(m)
R> obj <- sum(y)
R> constr <- list(x[1] == 0, y[1] == 0, x[m] == 1, y[m] == 1,
+    diff(x)^2 + diff(y)^2 <= h^2)
R> prob <- Problem(Minimize(obj), constr)
R> result <- solve(prob)
```

A more interesting situation arises when the ground is not flat. Let $g \in \mathbf{R}^m$ be the elevation vector (relative to the x-axis), and suppose the right endpoint of our chain has been lowered

by $\Delta y_m = 0.5$. The analytical solution in this case would be difficult to calculate. However, we need only add two lines to our constraint definition,

```
R> constr[[4]] <- (y[m] == 0.5)
R> constr <- c(constr, y >= g)
```

to obtain the new result. Figure 6 depicts the solution of this modified catenary problem for $m = 101$ and $h = 0.02$. The chain is shown hanging in blue, bounded below by the red staircase structure, which represents the ground.

Portfolio optimization

In this example, we solve the Markowitz portfolio problem under various different constraints (Markowitz 1952; Roy 1952; Lobo, Fazel, and Boyd 2007). We have n assets or stocks in our portfolio and must determine the amount of money to invest in each. Let w_i denote the fraction of our budget invested in asset $i = 1, \ldots, n$, and let r_i be the returns (i.e., fractional change in price) over the period of interest. We model returns as a random vector $r \in \mathbf{R}^n$ with known mean $\mathsf{E}[r] = \mu$ and covariance $\mathsf{VAR}(r) = \Sigma$. Thus, given a portfolio $w \in \mathbf{R}^n$, the overall return is $R = r^\top w$.

Portfolio optimization involves a trade-off between the expected return $\mathsf{E}[R] = \mu^\top w$ and associated risk, which we take as the return variance $\mathsf{VAR}(R) = w^\top \Sigma w$. Initially, we consider only long portfolios, so our problem is

$$\begin{array}{ll} \underset{w}{\text{maximize}} & \mu^\top w - \gamma w^\top \Sigma w \\ \text{subject to} & w \geq 0, \quad \sum_{i=1}^n w_i = 1, \end{array}$$

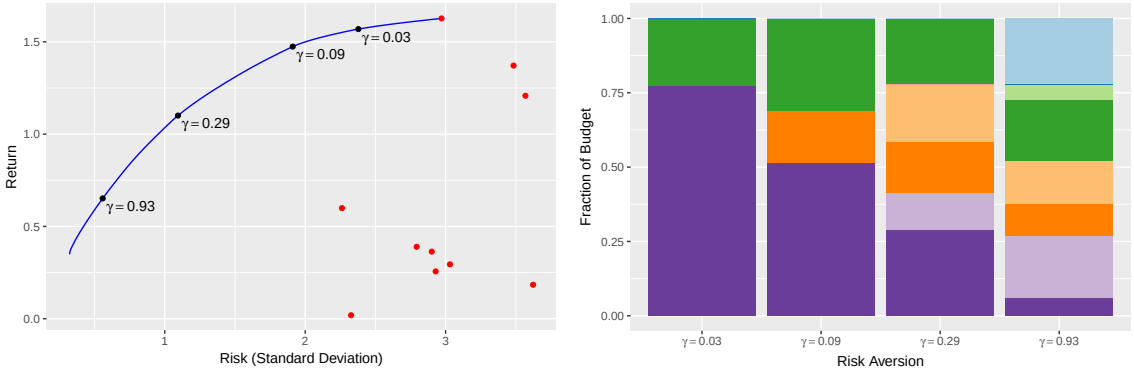
where the objective is the risk-adjusted return and $\gamma > 0$ is a risk aversion parameter.

```
R> w <- Variable(n)
R> ret <- t(mu) %*% w
R> risk <- quad_form(w, Sigma)
R> obj <- ret - gamma * risk
R> constr <- list(w >= 0, sum(w) == 1)
R> prob <- Problem(Maximize(obj), constr)
R> result <- solve(prob)
```

In this case, it is necessary to specify the quadratic form with `quad_form` rather than the usual `t(w) %*% Sigma %*% w` because the latter will be interpreted by the **CVXR** parser as a product of two affine terms and rejected for not being DCP. We can obtain the risk and return by directly evaluating the value of the separate expressions:

```
R> result$getValue(risk)
R> result$getValue(ret)
```

Figure 7a depicts the risk-return trade-off curve for $n = 10$ assets and μ and $\Sigma^{1/2}$ drawn from a standard normal distribution. The x-axis represents the standard deviation of the return. Red points indicate the result from investing the entire budget in a single asset. As γ increases, our portfolio becomes more diverse (Figure 7b), reducing risk but also yielding a lower return.



(a) Risk-return curve. (b) Asset portfolios.

Figure 7: (a) Risk-return trade-off curve for various γ. Portfolios that invest completely in one asset are plotted in red. (b) Fraction of budget invested in each asset.

Many variations on the classical portfolio problem exist. For instance, we could allow long and short positions, but impose a leverage limit $\|w\|_1 \leq L^{\max}$ by changing

```
R> constr <- list(p_norm(w, 1) <= Lmax, sum(w) == 1)
```

An alternative is to set a lower bound on the return and minimize just the risk. To account for transaction costs, we could add a term to the objective that penalizes deviations of w from the previous portfolio. These extensions and more are described in Boyd, Busseti, Diamond, Kahn, Koh, Nystrup, and Speth (2017). The key takeaway is that all of these convex problems can be easily solved in **CVXR** with just a few alterations to the code above.

Kelly gambling

In Kelly gambling (Kelly 1956), we are given the opportunity to bet on n possible outcomes, which yield a random non-negative return of $r \in \mathbf{R}^n_+$. The return r takes on exactly K values $r_1, \ldots, r_K$ with known probabilities $\pi_1, \ldots, \pi_K$. This gamble is repeated over T periods. In a given period t, let $b_i \geq 0$ denote the fraction of our wealth bet on outcome i. Assuming the nth outcome is equivalent to not wagering (it returns one with certainty), the fractions must satisfy $\sum_{i=1}^n b_i = 1$. Thus, at the end of the period, our cumulative wealth is $w_t = (r^\top b) w_{t-1}$. Our goal is to maximize the average growth rate with respect to $b \in \mathbf{R}^n$:

$$\begin{array}{ll} \underset{b}{\text{maximize}} & \sum_{j=1}^K \pi_j \log(r_j^\top b) \\ \text{subject to} & b \geq 0, \quad \sum_{i=1}^n b_i = 1. \end{array}$$

In the following code, `rets` is the K by n matrix of possible returns with rows r_j, while `ps` is the vector of return probabilities $(\pi_1, \ldots, \pi_K)$.

```
R> b <- Variable(n)
R> obj <- t(ps) %*% log(rets %*% b)
R> constr <- list(b >= 0, sum(b) == 1)
R> prob <- Problem(Maximize(obj), constr)
R> result <- solve(prob)
```

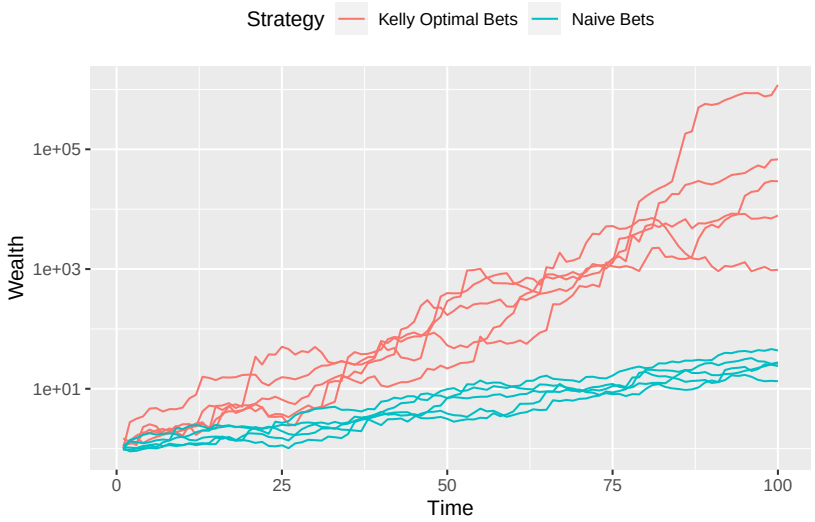


Figure 8: Wealth trajectories for the Kelly optimal bets (red) and naïve bets (cyan). The naïve betting scheme holds onto 15% of the wealth and splits the rest in direct proportion to the expected returns.

We solve the Kelly gambling problem for $K = 100$ and $n = 20$. The probabilities $\pi_j \sim \text{Uniform}(0, 1)$, and the potential returns $r_{ji} \sim \text{Uniform}(0.5, 1.5)$ except for $r_{jn} = 1$, which represents the payoff from not wagering. With an initial wealth of $w_0 = 1$, we simulate the growth trajectory of our Kelly optimal bets over $P = 100$ periods, assuming returns are i.i.d. over time.

```
R> bets <- result$getValue(b)
R> idx <- sample.int(K, size = P, probs = ps, replace = TRUE)
R> winnings <- rets[idx,] %*% bets
R> wealth <- w0 * cumprod(winnings)
```

For comparison, we also calculate the trajectory for a naïve betting scheme, which holds onto 15% of the wealth at the beginning of each period and divides the other 85% over the bets in direct proportion to their expect returns.

Growth curves for five independent trials are plotted in Figure 8. Red lines represent the wealth each period from the Kelly bets, while cyan lines are the result of the naïve bets. Clearly, Kelly optimal bets perform better, producing greater net wealth by the final period. However, as observed in some trajectories, wealth tends to drop by a significant amount before increasing eventually. One way to reduce this drawdown risk is to add a convex constraint as proposed in Busseti, Ryu, and Boyd (2016, Section 5.3),

$$\log\left(\sum_{j=1}^{K} \exp(\log \pi_j - \lambda \log(r_j^\top b))\right) \leq 0,$$

where $\lambda \geq 0$ is the risk-aversion parameter. With **CVXR**, this can be accomplished in a single line using the `log_sum_exp` atom. Other extensions like wealth goals, betting restrictions, and VaR/CVaR bounds are also readily incorporated.

Channel capacity

The following problem comes from an exercise in Boyd and Vandenberghe (2004, pp. 207–208). Consider a discrete memoryless communication channel with input $X(t) \in \{1, \ldots, n\}$ and output $Y(t) \in \{1, \ldots, m\}$ for $t = 1, 2, \ldots$. The relation between the input and output is given by a transition matrix $P \in \mathbf{R}_+^{m \times n}$ with

$$P_{ij} = \mathsf{P}(Y(t) = i | X(t) = j), \quad i = 1, \ldots, m, \quad j = 1, \ldots, n.$$

Assume that X has a probability distribution denoted by $x \in \mathbf{R}^n$, i.e., $x_j = \mathsf{P}(X(t) = j)$ for $j = 1, \ldots, n$. A famous result by Shannon and Weaver (1949) states that the channel capacity is found by maximizing the mutual information between X and Y,

$$I(X, Y) = \sum_{j=1}^{n} x_j \sum_{i=1}^{m} P_{ij} \log_2 P_{ij} - \sum_{i=1}^{m} y_i \log_2 y_i,$$

where $y = Px$ is the probability distribution of Y. Since I is concave, this is equivalent to solving the convex optimization problem

$$\begin{array}{ll} \underset{x,y}{\text{maximize}} & \sum_{j=1}^{n} x_j \sum_{i=1}^{m} P_{ij} \log P_{ij} - \sum_{i=1}^{m} y_i \log y_i \\ \text{subject to} & x \geq 0, \quad \sum_{i=1}^{m} x_i = 1, \quad y = Px \end{array}$$

for $x \in \mathbf{R}^n$ and $y \in \mathbf{R}^m$. The associated code in **CVXR** is

```
R> x <- Variable(n)
R> y <- P %*% x
R> c <- apply(P * log2(P), 2, sum)
R> obj <- t(c) %*% x + sum(entr(y))
R> constr <- list(sum(x) == 1, x >= 0)
R> prob <- Problem(Maximize(obj), constr)
R> result <- solve(prob)
```

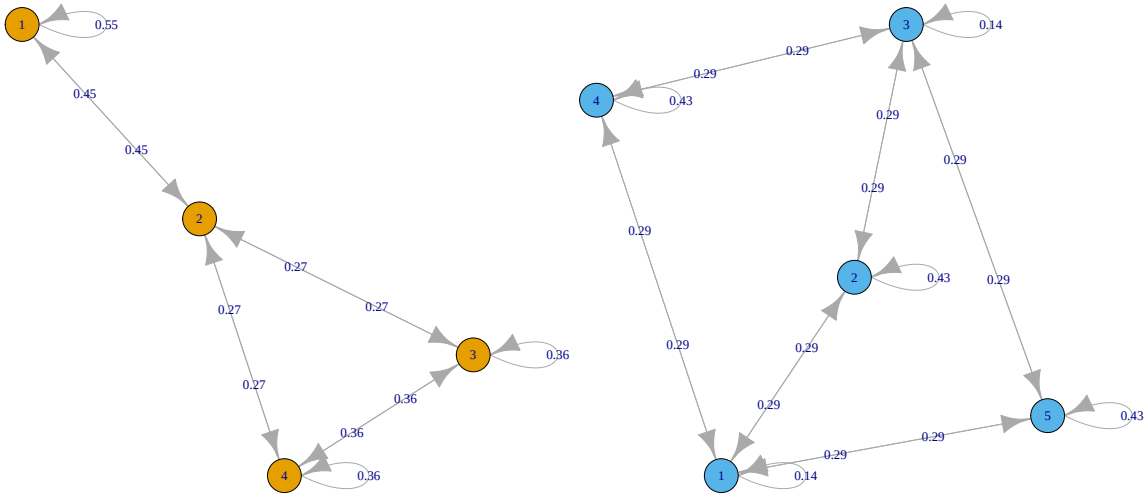
The channel capacity is simply the optimal objective, `result$value`.

Fastest mixing Markov chain

This example is derived from the results in Boyd, Diaconis, and Xiao (2004, Section 2). Let $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ be a connected graph with vertices $\mathcal{V} = \{1, \ldots, n\}$ and edges $\mathcal{E} \subseteq \mathcal{V} \times \mathcal{V}$. Assume that $(i, i) \in \mathcal{E}$ for all $i = 1, \ldots, n$, and $(i, j) \in \mathcal{E}$ implies $(j, i) \in \mathcal{E}$. Under these conditions, a discrete-time Markov chain on $\mathcal{V}$ will have the uniform distribution as one of its equilibrium distributions. We are interested in finding the Markov chain, i.e., constructing the transition probability matrix $P \in \mathbf{R}_+^{n \times n}$, that minimizes its asymptotic convergence rate to the uniform distribution. This is an important problem in Markov chain Monte Carlo (MCMC) simulations, as it directly affects the sampling efficiency of an algorithm.

The asymptotic rate of convergence is determined by the second largest eigenvalue of P, which in our case is $\mu(P) := \sigma_{\max}(P - \frac{1}{n}\mathbf{1}\mathbf{1}^\top)$ where $\sigma_{\max}(A)$ denotes the maximum singular value of A. As $\mu(P)$ decreases, the mixing rate increases and the Markov chain converges faster to equilibrium. Thus, our optimization problem is

$$\begin{array}{ll} \underset{P}{\text{minimize}} & \sigma_{\max}(P - \frac{1}{n}\mathbf{1}\mathbf{1}^\top) \\ \text{subject to} & P \geq 0, \quad P\mathbf{1} = \mathbf{1}, \quad P = P^\top \\ & P_{ij} = 0, \quad (i, j) \notin \mathcal{E}. \end{array}$$



(a) Triangle + 1 edge. (b) Bipartite 2 + 3.

Figure 9: Markov chains with transition probabilities that achieve the fastest mixing rate.

The element P_{ij} of our transition matrix is the probability of moving from state i to state j. Our assumptions imply that P is non-negative, symmetric, and doubly stochastic. The last constraint ensures transitions do not occur between unconnected vertices.

The function $\sigma_{\max}$ is convex, so this problem is solvable in **CVXR**. For instance, the code for the Markov chain in Figure 9a is

```
R> P <- Variable(n, n)
R> ones <- matrix(1, nrow = n, ncol = 1)
R> obj <- sigma_max(P - 1/n)
R> constr1 <- list(P >= 0, P %*% ones == ones, P == t(P))
R> constr2 <- list(P[1, 3] == 0, P[1, 4] == 0)
R> prob <- Problem(Minimize(obj), c(constr1, constr2))
R> result <- solve(prob, solver = "SCS")
```

where we have set $n = 4$. We could also have specified $P\mathbf{1} = \mathbf{1}$ with `sum_entries(P, 1) == 1`, which uses the `sum_entries` atom to represent the row sums.

It is easy to extend this example to other Markov chains. To change the number of vertices, we would simply modify `n`, and to add or remove edges, we need only alter the constraints in `constr2`. For instance, the bipartite chain in Figure 9b is produced by setting $n = 5$ and

```
R> constr2 <- list(P[1, 3] == 0, P[2, 4] == 0, P[2, 5] == 0, P[4, 5] == 0)
```

4. Implementation

CVXR represents the atoms, variables, constraints, and other parts of an optimization problem using S4 class objects. S4 enables us to overload standard mathematical operations so **CVXR** combines seamlessly with native R code and other packages. When an operation is invoked on a variable, a new object is created that represents the corresponding expression

tree with the operator as the root node and the arguments as leaves. This tree grows automatically as more elements are added, allowing us to encapsulate the structure of an objective function or constraint.

Once the user calls `solve`, DCP verification occurs. **CVXR** traverses the expression tree recursively, determining the sign and curvature of each sub-expression based on the properties of its component atoms. If the problem is deemed compliant, it is transformed into an equivalent cone program using graph implementations of convex functions (Grant *et al.* 2006). Then, **CVXR** passes the problem's description to the **CVXcanon** C++ library (Miller, Quigley, and Zhu 2015), which generates data for the cone program, and sends this data to the solver-specific R interface. The solver's results are returned to the user in a list. This object-oriented design and infrastructure were largely borrowed from **CVXPY**.

CVXR interfaces with the open-source cone solvers **ECOS** (Domahidi, Chu, and Boyd 2013) and **SCS** (O'Donoghue *et al.* 2016) through their respective R packages. **ECOS** is an interior-point solver, which achieves high accuracy for small and medium-sized problems, while **SCS** is a first-order solver that is capable of handling larger problems and semidefinite constraints. As noted by Domahidi *et al.* (2013, Section I.A), first-order methods can be slow if the problem is not well conditioned or if it has a feasible set that does not allow for an efficient projection, while interior-point methods have a convergence rate that is independent of the problem data and the particular feasible set. Furthermore, starting from version 0.99, **CVXR** also provides support for the commercial solvers **MOSEK** (Andersen and Andersen 2000) and **GUROBI** (**Gurobi** Optimization, Inc 2016) through binary R packages published by the respective vendors. It is not difficult to connect additional solvers so long as the solver has an API that can communicate with R. Users who wish to employ a custom solver may obtain the canonicalized data for a problem and solver combination directly with `get_problem_data(problem, solver)`. When more than one solver is capable of solving a problem, the `solver` argument to the `solve` function can be used to indicate a preference. Available solvers, depending on installed packages in a session, are returned via `installed_solvers()`. Interested users should consult tutorial examples on the web page `https://cvxr.rbind.io` for further guidance.

We have provided a large library of atoms, which should be sufficient to model most convex optimization problems. However, it is possible for a sophisticated user to incorporate new atoms into this library. The process entails creating a S4 class for the atom, overloading methods that characterize its DCP properties, and representing its graph implementation as a list of linear operators that specify the corresponding feasibility problem. For instance, the absolute value function $f(x) = |x|$ is represented by the '`Abs`' class, which inherits from '`Atom`'. We defined its curvature by overloading the S4 method `is_atom_convex`, used in the DCP verification step, to return `TRUE` when called on an '`Abs`' object. Then, we derived the graph form of the absolute value to be $f(x) = \inf\{t | -t \leq x \leq t\}$. This form's objective and constraints were coded into lists in the atom's `graph_implementation` function. A full mathematical exposition may be found in Grant *et al.* (2006, Section 10). In general, we suggest users try to reformulate their optimization problem first before attempting to add a novel atom.

4.1. Speed considerations

Usually, **CVXR** will be slower than a direct call to a solver, because in the latter case, the user would have already done the job of translating a mathematical problem into code and

constraints ingestible by the solver. **CVXR** does this translation for the user starting from a DCP formulation of the problem by walking the abstract syntax tree, which represents the canonicalized objectives and constraints, and building appropriate matrix structures for the solver. The matrix data are passed to a compatible solver using either **Rcpp** (Eddelbuettel and François 2011) or calls to a solver-specific R package. **CVXR** stores data in sparse matrices, thereby allowing large problems to be specified. However, the restrictions imposed by R on sparse matrices (Bates and Maechler 2019) still apply: each dimension cannot exceed the integer limit of $2^{31} - 1$.

Currently, the canonicalization and construction of data in R for the solver dominates computation time, particularly for complex expressions that involve indexing into individual elements of a matrix or vector. Using available **CVXR** functions for vectorized operations provides substantial speed improvements.

CVXR also provides a '`Parameter`' object that can be combined with warm starts, if such an option is available in the solver. A '`Parameter`' is a constant expression whose value can be modified after a '`Problem`' is created. This can yield significant reductions in computation time when solving a family of parametrized problems. The code below exploits warm starts to solve a lasso problem with two different values of the penalization parameter λ.

```
R> beta <- Variable(n)
R> lambda <- Parameter(pos = TRUE)
R> obj <- 0.5 * sum((y - X %*% beta)^2) + lambda * p_norm(beta, 1)
R> constr <- list(beta >= 0)
R> prob <- Problem(Minimize(obj), constr)
```

The problem is solved with a first value of `lambda`:

```
R> value(lambda) <- 1
R> result <- solve(prob, solver = "OSQP")
```

The problem is solved again with a second value of `lambda` and using a warm start:

```
R> value(lambda) <- 2
R> result <- solve(prob, solver = "OSQP", warm_start = TRUE)
```

On a commodity Macintosh laptop, with $X \in \mathbf{R}^{2000\times500}$ and $y \in \mathbf{R}^{2000}$, the first solution took 7.153 seconds, while the second took only 0.763 seconds.

5. Conclusion

Convex optimization plays an essential role in many fields, particularly machine learning and statistics. **CVXR** provides an object-oriented language with which users can easily formulate, modify, and solve a broad range of convex optimization problems. While other R packages may perform faster on a subset of these problems, **CVXR**'s advantage is its flexibility and simple intuitive syntax, making it an ideal tool for prototyping new models for which custom R code does not exist. For more information, see the official web page of the package on the Comprehensive R Archive Network (CRAN) at https://CRAN.R-project.org/package=CVXR and documentation.

Acknowledgments

The authors would like to thank Trevor Hastie, Robert Tibshirani, John Chambers, and David Donoho for their thoughtful advice and comments on this project. The authors also thank the referees for suggesting improvements and drawing our attention to some references. We are grateful to Steven Diamond, John Miller, and Paul Kunsberg Rosenfield for their contributions to the software's development. In particular, we are indebted to Steven for his work on **CVXPY**. Most of **CVXR**'s code, documentation, and examples were ported from his Python library.

Anqi Fu's research was supported by the Stanford Graduate Fellowship and DARPA X-DATA program. Balasubramanian Narasimhan's work was supported by the Clinical and Translational Science Award 1UL1 RR025744 for the Stanford Center for Clinical and Translational Education and Research (Spectrum) from the National Center for Research Resources, National Institutes of Health.

References

Andersen ED, Andersen KD (2000). "The **MOSEK** Interior Point Optimizer for Linear Programming: An Implementation of the Homogeneous Algorithm." In *High Performance Optimization*, pp. 197–232. Springer-Verlag.

Andersen M, Dahl J, Liu Z, Vandenberghe L (2011). "Interior-Point Methods for Large-Scale Cone Programming." In *Optimization for Machine Learning*. MIT. doi:10.7551/mitpress/8996.003.0005.

Banerjee O, Ghaoui LE, d'Aspremont A (2008). "Model Selection Through Sparse Maximum Likelihood Estimation for Multivariate Gaussian or Binary Data." *Journal of Machine Learning Research*, **9**, 485–516.

Bates D, Maechler M (2019). ***Matrix**: Sparse and Dense Matrix Classes and Methods*. R package version 1.2-18, URL https://CRAN.R-project.org/package=Matrix.

Bezanson J, Karpinski S, Shah VB, Edelman A (2012). "Julia: A Fast Dynamic Language for Technical Computing." arXiv:1209.5145 [cs.PL], URL http://arxiv.org/abs/1209.5145.

Boyd N, Hastie T, Boyd S, Recht B, Jordan MI (2018). "Saturating Splines and Feature Selection." *Journal of Machine Learning Reserach*, **18**(197), 1–32.

Boyd S, Busseti E, Diamond S, Kahn RN, Koh K, Nystrup P, Speth J (2017). "Multi-Period Trading via Convex Optimization." *Foundations and Trends in Optimization*, **3**(1), 1–76. doi:10.1561/2400000023.

Boyd S, Diaconis P, Xiao L (2004). "Fastest Mixing Markov Chain on a Graph." *SIAM Review*, **46**(4), 667–689. doi:10.1137/s0036144503423264.

Boyd S, Parikh N, Chu E, Peleato B, Eckstein J (2011). "Distributed Optimization and Statistical Learning via the Alternating Direction Method of Multipliers." *Foundations and Trends in Machine Learning*, **3**(1), 1–122. doi:10.1561/2200000016.

Boyd S, Vandenberghe L (2004). *Convex Optimization*. Cambridge University Press.

Busseti E, Ryu EK, Boyd S (2016). "Risk-Constrained Kelly Gambling." *Journal of Investing*, **25**(3), 118–134. doi:10.3905/joi.2016.25.3.118.

Canty A, Ripley B (2020). ***boot**: Bootstrap Functions (Originally by Angelo Canty for S)*. R package version 1.3-25, URL https://CRAN.R-project.org/package=boot.

Cox DR (1958). "The Regression Analysis of Binary Sequences." *Journal of the Royal Statistical Society B*, **20**(2), 215–242. doi:10.1111/j.2517-6161.1958.tb00292.x.

Diamond S, Boyd S (2016). "**CVXPY**: A Python-Embedded Modeling Language for Convex Optimization." *Journal of Machine Learning Research*, **17**(83), 1–5.

Domahidi A, Chu E, Boyd S (2013). "**ECOS**: An SOCP Solver for Embedded Systems." In *Proceedings of the European Control Conference*, pp. 3071–3076.

Dümbgen L, Rufibach K (2009). "Maximum Likelihood Estimation of a Log-Concave Density and Its Distribution Function: Basic Properties and Uniform Consistency." *Bernoulli*, **15**(1), 40–68. doi:10.3150/08-bej141.

Dümbgen L, Rufibach K (2011). "**logcondens**: Computations Related to Univariate Log-Concave Density Estimation." *Journal of Statistical Software*, **39**(6), 1–28. doi:10.18637/jss.v039.i06.

Eddelbuettel D, François R (2011). "**Rcpp**: Seamless R and C++ Integration." *Journal of Statistical Software*, **40**(8), 1–18. doi:10.18637/jss.v040.i08.

Freedman DA (2009). *Statistical Models: Theory and Practice*. Cambridge University Press.

Friedman J, Hastie T, Tibshirani R (2008). "Sparse Inverse Covariance Estimation with the Graphical Lasso." *Biostatistics*, **9**(3), 432–441. doi:10.1093/biostatistics/kxm045.

Friedman J, Hastie T, Tibshirani R (2010). "Regularization Paths for Generalized Linear Models via Coordinate Descent." *Journal of Statistical Software*, **33**(1), 1–22. doi:10.18637/jss.v033.i01.

Fu A, Narasimhan B, Kang DW, Diamond S, Miller J (2020). ***CVXR**: Disciplined Convex Optimization*. R package version 1.0-8, URL https://CRAN.R-project.org/package=CVXR.

Gelfand IM, Fomin SV (1963). *Calculus of Variations*. Prentice-Hall.

Grant M, Boyd S (2014). "**CVX**: MATLAB Software for Disciplined Convex Programming, Version 2.1." URL https://cvxr.com/cvx.

Grant M, Boyd S, Ye Y (2006). "Disciplined Convex Programming." In L Liberti, N Maculan (eds.), *Global Optimization: From Theory to Implementation*, Nonconvex Optimization and Its Applications, pp. 155–210. Springer-Verlag.

Griva IA, Vanderbei RJ (2005). "Case Studies in Optimization: Catenary Problem." *Optimization and Engineering*, **6**(4), 463–482. doi:10.1007/s11081-005-2068-0.

Gurobi Optimization, Inc (2016). ***Gurobi** Optimizer Reference Manual*. URL http://www.gurobi.com/.

Hastie T, Tibshirani R, Friedman J (2001). *The Elements of Statistical Learning*. Springer-Verlag, New York.

Hastie T, Zou H (2005). "Regularization and Variable Selection via the Elastic-Net." *Journal of the Royal Statistical Society B*, **67**(2), 301–320. doi:10.1111/j.1467-9868.2005.00503.x.

Huber PJ (1964). "Robust Estimation of a Location Parameter." *The Annals of Mathematical Statistics*, **35**(1), 73–101.

Johnson SG (2008). *The **NLopt** Nonlinear-Optimization Package*. URL http://ab-initio.mit.edu/nlopt/.

Kelly JL (1956). "A New Interpretation of Information Rate." *Bell System Technical Journal*, **35**(4), 917–926. doi:10.1109/TIT.1956.1056803.

Koenker R (2005). *Quantile Regression*. Cambridge University Press.

Koenker R, Mizera I (2014). "Convex Optimization in R." *Journal of Statistical Software*, **60**(5), 1–23. doi:10.18637/jss.v060.i05.

Lobo M, Boyd S (2000). "The Worst-Case Risk of a Portfolio." URL http://stanford.edu/~boyd/papers/pdf/risk_bnd.pdf.

Lobo MS, Fazel M, Boyd S (2007). "Portfolio Optimization with Linear and Fixed Transaction Costs." *Annals of Operations Research*, **152**(1), 341–365. doi:10.1007/s10479-006-0145-1.

Lofberg J (2004). "**YALMIP**: A Toolbox for Modeling and Optimization in MATLAB." In *Proceedings of the IEEE International Symposium on Computed Aided Control Systems Design*, pp. 294–289. URL http://users.isy.liu.se/johanl/yalmip.

Lumley T (2004). "Analysis of Complex Survey Samples." *Journal of Statistical Software*, **9**(8), 1–19. doi:10.18637/jss.v009.i08.

Lumley T (2020). "**survey**: Analysis of Complex Survey Samples." R package version 4.0, URL https://CRAN.R-project.org/package=survey.

Lumley TS (2010). *Complex Surveys: A Guide to Analysis Using R*. John Wiley & Sons.

Markowitz HM (1952). "Portfolio Selection." *Journal of Finance*, **7**(1), 77–91. doi:10.1111/j.1540-6261.1952.tb01525.x.

Miller J, Quigley P, Zhu J (2015). "**CVXcanon**: Automatic Canonicalization of Disciplined Convex Programs." URL https://stanford.edu/class/ee364b/projects/2015projects/reports/miller_quigley_zhu_report.pdf.

Mullen KM, Van Stokkum IHM (2012). ***nnls**: The Lawson-Hanson Algorithm for Non-Negative Least Squares (NNLS)*. R package version 1.4, URL https://CRAN.R-project.org/package=nnls.

Nash JC, Varadhan R (2011). "Unifying Optimization Algorithms to Aid Software System Users: **optimx** for R." *Journal of Statistical Software*, **43**(9), 1–14. doi:10.18637/jss.v043.i09.

O'Donoghue B, Chu E, Parikh N, Boyd S (2016). "Conic Optimization via Operator Splitting and Homogeneous Self-Dual Embedding." *Journal of Optimization Theory and Applications*, **169**(3), 1042–1068. doi:10.1007/s10957-016-0892-3.

R Core Team (2020). *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria. URL https://www.R-project.org/.

Roy AD (1952). "Safety First and the Holding of Assets." *Econometrica*, **20**(3), 431–449. doi:10.2307/1907413.

Shannon CE, Weaver W (1949). *The Mathematical Theory of Communication*. University of Illinois Press.

Skajaa A, Ye Y (2015). "A Homogeneous Interior-Point Algorithm for Nonsymmetric Convex Conic Optimization." *Mathematical Programming*, **150**(2), 391–422. doi:10.1007/s10107-014-0773-1.

The MathWorks Inc (2018). "Using Credit Scorecards with Constrained Logistic Regression Coefficients." URL https://www.mathworks.com/help/finance/examples/credit-scorecards-with-constrained-logistic-regression-coefficients.html.

The MathWorks Inc (2019). *MATLAB – The Language of Technical Computing, Version R2019a*. Natick. URL http://www.mathworks.com/products/matlab/.

Theußl S, Schwendinger F, Borchers HW (2020a). "CRAN Task View: Optimization and Mathematical Programming." Version 2020-05-21, URL https://CRAN.R-project.org/view=Optimization.

Theußl S, Schwendinger F, Hornik K (2020b). "**ROI**: The R Optimization Infrastructure Package." *Journal of Statistical Software*, **94**(15), 1–64. doi:10.18637/jss.v094.i15.

Tibshirani RJ, Hoefling H, Tibshirani R (2011). "Nearly-Isotonic Regression." *Technometrics*, **53**(1), 54–61. doi:10.1198/tech.2010.10111.

Udell M, Mohan K, Zeng D, Hong J, Diamond S, Boyd S (2014). "Convex Optimization in Julia." In *Proceedings of the Workshop for High Performance Technical Computing in Dynamic Languages*, pp. 18–28.

Van Rossum G, *et al.* (2011). *Python Programming Language*. URL https://www.python.org/.

Wright SJ (1997). *Primal-Dual Interior-Point Methods*. SIAM.

Yuan M, Lin Y (2007). "Model Selection and Estimation in the Gaussian Graphical Model." *Biometrika*, **94**(1), 19–35. doi:10.1093/biomet/asm018.

A. Expressions and functions

CVXR uses the function information in this section and the DCP tools to assign expressions a sign and curvature. In what follows, the domain $\mathbf{S}^n$ refers to the set of symmetric matrices, with $\mathbf{S}^n_+$ and $\mathbf{S}^n_-$ referring to the set of positive semidefinite and negative semidefinite matrices, respectively.

A.1. Operators

The infix operators `+`, `-`, `*`, `%*%`, `/` are treated as functions. Both `+` and `-` are affine functions. In **CVXR**, `*` and `/` are affine because `expr1 * expr2` and `expr1 %*% expr2` are allowed only when one of the expressions is constant and `expr1 / expr2` is allowed only when `expr2` is a scalar constant.

The transpose of any expression can be obtained using `t(expr)`. Transpose is an affine function. The construct `expr^p` is equivalent to the function `power(expr, p)`.

A.2. Indexing and slicing

All non-scalar expressions can be indexed using `expr[i, j]`. Indexing is an affine function. The syntax `expr[i]` can be used as a shorthand for `expr[i, 1]` when `expr` is a column vector. Similarly, `expr[i]` is shorthand for `expr[1, i]` when `expr` is a row vector.

Non-scalar expressions can also be sliced using the standard R slicing syntax. For example, `expr[i:j, r]` selects rows `i` through `j` of column `r` and returns a vector.

CVXR supports advanced indexing using lists of indices or boolean arrays. The semantics are the same as in R. Any time R might return a numeric vector, **CVXR** returns a column vector.

A.3. Scalar functions

CVXR provides the scalar functions displayed in Tables 2 and 3, which take in one or more scalars, vectors, or matrices as arguments and return a scalar.

For a vector expression `x`, `cvxr_norm(x)` and `cvxr_norm(x, 2)` give the Euclidean norm. For a matrix expression `X`, however, `cvxr_norm(X)` and `cvxr_norm(X, 2)` give the spectral norm. The function `cvxr_norm(X, "fro")` gives the Frobenius norm and `cvxr_norm(X, "nuc")` the nuclear norm. The nuclear norm can also be defined as the sum of the singular values of `X`.

The functions `max_entries` and `min_entries` give the largest and smallest entry, respectively, in a single expression. These functions should not be confused with `max_elemwise` and `min_elemwise` (see Section A.4). The functions `max_elemwise` and `min_elemwise` return the maximum or minimum of a list of scalar expressions.

The function `sum_entries` sums all the entries in a single expression. The built-in R `sum` should be used to add together a list of expressions. For example, the following code sums three expressions.

```
R> sum(expr1, expr2, expr3)
```

Some functions such as `sum_entries`, `cvxr_norm`, `max_entries`, and `min_entries` can be applied along an axis. Given an m by n expression `expr`, the line `func(expr, axis = 1)`

Function	Meaning	Domain	Sign	Curvature	Monotonicity

`geo_mean(x)`
`geo_mean(x, p)`
$p \in \mathbf{R}^n_+$, $p \neq 0$
	$x_1^{1/n} \cdots x_n^{1/n}$
	$(x_1^{p_1} \cdots x_n^{p_n})^{\frac{1}{\mathbf{1}^\top p}}$
		$x \in \mathbf{R}^n_+$	$+$	concave	$\nearrow$

`harmonic_mean(x)`	$\frac{n}{\frac{1}{x_1}+\cdots+\frac{1}{x_n}}$	$x \in \mathbf{R}^n_+$	$+$	concave	$\nearrow$
`lambda_max(X)`	$\lambda_{\max}(X)$	$X \in \mathbf{S}^n$	$\pm$	convex	none
`lambda_min(X)`	$\lambda_{\min}(X)$	$X \in \mathbf{S}^n$		concave	none
`lambda_sum_largest(X, k)` $k = 1, \ldots, n$	sum of k largest eigenvalues of X	$X \in \mathbf{S}^n$	$\pm$	convex	none
`lambda_sum_smallest(X, k)` $k = 1, \ldots, n$	sum of k smallest eigenvalues of X	$X \in \mathbf{S}^n$	$\pm$	concave	none
`log_det(X)`	$\log(\det(X))$	$X \in \mathbf{S}^n_+$	$\pm$	concave	none
`log_sum_exp(X)`	$\log\left(\sum_{ij} e^{X_{ij}}\right)$	$X \in \mathbf{R}^{m\times n}$	$\pm$	convex	$\nearrow$
`matrix_frac(x, P)`	$x^\top P^{-1} x$	$x \in \mathbf{R}^n$, $P \in \mathbf{S}^n_{++}$	$+$	convex	none
`max_entries(X)`	$\max_{ij}\{X_{ij}\}$	$X \in \mathbf{R}^{m\times n}$	same as X	convex	$\nearrow$
`min_entries(X)`	$\min_{ij}\{X_{ij}\}$	$X \in \mathbf{R}^{m\times n}$	same as X	concave	$\nearrow$
`mixed_norm(X, p, q)`	$\left(\sum_k \left(\sum_l \lvert x_{k,l}\rvert^p\right)^{q/p}\right)^{1/q}$	$X \in \mathbf{R}^{n\times n}$	$+$	convex	none
`cvxr_norm(x)` `cvxr_norm(x, 2)`	$\sqrt{\sum_i x_i^2}$	$X \in \mathbf{R}^n$	$+$	convex	$\nearrow$ for $x_i \geq 0$, $\swarrow$ for $x_i \leq 0$
`cvxr_norm(X, "fro")`	$\sqrt{\sum_{ij} X_{ij}^2}$	$X \in \mathbf{R}^{m\times n}$	$+$	convex	$\nearrow$ for $X_{ij} \geq 0$, $\swarrow$ for $X_{ij} \leq 0$
`cvxr_norm(X, 1)`	$\sum_{ij}\lvert X_{ij}\rvert$	$X \in \mathbf{R}^{m\times n}$	$+$	convex	$\nearrow$ for $X_{ij} \geq 0$, $\swarrow$ for $X_{ij} \leq 0$
`cvxr_norm(X, "inf")`	$\max_{ij}\{\lvert X_{ij}\rvert\}$	$X \in \mathbf{R}^{m\times n}$	$+$	convex	$\nearrow$ for $X_{ij} \geq 0$, $\swarrow$ for $X_{ij} \leq 0$
`cvxr_norm(X, "nuc")`	$\mathrm{tr}\left(\left(X^\top X\right)^{1/2}\right)$	$X \in \mathbf{R}^{m\times n}$	$+$	convex	none
`cvxr_norm(X)` `cvxr_norm(X, 2)`	$\sqrt{\lambda_{\max}\left(X^\top X\right)}$	$X \in \mathbf{R}^{m\times n}$	$+$	convex	none

Table 2: Scalar functions.

Function	Meaning	Domain	Sign	Curvature	Monotonicity
`p_norm(X, p)` $p \geq 1$ or $p = \infty$	$\|X\|_p = \left(\sum_{ij} \|X_{ij}\|^p\right)^{1/p}$	$X \in \mathbf{R}^{m \times n}$	$+$	convex	$\nearrow$ for $X_{ij} \geq 0$, $\swarrow$ for $X_{ij} \leq 0$
`p_norm(X, p)` $p < 1$, $p \neq 0$	$\|X\|_p = \left(\sum_{ij} X_{ij}^p\right)^{1/p}$	$X \in \mathbf{R}_+^{m \times n}$	$+$	concave	$\nearrow$
`quad_form(x, P)` constant $P \in \mathbf{S}_+^n$	$x^\top P x$	$x \in \mathbf{R}^n$	$+$	convex	$\nearrow$ for $x_i \geq 0$, $\swarrow$ for $x_i \leq 0$
`quad_form(x, P)` constant $P \in \mathbf{S}_-^n$	$x^\top P x$	$x \in \mathbf{R}^n$	$-$	concave	$\nearrow$ for $x_i \geq 0$, $\swarrow$ for $x_i \leq 0$
`quad_form(c, X)` constant $c \in \mathbf{R}^n$	$c^\top X c$	$X \in \mathbf{R}^{n \times n}$	depends on c, X	affine	depends on c
`quad_over_lin(X, y)`	$\left(\sum_{ij} X_{ij}^2\right)/y$	$x \in \mathbf{R}^n$, $y > 0$	$+$	convex	$\nearrow$ for $X_{ij} \geq 0$, $\swarrow$ for $X_{ij} \leq 0$, $\swarrow$ in y
`sum_entries(X)`	$\sum_{ij} X_{ij}$	$X \in \mathbf{R}^{m \times n}$	same as X	affine	$\nearrow$
`sum_largest(X, k)` $k = 1, 2, \ldots$	sum of k largest X_{ij}	$X \in \mathbf{R}^{m \times n}$	same as X	convex	$\nearrow$
`sum_smallest(X, k)` $k = 1, 2, \ldots$	sum of k smallest X_{ij}	$X \in \mathbf{R}^{m \times n}$	same as X	concave	$\nearrow$
`sum_squares(X)`	$\sum_{ij} X_{ij}^2$	$X \in \mathbf{R}^{m \times n}$	$+$	convex	$\nearrow$ for $X_{ij} \geq 0$, $\swarrow$ for $X_{ij} \leq 0$
`matrix_trace(X)`	$\mathrm{tr}\,(X)$	$X \in \mathbf{R}^{n \times n}$	same as X	affine	$\nearrow$
`tv(x)`	$\sum_i \|x_{i+1} - x_i\|$	$x \in \mathbf{R}^n$	$+$	convex	none
`tv(X)`	$\sum_{ij} \left\| \begin{bmatrix} X_{i+1,j} - X_{ij} \\ X_{i,j+1} - X_{ij} \end{bmatrix} \right\|_2$	$X \in \mathbf{R}^{m \times n}$	$+$	convex	none
`tv(X1,...,Xk)`	$\sum_{ij} \left\| \begin{bmatrix} X_{i+1,j}^{(1)} - X_{ij}^{(1)} \\ X_{i,j+1}^{(1)} - X_{ij}^{(1)} \\ \vdots \\ X_{i+1,j}^{(k)} - X_{ij}^{(k)} \\ X_{i,j+1}^{(k)} - X_{ij}^{(k)} \end{bmatrix} \right\|_2$	$X^{(i)} \in \mathbf{R}^{m \times n}$	$+$	convex	none

Table 3: More scalar functions.

Function	Meaning	Domain	Sign	Curvature	Monotonicity
`abs(x)`	$\lvert x\rvert$	$x \in \mathbf{R}$	$+$	convex	$\nearrow$ for $x \geq 0$, $\swarrow$ for $x \leq 0$
`entr(x)`	$-x\log(x)$	$x > 0$	$\pm$	concave	none
`exp(x)`	e^x	$x \in \mathbf{R}$	$+$	convex	$\nearrow$
`huber(x, M = 1)` $M \geq 0$	$\begin{cases} x^2 & \lvert x\rvert \leq M \\ 2M\lvert x\rvert - M^2 & \lvert x\rvert > M \end{cases}$	$x \in \mathbf{R}$	$+$	convex	$\nearrow$ for $x \geq 0$, $\swarrow$ for $x \leq 0$
`inv_pos(x)`	$1/x$	$x > 0$	$+$	convex	$\swarrow$
`kl_div(x, y)`	$x\log(x/y) - x + y$	$x > 0$, $y > 0$	$+$	convex	none
`log(x)`	$\log(x)$	$x > 0$	$\pm$	concave	$\nearrow$
`log1p(x)`	$\log(x+1)$	$x > -1$	same as x	concave	$\nearrow$
`logistic(x)`	$\log(1+e^x)$	$x \in \mathbf{R}$	$+$	convex	$\nearrow$
`max_elemwise(x1,..., xk)`	$\max\{x_1, \ldots, x_k\}$	$x_i \in \mathbf{R}$	$\max(\text{sign}(x_1))$	convex	$\nearrow$
`min_elemwise(x1,..., xk)`	$\min\{x_1, \ldots, x_k\}$	$x_i \in \mathbf{R}$	$\min(\text{sign}(x_1))$	concave	$\nearrow$
`multiply(c, x)` $c \in \mathbf{R}$	$c \times x$	$x \in \mathbf{R}$	$\text{sign}(cx)$	affine	depends on c
`neg(x)`	$\max\{-x, 0\}$	$x \in \mathbf{R}$	$+$	convex	$\swarrow$
`pos(x)`	$\max\{x, 0\}$	$x \in \mathbf{R}$	$+$	convex	$\nearrow$
`power(x, 0)`	1	$x \in \mathbf{R}$	$+$	constant	
`power(x, 1)`	x	$x \in \mathbf{R}$	same as x	affine	$\nearrow$
`power(x, p)` $p = 2, 4, 8, \ldots$	x^p	$x \in \mathbf{R}$	$+$	convex	$\nearrow$ for $x \geq 0$, $\swarrow$ for $x \leq 0$
`power(x, p)` $p < 0$	x^p	$x > 0$	$+$	convex	$\swarrow$
`power(x, p)` $0 < p < 1$	x^p	$x \geq 0$	$+$	concave	$\nearrow$
`power(x, p)` $p > 1,\ p \neq 2, 4, 8, \ldots$	x^p	$x \geq 0$	$+$	convex	$\nearrow$
`scalene(x, alpha, beta)` alpha ≥ 0, beta ≥ 0	$\alpha\text{pos}(x) + \beta\text{neg}(x)$	$x \in \mathbf{R}$	$+$	convex	$\nearrow$ for $x \geq 0$, $\swarrow$ for $x \leq 0$
`sqrt(x)`	$\sqrt{x}$	$x \geq 0$	$+$	concave	$\nearrow$
`square(x)`	x^2	$x \in \mathbf{R}$	$+$	convex	$\nearrow$ for $x \geq 0$, $\swarrow$ for $x \leq 0$

Table 4: Elementwise functions.

Function	Meaning	Domain	Sign	Curvature	Monotonicity
`bmat([[X11,..., X1q], ..., [Xp1,..., Xpq]])`	$\begin{bmatrix} X^{(1,1)} & \cdots & X^{(1,q)} \\ \vdots & & \vdots \\ X^{(p,1)} & \cdots & X^{(p,q)} \end{bmatrix}$	$X^{(i,j)} \in \mathbf{R}^{m_i \times n_j}$	sign $\left(\sum_{ij} X_{11}^{(i,j)}\right)$	affine	$\nearrow$
`conv(c, x)` $c \in \mathbf{R}^m$	$c * x$	$x \in \mathbf{R}^n$	sign $(c_1 x_1)$	affine	depends on c
`cumsum_axis(X, axis = 1)`	cumulative sum along given axis	$X \in \mathbf{R}^{m \times n}$	same as X	affine	$\nearrow$
`diag(x)`	$\begin{bmatrix} x_1 & & \\ & \ddots & \\ & & x_n \end{bmatrix}$	$x \in \mathbf{R}^n$	same as x	affine	$\nearrow$
`diag(X)`	$\begin{bmatrix} X_{11} \\ \vdots \\ X_{nn} \end{bmatrix}$	$X \in \mathbf{R}^{n \times n}$	same as X	affine	$\nearrow$
`diff(X, k = 1, axis = 1)` $k = 0, 1, 2, \ldots$	kth order differences (argument k is actually named *differences* and *lag* can also be used) along given axis	$X \in \mathbf{R}^{m \times n}$	same as X	affine	$\nearrow$
`hstack(X1,..., Xk)`	$\begin{bmatrix} X^{(1)} \cdots X^{(k)} \end{bmatrix}$	$X^{(i)} \in \mathbf{R}^{m \times n_i}$	sign $\left(\sum_i X_{11}^{(i)}\right)$	affine	$\nearrow$
`kronecker(C, X)` $C \in \mathbf{R}^{p \times q}$	$\begin{bmatrix} C_{11}X & \cdots & C_{1q}X \\ \vdots & & \vdots \\ C_{p1}X & \cdots & C_{pq}X \end{bmatrix}$	$X \in \mathbf{R}^{m \times n}$	sign $(C_{11}X_{11})$	affine	depends on C
`reshape_expr(X, c(m', n'))`	$X' \in \mathbf{R}^{m' \times n'}$	$X \in \mathbf{R}^{m \times n}$ $m'n' = mn$	same as X	affine	$\nearrow$
`vec(X)`	$x' \in \mathbf{R}^{mn}$	$X \in \mathbf{R}^{m \times n}$	same as X	affine	$\nearrow$
`vstack(X1,..., Xk)`	$\begin{bmatrix} X^{(1)} \\ \vdots \\ X^{(k)} \end{bmatrix}$	$X^{(i)} \in \mathbf{R}^{m_i \times n}$	sign $\left(\sum_i X_{11}^{(i)}\right)$	affine	$\nearrow$

Table 5: Vector and matrix functions.

applies `func` to each row, returning an m by 1 expression. The line `func(expr, axis = 2)` applies `func` to each column, returning a 1 by n expression. For example, the following code sums along the columns and rows of a matrix variable:

```
R> X <- Variable(5, 4)
R> row_sums <- sum_entries(X, axis = 1)   # Has size (5, 1)
R> col_sums <- sum_entries(X, axis = 2)   # Has size (1, 4)
```

CVXR ensures the implementation aligns with the `base::apply` function. The default in most cases is `axis = NA`, which treats an input matrix as one long vector, basically the same as `base::apply` with `MARGIN = c(1, 2)`. The exception is `cumsum_axis` (see Table 4), which cannot take `axis = NA` and will throw an error.

A.4. Elementwise functions

These functions operate on each element of their arguments and are displayed in Table 4. For example, if `X` is a 5 by 4 matrix variable, then `abs(X)` is a 5 by 4 matrix expression. Also, `abs(X)[1, 2]` is equivalent to `abs(X[1, 2])`.

Elementwise functions that take multiple arguments, e.g., `max_elemwise` and `multiply`, operate on the corresponding elements of each argument. For instance, if `X` and `Y` are both 3 by 3 matrix variables, then `max_elemwise(X, Y)` is a 3 by 3 matrix expression, where `max_elemwise(X, Y)[2, 1]` is equivalent to `max_elemwise(X[2, 1], Y[2, 1])`. Thus all arguments must have the same dimensions or be scalars, which are promoted appropriately.

A.5. Vector and matrix functions

These functions, shown in Table 5, take one or more scalars, vectors, or matrices as arguments and return a vector or matrix.

The input to `bmat` is a list of lists of **CVXR** expressions. It constructs a block matrix. The elements of each inner list are stacked horizontally, and then the resulting block matrices are stacked vertically.

The output of `vec(X)` is the matrix `X` flattened in column-major order into a vector.

The output of `reshape_expr(X, c(m1, n1))` is the matrix `X` cast into an `m1` by `n1` matrix. The entries are taken from `X` in column-major order and stored in the output in column-major order.

Affiliation:

Anqi Fu, Stephen Boyd
Department of Electrical Engineering
David Packard Building
350 Jane Stanford Way
Stanford, CA 94305, United States of America
E-mail: anqif@stanford.edu, boyd@stanford.edu
URL: https://web.stanford.edu/~anqif/, https://web.stanford.edu/~boyd/

Balasubramanian Narasimhan
Department of Biomedical Data Sciences
and
Department of Statistics
Stanford University
390 Jane Stanford Way
Stanford, CA 94305, United States of America
E-mail: naras@stanford.edu
URL: https://web.stanford.edu/~naras/

Journal of Statistical Software http://www.jstatsoft.org/
published by the Foundation for Open Access Statistics http://www.foastat.org/
August 2020, Volume 94, Issue 14 *Submitted:* 2017-12-01
doi:10.18637/jss.v094.i14 *Accepted:* 2019-04-02