CVXR: An R Package for Disciplined Convex Optimization

Anqi Fu, Balasubramanian Narasimhan, and Stephen Boyd

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 2018). Our package, CVXR, 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, CVXPY (Diamond and Boyd 2016) in Python, and Convex.jl (Udell, Mohan, Zeng, Hong, Diamond, and Boyd 2014) in Julia. 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 $X \in \mathbb{R}^{m \times n}$ and $y \in \mathbb{R}^m$, and we want to solve the ordinary least squares (OLS) problem

$$
\min_{\beta} \|y - X\beta\|_2^2
$$

with optimization variable $\beta \in \mathbb{R}^n$. 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 $\beta$ using the code

```r
beta <- Variable(n)
obj <- sum((y - X %*% beta)^2)
```

```r
```
prob <- Problem(Minimize(obj))
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 are retrieved with

result$value # Optimal objective
result$getValue(beta) # Optimal variables
result$metrics # Runtime metrics

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:

\[
\min_{\beta} \|y - X\beta\|_2^2 \\
\text{subject to } \beta_j \leq \beta_{j+1}, \quad j = 1, \ldots, n - 1.
\]

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 or write our own custom solver. With CVXR though, we need only add the constraint as a second argument to the problem:

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

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 3.2.3.

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 and Borchers 2014). 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 2017) 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 CRAN and the official CVXR site.

2. Disciplined Convex Optimization

The general convex optimization problem is of the form

\[
\begin{aligned}
\text{minimize} & \quad f_0(v) \\
\text{subject to} & \quad f_i(v) \leq 0, \quad i = 1, \ldots, M \\
& \quad Av = b,
\end{aligned}
\]

where \( v \in \mathbb{R}^n \) is our variable of interest, and \( A \in \mathbb{R}^{m \times n} \) and \( b \in \mathbb{R}^n \) 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 \( f_i : \mathbb{R}^n \to \mathbb{R} \) that satisfy

\[
f_i(\theta u + (1 - \theta)v) \leq \theta f_i(u) + (1 - \theta)f_i(v)
\]

for all \( u, v \in \mathbb{R}^n \) 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,

\[
f(g_1, \ldots, g_k) \text{ is convex if } f \text{ is convex and for each } i = 1, \ldots, k, \text{ either}
\]

- \( g_i \) 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 \(x_i \in \mathbb{R}^n\) and \(y_i \in \mathbb{R}\). We represent these observations in matrix form as \(X \in \mathbb{R}^{m \times n}\) with stacked rows \(x_i^T\) and \(y \in \mathbb{R}^m\). 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

\[
\min_{\beta} \sum_{i=1}^{m} \phi(y_i - x_i^T \beta)
\]

for variable \(\beta \in \mathbb{R}^n\), 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)
obj <- sum(huber(y - X %*% beta, M))
prob <- Problem(Minimize(obj))
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 }
obj <- sum(quant_loss(y - X %*% beta, t))
prob <- Problem(Minimize(obj))
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 (H. Zou 2005; Friedman, Hastie, and Tibshirani 2010),

\[
\min_{\beta} \frac{1}{2m} \| y - X\beta \|_2^2 + \lambda (\frac{1-\alpha}{2} \| \beta \|_2^2 + \alpha \| \beta \|_1).
\]

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)
obj <- loss + elastic_reg(beta, lambda, alpha)
prob <- Problem(Minimize(obj))
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,
loss <- 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 \( y_i | x_i \sim \text{Bernoulli}(g_{\beta}(x_i)) \), where \( g_{\beta}(x) = \frac{1}{1 + e^{-x^T \beta}} \) is the logistic function, and maximize the log-likelihood function, yielding the optimization problem

\[
\maximize_{\beta} \sum_{i=1}^{m} \{ y_i \log(g_{\beta}(x_i)) + (1 - y_i) \log(1 - g_{\beta}(x_i)) \}.
\]

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

\[
\text{obj} <- -\text{sum}(X[y == 0,] %*% beta) - \text{sum}(\text{logistic}(-X %*% beta))
\]

\[
\text{prob} <- \text{Problem}(\text{Maximize}(\text{obj}))
\]

\[
\text{result} <- \text{solve}(\text{prob})
\]

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 \text{result$getValue}. For instance, the log-odds are

\[
\text{log_odds} <- \text{result$getValue}(X %*% beta)
\]

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

\[
\text{beta_res} <- \text{result$getValue}(\text{beta})
\]

\[
\text{y_probs} <- 1 / (1 + \exp(-X %*% beta_res))
\]

\[
\log(\frac{\text{y_probs} \, / \, (1 - \text{y_probs})}{})
\]

We illustrate with a logistic regression fit from a credit scoring example (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 \( \beta_2 \) and \( \beta_6 \):

\[
|\beta_2 - \beta_6| < 0.5.
\]  

(1)

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

\[
\text{constr} <- \text{list}(\beta[2:10] >= 0, \beta[2:10] <= 1, \text{abs}(\beta[2] - \beta[6]) < 0.05)
\]

\[
\text{result} <- \text{solve}(\text{prob}, \text{constr})
\]

\[
\text{beta_res} <- \text{result$getValue}(\text{beta})
\]
Figure 1: Logistic regression with constraints using data from Mathworks Inc (2018). The addition of constraint 1 moves the coefficients for customer age and customer income closer to each other.

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 an $l_2$-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 $\Sigma \in S^n_+$, the set of symmetric positive semidefinite matrices, has a sparse inverse $S = \Sigma^{-1}$. Let $Q = \frac{1}{m-1} \sum_{i=1}^m (x_i - \bar{x})(x_i - \bar{x})^T$ be our sample covariance. One way to estimate $\Sigma$ is to maximize the log-likelihood with an $l_1$-norm constraint (Yuan and Lin 2007; Banerjee, El Ghaoui, and d’Aspremont 2008; Friedman, Hastie, and Tibshirani 2008), which amounts to the optimization problem

$$\maximize \quad \log \det(S) - \text{tr}(S Q)$$

subject to

$$S \in S^n_+, \quad \sum_{i=1}^n \sum_{j=1}^n |S_{ij}| \leq \alpha.$$  

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

$S <- \text{Semidef}(n)$
The `Semidef` 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 `Semidef` 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 \( \alpha \), 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 (2016). 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.

For simplicity, consider the univariate case with \( n = 1 \). Assume we are given knots \( t_1 < t_2 < \cdots < t_k \) where each \( t_j \in \mathbb{R} \). 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

\[
\begin{align*}
\text{minimize} & \quad \sum_{i=1}^m \ell(y_i, f(x_i)) + \lambda \|w\|_1 \\
\text{subject to} & \quad \sum_{j=1}^k w_j = 0
\end{align*}
\]

for variables \((w_0, w) \in \mathbb{R} \times \mathbb{R}^k\). The function \( \ell: \mathbb{R} \times \mathbb{R} \to \mathbb{R} \) 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 \([t_1, t_k]\).

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
}
```
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 white regions indicate where $S_{ij} = 0$. 
(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).

```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()
w <- Variable(k)
```

```r
loss <- sum(loss.obs(y, f_est(X, knots, w0, w)))
reg <- lambda * p_norm(w, 1)
obj <- loss + reg
constr <- list(sum(w) == 0)
prob <- Problem(Minimize(obj), constr)
result <- solve(prob)
```

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

```r
w0s <- result$getValue(w0)
wss <- result$getValue(w)
```

We plot the fitted saturating hinges in Figure 3a. As expected, when $\lambda$ 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 in Section 3.1.1, 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 $K \in \mathbb{Z}_+$. Define $p_k := P(X = k)$ 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 (D"umbgen and Rufibach 2009), i.e.,

\[
\begin{align*}
\max_p & \quad \sum_{k=0}^{K} M_k \log p_k \\
\text{subject to} & \quad p \geq 0, \quad \sum_{k=0}^{K} p_k = 1, \\
& \quad p_k \geq \sqrt{p_{k-1}p_{k+1}}, \quad k = 1, \ldots, K - 1,
\end{align*}
\]

where $p \in \mathbb{R}^{K+1}$ 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

\[
\begin{align*}
\max_u & \quad \sum_{k=0}^{K} M_k u_k \\
\text{subject to} & \quad \sum_{k=0}^{K} e^{u_k} \leq 1, \\
& \quad u_k - u_{k-1} \geq u_{k+1} - u_k, \quad k = 1, \ldots, K - 1.
\end{align*}
\]

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

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

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

```r
pmf <- result$getValue(exp(u))
```

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

As an example, we consider the reliability data from D"umbgen and Rufibach (2011) that was collected as part of a consulting project at the Institute for Mathematical Statistics and Actuarial Science, University of Bern (D"umbgen 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 \texttt{logcondens}. Figure 4 shows that the curve obtained from the \texttt{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 \(X \in \mathbb{R}^{m \times n}\) be the matrix of survey samples, with each column corresponding to an auxiliary variable. Reweighting can be expressed as the optimization problem

\[
\begin{align*}
\text{minimize} & \quad \sum_{i=1}^{m} d_i \phi(g_i) \\
\text{subject to} & \quad A^T g = r
\end{align*}
\]

with respect to \(g \in \mathbb{R}^m\), where \(\phi : \mathbb{R} \to \mathbb{R}\) is a strictly convex function with \(\phi(1) = 0, r \in \mathbb{R}^n\) are the known population totals of the auxiliary variables, and \(A \in \mathbb{R}^{m \times n}\) 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 \texttt{survey} package (Lumley 2018), which also supplies facilities for calibration via the function \texttt{calibrate}. Both the population dataset (\texttt{apipop}) and a simple random sample of \(m = 200\) (\texttt{apisrs}) are provided. Suppose that we wish to reweight the observations in the sample using known totals for two variables from the population: \texttt{sctype}, the school type (elementary, middle or high) and \texttt{sch.wide}, whether the school met the yearly target or not. This reweighting would make the sample more representative of the general population.
Table 1: Raking weight estimates with \texttt{survey} package and \texttt{CVXR} for California Academic Performance Index data.

<table>
<thead>
<tr>
<th>School Type</th>
<th>Target Met?</th>
<th>survey</th>
<th>Weight</th>
<th>Frequency</th>
<th>CVXR</th>
<th>Weight</th>
<th>Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>E</td>
<td>Yes</td>
<td>29.00</td>
<td>15</td>
<td></td>
<td>29.00</td>
<td>15</td>
<td></td>
</tr>
<tr>
<td>H</td>
<td>No</td>
<td>31.40</td>
<td>13</td>
<td></td>
<td>31.40</td>
<td>13</td>
<td></td>
</tr>
<tr>
<td>M</td>
<td>Yes</td>
<td>29.03</td>
<td>9</td>
<td></td>
<td>29.03</td>
<td>9</td>
<td></td>
</tr>
<tr>
<td>E</td>
<td>No</td>
<td>28.91</td>
<td>127</td>
<td></td>
<td>28.91</td>
<td>127</td>
<td></td>
</tr>
<tr>
<td>H</td>
<td>Yes</td>
<td>31.50</td>
<td>12</td>
<td></td>
<td>31.50</td>
<td>12</td>
<td></td>
</tr>
<tr>
<td>M</td>
<td>No</td>
<td>31.53</td>
<td>24</td>
<td></td>
<td>31.53</td>
<td>24</td>
<td></td>
</tr>
</tbody>
</table>

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

```r
m <- nrow(apisrs)
di <- apisrs$pw
formula <- ~stype + sch.wide
r <- apply(model.matrix(object = formula, data = apipop), 2, sum)
X <- model.matrix(object = formula, data = apisrs)
A <- di * X

g <- Variable(m)
obj <- sum(di*(-entr(g) - g + 1))
constr <- list(t(A) %*% g == r)
prob <- Problem(Minimize(obj), constr)
result <- solve(prob)
w_cvxr <- di * result$getValue(g)
```

Table 1 shows that the results are identical to those obtained from \texttt{survey}. \texttt{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 $y \in \mathbb{R}^m$, Tibshirani, Hoefling, and Tibshirani (2011) fit a nearly-isotonic approximation $\beta \in \mathbb{R}^m$ by solving

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

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

```r
neariso_fit <- function(y, lambda) {
    m <- length(y)
    beta <- Variable(m)
    penalty <- sum(pos(diff(beta)))
    obj <- 0.5 * sum(((y - beta)^2) + lambda * penalty
```
In CVXR
\(\beta\)
This replaces the first difference term with an approximation to the second derivative at \(\lambda\).

\[
\text{penalty} <- \text{sum}(\text{pos}(\text{diff}(x, \text{differences} = 2)))
\]

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 \texttt{near.fit} with the \texttt{boot} library, we can obtain the standard errors and confidence intervals for our estimate in just a few lines of code.

\begin{verbatim}
neariso_fit_stat <- function(data, index, lambda) {
  sample <- data[index,] # Bootstrap sample of rows
  sample <- sample[order(sample$year),] # Order ascending by year
  near_fit(sample$annual, lambda)
}
boot.out <- boot(CDIAC, neariso_fit_stat, R = 999, lambda = 0.44)
\end{verbatim}

Figure 5a shows a nearly-isotonic fit with \(\lambda = 0.44\) and 95% normal confidence bands, which were generated using \(R = 999\) 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:

\[
\text{minimize } \frac{1}{2} \sum_{i=1}^{m} (y_i - \beta_i)^2 + \lambda \sum_{i=1}^{m-2} (\beta_i - 2\beta_{i+1} + \beta_{i+2})_+
\]

This replaces the first difference term with an approximation to the second derivative at \(\beta_{i+1}\).

In CVXR, the only change necessary is the penalty line in \texttt{near_fit},

\[
\text{penalty} <- \text{sum}(\text{pos}(\text{diff}(x, \text{differences} = 2)))
\]
The resulting curve is depicted in Figure 5b with 95% confidence bands generated from $R = 100$ 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 $y = w^T x = \sum_{i=1}^m w_i x_i$, where $w \in \mathbb{R}^m$ is a given vector of weights. We are provided limited information on the elements of $\Sigma$. For example, we may know the specific value or sign of certain $\Sigma_{jk}$, which are represented by upper and lower bound matrices $L$ and $U \in \mathbb{R}^{n \times n}$, 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

$$\begin{align*}
\text{maximize} & \quad w^T \Sigma w \\
\text{subject to} & \quad \Sigma \in \mathcal{S}_+^n, \quad L_{jk} \leq \Sigma_{jk} \leq U_{jk}, \quad j, k = 1, \ldots, n.
\end{align*}$$

Consider the specific case

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

where a + means the element is nonnegative, a − means the element is nonpositive, and a ± means the element can be any real number. In CVXR, this semidefinite program is

```r
Sigma <- Semidef(n)
obj <- quad_form(w, Sigma)
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)
prob <- Problem(Maximize(obj), constr)
result <- solve(prob)
```

It is necessary to specify a 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. Our result for this numerical case is

$$\Sigma = \begin{bmatrix}
0.1999 & 0.0973 & 0.0006 & 0.0743 \\
0.0973 & 0.0998 & -0.1012 & 0.0000 \\
0.0006 & -0.1012 & 0.3000 & 0.0005 \\
0.0743 & 0.0000 & 0.0005 & 0.1001
\end{bmatrix}$$

This example can be generalized to include arbitrary convex constraints on $\Sigma$. 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

\[
\begin{align*}
\text{minimize} & \quad \sum_{i=1}^{m} y_i \\
\text{subject to} & \quad x_1 = 0, \quad y_1 = 1, \quad x_m = 1, \quad y_m = 1 \\
& \quad (x_{i+1} - x_i)^2 + (y_{i+1} - y_i)^2 \leq h^2, \quad i = 1, \ldots, m - 1
\end{align*}
\]

with variables \(x \in \mathbb{R}^m\) and \(y \in \mathbb{R}^m\). 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)
y <- Variable(m)
obj <- sum(y)
constr <- c(x[1] == 0, y[1] == 0, x[m] == 1, y[m] == 1,
           (x[i+1] - x[i])^2 + (y[i+1] - y[i])^2 <= h^2, i = 1, \ldots, m - 1)
prob <- Problem(Minimize(obj), constr)
result <- solve(prob)
```

A more interesting situation arises when the ground is not flat. Let \(g \in \mathbb{R}^m\) 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,
constr[[4]] <- (y[m] == 0.5)
constr <- c(constr, y >= g)

Figure 6 depicts the solution of this modified catenary problem for \(m = 101\) and \(h = 0.04\). 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 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 \(i = 1, \ldots, m\), 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 \(r \in \mathbb{R}^n\) with known mean \(E[r] = \mu\) and covariance \(\text{Var}(r) = \Sigma\). Thus, given a portfolio \(w \in \mathbb{R}^n\), the overall return is \(R = r^T w\).

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

\[
\begin{align*}
\text{maximize} & \quad \mu^T w - \gamma w^T \Sigma w \\
\text{subject to} & \quad w \geq 0, \quad \sum_{i=1}^n w = 1
\end{align*}
\]

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

\[
w <- \text{Variable}(n) \\
\text{ret} <- t(mu) \times w \\
\text{risk} <- \text{quad_form}(w, Sigma) \\
\text{obj} <- \text{ret} - \gamma \text{risk} \\
\text{constr} <- \text{list}(w >= 0, \text{sum}(w) == 1) \\
\text{prob} <- \text{Problem}(\text{Maximize(obj), constr}) \\
\text{result} <- \text{solve(prob)}
\]

We can obtain the risk and return by directly evaluating the value of the separate expressions:

\[
\text{result}\$\text{getValue(risk)} \\
\text{result}\$\text{getValue(ret)}
\]

Figure 7a depicts the risk-return trade-off curve for \(n = 10\) assets and \(\mu\) 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 \(\gamma\) increases, our portfolio becomes more diverse (Figure 7b), reducing risk but also yielding a lower return.

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

\[
\text{constr} <- \text{list(p_norm(w,1) <= Lmax, sum(w) == 1)}
\]
Figure 7: (a) Risk-return trade-off curve for various $\gamma$. Portfolios that invest completely in one asset are plotted in red. (b) Fraction of budget invested in each asset.

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 $r \in \mathbb{R}_+^n$. 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 $n$th 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 $w_t = (r^T b) w_{t-1}$.

Our goal is to maximize the average growth rate with respect to $b \in \mathbb{R}^n$:

$$\max_b \sum_{j=1}^K \pi_j \log(r_j^T b)$$

subject to $b \geq 0$, $\sum_{i=1}^n b_i = 1$.

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

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

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 $r_j \sim \text{Uniform}(0.5,1.5)$ except for $r_n = 1$, which
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.

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)
idx <- sample.int(K, size = P, probs = ps, replace = TRUE)
winnings <- rets[idx,] %*% bets
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 naive 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),

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

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 $P \in \mathbb{R}^{m \times n}$ with

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

Assume that $X$ has a probability distribution denoted by $x \in \mathbb{R}^n$, i.e., $x_j = P(X(t) = j)$ 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

$$\begin{align*}
\text{maximize} & \quad \sum_{j=1}^n x_j \sum_{i=1}^m P_{ij} \log_2 P_{ij} - \sum_{i=1}^m y_i \log_2 y_i \\
\text{subject to} & \quad x \geq 0, \quad \sum_{i=1}^n x_i = 1, \quad y = Px
\end{align*}$$

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

```r
x <- Variable(n)
y <- P %*% x
c <- apply(P * log2(P), 2, sum)
obj <- c %*% x + sum(entr(y))
constr <- list(sum(x) == 1, x >= 0)
prob <- Problem(Maximize(obj), constr)
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 $G = (V,E)$ be a connected graph with vertices $V = \{1, \ldots, n\}$ and edges $E \subseteq V \times V$. Assume that $(i,i) \in E$ for all $i = 1, \ldots, n$, and $(i,j) \in E$ implies $(j,i) \in E$. Under these conditions, a discrete-time Markov chain on $V$ 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 $P \in \mathbb{R}^{n \times n}$, 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 $\mu(P) := \lambda_{\max}(P - \frac{1}{n}11^T)$ where $\lambda_{\max}(A)$ denotes the maximum eigenvalue of $A$. As $\mu(P)$ decreases, the mixing rate increases and the Markov chain converges faster to equilibrium. Thus, our optimization problem is

$$\begin{align*}
\text{minimize} & \quad \lambda_{\max}(P - \frac{1}{n}11^T) \\
\text{subject to} & \quad P \geq 0, \quad P1 = 1, \quad P = P^T \\
P_{ij} = 0, & \quad (i,j) \notin E.
\end{align*}$$
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 nonnegative, symmetric, and doubly stochastic. The last constraint ensures transitions do not occur between unconnected vertices.

(a) Triangle + 1 Edge

(b) Bipartite 2 + 3

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

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

\begin{verbatim}
P <- Variable(n,n)
one <- matrix(1, nrow = n, ncol = 1)
obj <- Minimize(lambda_max(P - 1/n))
constr1 <- list(P >= 0, P %*% ones == ones, P == t(P))
constr2 <- list(P[1,3] == 0, P[1,4] == 0)
prob <- Problem(obj, c(constr1, constr2))
result <- solve(prob)
\end{verbatim}

where we have set $n = 4$. We could also have specified $P1 = 1$ with \texttt{sum_entries(P,1)} = 1, which uses the \texttt{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 \texttt{constr2}. For instance, the bipartite chain in Figure 9b is produced by setting $n = 5$ and

\begin{verbatim}
constr2 <- list(P[1,3] == 0, P[2,4] == 0, P[2,5] == 0, P[4,5] == 0)
\end{verbatim}

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 script 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. Both solvers run single-threaded at present. Support for multi-threaded SCS will be added in the future. Furthermore, CVXR version 0.99 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 CVXR site 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 performs 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 2018) 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 access individual elements by indexing into matrices or vectors. Using available CVXR functions for vectorized operations provides substantial speed improvements. Gao and Shi (2018) present speed and accuracy comparisons between different convex optimization packages on the classifier-Lasso (C-Lasso) high-dimensional estimator proposed by Su, Shi, and Phillips (2016). Of the included packages – nloptr, Rmosek (MOSEK ApS 2017), CVX, and CVXR – the authors note that only the latter three obtained correct results. The timing results in Gao and Shi (2018, Table 1) show Rmosek to be the fastest, with a hundred-fold speedup over CVXR. However, if we reformulate the objective in Gao (2018) using matrix operations and CVXR atoms, this advantage is reduced to a factor between 2.5 and 3.5. In fact, since the problem is known to be convex, following our Getting Faster Results tutorial further decreases the advantage to between 1.5 and 1.8, as shown in Table 2.

Table 2: Timing results per Gao and Shi (2018) for computing the C-Lasso estimator of Su et al. (2016). For CVXR, the objective was reformulated using matrix functions. The “nodecp” version skips DCP checks and directly constructs the matrix data structures. The parameters are \( n \), the number of cross sectional units and \( T \) the time length. The results for the \((n, T) = (200, 50)\) could not be replicated. We used 100 replications rather than the original 500. The experiments were run on a Macbook Pro equipped with a 4-core Intel Core i7 processor and 16 GB of RAM.

<table>
<thead>
<tr>
<th>( (n, T) )</th>
<th>((100, 15))</th>
<th>((100, 25))</th>
<th>((100, 50))</th>
<th>((200, 15))</th>
<th>((200, 25))</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>RMSE</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RMosek</td>
<td>0.0662</td>
<td>0.0410</td>
<td>0.0251</td>
<td>0.0414</td>
<td>0.0267</td>
</tr>
<tr>
<td>CVXR</td>
<td>0.0556</td>
<td>0.0382</td>
<td>0.0265</td>
<td>0.0431</td>
<td>0.0284</td>
</tr>
<tr>
<td>CVXR (nodcp)</td>
<td>0.0999</td>
<td>0.0352</td>
<td>0.0257</td>
<td>0.0441</td>
<td>0.0286</td>
</tr>
<tr>
<td><strong>Correct Ratio</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RMosek</td>
<td>0.8963</td>
<td>0.9650</td>
<td>0.9971</td>
<td>0.9041</td>
<td>0.9666</td>
</tr>
<tr>
<td>CVXR</td>
<td>0.9039</td>
<td>0.9664</td>
<td>0.9968</td>
<td>0.9023</td>
<td>0.9655</td>
</tr>
<tr>
<td>CVXR (nodcp)</td>
<td>0.9002</td>
<td>0.9655</td>
<td>0.9962</td>
<td>0.9021</td>
<td>0.9668</td>
</tr>
<tr>
<td><strong>Running Time (in minutes)</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RMosek</td>
<td>2.1649</td>
<td>1.3248</td>
<td>1.0597</td>
<td>3.4226</td>
<td>2.0508</td>
</tr>
<tr>
<td>CVXR</td>
<td>7.5004</td>
<td>4.3567</td>
<td>3.3543</td>
<td>8.7074</td>
<td>5.1013</td>
</tr>
<tr>
<td>CVXR (nodcp)</td>
<td>3.5653</td>
<td>2.0942</td>
<td>1.8425</td>
<td>5.9993</td>
<td>2.9410</td>
</tr>
</tbody>
</table>
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 CRAN site and documentation.

Acknowledgements

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


Anqi Fu, Balasubramanian Narasimhan, Stephen Boyd


Wright SJ (1997). Primal-dual interior-point methods. SIAM.


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 $S^n$ refers to the set of symmetric matrices, with $S^n_+$ and $S^n_-$ referring to the set of positive semidefinite and negative semidefinite matrices, respectively.
### Table 3: Scalar Functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Meaning</th>
<th>Domain</th>
<th>Sign</th>
<th>Curvature</th>
<th>Monotonicity</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>geo_mean(x)</code></td>
<td>$x_1^{1/n} \ldots x_n^{1/n}$</td>
<td>$x \in \mathbb{R}_+^n$</td>
<td>+</td>
<td>concave</td>
<td>↗</td>
</tr>
<tr>
<td><code>geo_mean(x, p)</code></td>
<td>$(x_1^{p_1} \ldots x_n^{p_n})^{1/p}$</td>
<td>$x \in \mathbb{R}_+^n, p \neq 0$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>harmonic_mean(x)</code></td>
<td>$\frac{1}{x_1} + \ldots + \frac{1}{x_n}$</td>
<td>$x \in \mathbb{R}^n_+$</td>
<td>+</td>
<td>concave</td>
<td>↗</td>
</tr>
<tr>
<td><code>lambda_max(X)</code></td>
<td>$\lambda_{\text{max}}(X)$</td>
<td>$X \in \mathbb{S}^n$</td>
<td>±</td>
<td>convex</td>
<td>none</td>
</tr>
<tr>
<td><code>lambda_min(X)</code></td>
<td>$\lambda_{\text{min}}(X)$</td>
<td>$X \in \mathbb{S}^n$</td>
<td>±</td>
<td>convex</td>
<td>none</td>
</tr>
<tr>
<td><code>lambda_sum_largest(X, k)</code></td>
<td>sum of $k$ largest eigenvalues of $X$</td>
<td>$X \in \mathbb{S}^n$</td>
<td>±</td>
<td>convex</td>
<td>none</td>
</tr>
<tr>
<td><code>lambda_sum_smallest(X, k)</code></td>
<td>sum of $k$ smallest eigenvalues of $X$</td>
<td>$X \in \mathbb{S}^n$</td>
<td>±</td>
<td>convex</td>
<td>none</td>
</tr>
<tr>
<td><code>log_det(X)</code></td>
<td>$\log(\det(X))$</td>
<td>$X \in \mathbb{S}^n_+$</td>
<td>±</td>
<td>concave</td>
<td>none</td>
</tr>
<tr>
<td><code>log_sum_exp(X)</code></td>
<td>$\log(\sum_{ij} e^{X_{ij}})$</td>
<td>$X \in \mathbb{R}^{m \times n}$</td>
<td>±</td>
<td>convex</td>
<td>↗</td>
</tr>
<tr>
<td><code>matrix_frac(x, P)</code></td>
<td>$x^T P^{-1} x$</td>
<td>$x \in \mathbb{R}^n$, $P \in \mathbb{S}^n_+^{++}$</td>
<td>+</td>
<td>convex</td>
<td>none</td>
</tr>
<tr>
<td><code>max_entries(X)</code></td>
<td>$\max_{ij} { X_{ij} }$</td>
<td>$X \in \mathbb{R}^{m \times n}$</td>
<td>same as $X$</td>
<td>convex</td>
<td>↗</td>
</tr>
<tr>
<td><code>min_entries(X)</code></td>
<td>$\min_{ij} { X_{ij} }$</td>
<td>$X \in \mathbb{R}^{m \times n}$</td>
<td>same as $X$</td>
<td>concave</td>
<td>↗</td>
</tr>
<tr>
<td><code>mixed_norm(X, p, q)</code></td>
<td>$\left( \sum_{ij} (\sum_k</td>
<td>x_{k,i}</td>
<td>^p)^{q/p} \right)^{1/q}$</td>
<td>$X \in \mathbb{R}^{m \times n}$</td>
<td>+</td>
</tr>
<tr>
<td><code>cvxr_norm(x)</code></td>
<td>$\sqrt{\sum_i x_i^2}$</td>
<td>$X \in \mathbb{R}^n$</td>
<td>+</td>
<td>convex</td>
<td>↗ for $x_i \geq 0$, ↘ for $x_i \leq 0$</td>
</tr>
<tr>
<td><code>cvxr_norm(x, 2)</code></td>
<td>$\lambda_{\text{max}}(X^T X)^{1/2}$</td>
<td>$X \in \mathbb{R}^{m \times n}$</td>
<td>+</td>
<td>convex</td>
<td>↗ for $X_{ij} \geq 0$, ↘ for $X_{ij} \leq 0$</td>
</tr>
<tr>
<td><code>cvxr_norm(X, &quot;fro&quot;)</code></td>
<td>$\sqrt{\sum_{ij} X_{ij}^2}$</td>
<td>$X \in \mathbb{R}^{m \times n}$</td>
<td>+</td>
<td>convex</td>
<td>↗ for $X_{ij} \geq 0$, ↘ for $X_{ij} \leq 0$</td>
</tr>
<tr>
<td><code>cvxr_norm(X, 1)</code></td>
<td>$\sum_{ij}</td>
<td>X_{ij}</td>
<td>$</td>
<td>$X \in \mathbb{R}^{m \times n}$</td>
<td>+</td>
</tr>
<tr>
<td><code>cvxr_norm(X, &quot;inf&quot;)</code></td>
<td>$\max_{ij} {</td>
<td>X_{ij}</td>
<td>}$</td>
<td>$X \in \mathbb{R}^{m \times n}$</td>
<td>+</td>
</tr>
<tr>
<td><code>cvxr_norm(X, &quot;nuc&quot;)</code></td>
<td>$\text{tr} \left((X^T X)^{1/2}\right)$</td>
<td>$X \in \mathbb{R}^{m \times n}$</td>
<td>+</td>
<td>convex</td>
<td>none</td>
</tr>
<tr>
<td><code>cvxr_norm(X)</code></td>
<td>$\sqrt{\lambda_{\text{max}}(X^T X)}$</td>
<td>$X \in \mathbb{R}^{m \times n}$</td>
<td>+</td>
<td>convex</td>
<td>none</td>
</tr>
</tbody>
</table>
Table 4: More Scalar Functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Meaning</th>
<th>Domain</th>
<th>Sign</th>
<th>Curvature</th>
<th>Monotonicity</th>
</tr>
</thead>
<tbody>
<tr>
<td>$p_{\text{norm}}(X, p)$</td>
<td>$|X|<em>p = \left(\sum</em>{ij}</td>
<td>X_{ij}</td>
<td>^p\right)^{1/p}$</td>
<td>$X \in \mathbb{R}^{m \times n}$</td>
<td>+</td>
</tr>
<tr>
<td>$p_{\text{norm}}(X, p)$</td>
<td>$|X|<em>p = \left(\sum</em>{ij} X_{ij}^p\right)^{1/p}$</td>
<td>$X \in \mathbb{R}_{+}^{m \times n}$</td>
<td>+</td>
<td>concave</td>
<td>$\uparrow$</td>
</tr>
<tr>
<td>quad_form($x, P$)</td>
<td>$x^T P x$</td>
<td>$x \in \mathbb{R}^n$</td>
<td>+</td>
<td>convex</td>
<td>$\uparrow$ for $x_i \geq 0$, $\downarrow$ for $x_i \leq 0$</td>
</tr>
<tr>
<td>quad_form($x, P$)</td>
<td>$x^T P x$</td>
<td>$x \in \mathbb{R}^n$</td>
<td>-</td>
<td>concave</td>
<td>$\uparrow$ for $x_i \geq 0$, $\downarrow$ for $x_i \leq 0$</td>
</tr>
<tr>
<td>quad_form($c, X$)</td>
<td>$c^T X c$</td>
<td>$X \in \mathbb{R}^{m \times n}$</td>
<td>affine &amp; depends on $c$, $X$</td>
<td>convex &amp; depends on $c$</td>
<td></td>
</tr>
<tr>
<td>quad_over_lin($X, y$)</td>
<td>$(\sum_{ij} X_{ij}^2) / y$</td>
<td>$x \in \mathbb{R}^n$, $y &gt; 0$</td>
<td>+</td>
<td>convex</td>
<td>$\uparrow$ for $X_{ij} \geq 0$, $\downarrow$ for $X_{ij} \leq 0$, $\downarrow$ in $y$</td>
</tr>
<tr>
<td>sum_entries($X$)</td>
<td>$\sum_{ij} X_{ij}$</td>
<td>$X \in \mathbb{R}^{m \times n}$</td>
<td>same as $X$</td>
<td>affine</td>
<td>$\uparrow$</td>
</tr>
<tr>
<td>sum_largest($X, k$)</td>
<td>sum of $k$ largest $X_{ij}$</td>
<td>$X \in \mathbb{R}^{m \times n}$</td>
<td>same as $X$</td>
<td>convex</td>
<td>$\uparrow$</td>
</tr>
<tr>
<td>sum_smallest($X, k$)</td>
<td>sum of $k$ smallest $X_{ij}$</td>
<td>$X \in \mathbb{R}^{m \times n}$</td>
<td>same as $X$</td>
<td>concave</td>
<td>$\uparrow$</td>
</tr>
<tr>
<td>sum_squares($X$)</td>
<td>$\sum_{ij} X_{ij}^2$</td>
<td>$X \in \mathbb{R}^{m \times n}$</td>
<td>+</td>
<td>convex</td>
<td>$\uparrow$ for $X_{ij} \geq 0$, $\downarrow$ for $X_{ij} \leq 0$</td>
</tr>
<tr>
<td>matrix_trace($X$)</td>
<td>$\text{tr}(X)$</td>
<td>$X \in \mathbb{R}^{n \times n}$</td>
<td>same as $X$</td>
<td>affine</td>
<td>$\uparrow$</td>
</tr>
<tr>
<td>tv($x$)</td>
<td>$\sum_{i}</td>
<td>x_{i+1} - x_i</td>
<td>$</td>
<td>$x \in \mathbb{R}^n$</td>
<td>+</td>
</tr>
<tr>
<td>tv($X$)</td>
<td>$\sum_{ij} \left|X_{i+1,j} - X_{ij}\right|_2$</td>
<td>$X \in \mathbb{R}^{m \times n}$</td>
<td>+</td>
<td>convex</td>
<td>none</td>
</tr>
<tr>
<td>tv($X_1, \ldots, X_k$)</td>
<td>$\sum_{ij} \left|X_{i+1,j}^{(1)} - X_{ij}^{(1)}\right|_2$</td>
<td>$X^{(i)} \in \mathbb{R}^{m \times n}$</td>
<td>+</td>
<td>convex</td>
<td>none</td>
</tr>
</tbody>
</table>
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[i, 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 3 and 4, 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.

\[
\text{sum}(expr1, expr2, expr3)
\]

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) 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:

X <- Variable(5, 4)
<table>
<thead>
<tr>
<th>Function</th>
<th>Meaning</th>
<th>Domain</th>
<th>Sign</th>
<th>Curvature</th>
<th>Monotonicity</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\text{abs}(x))</td>
<td>(</td>
<td>x</td>
<td>)</td>
<td>(x \in \mathbb{R})</td>
<td>+</td>
</tr>
<tr>
<td>(\text{entr}(x))</td>
<td>(-x \log(x))</td>
<td>(x &gt; 0)</td>
<td>±</td>
<td>concave</td>
<td>none</td>
</tr>
<tr>
<td>(\text{exp}(x))</td>
<td>(e^x)</td>
<td>(x \in \mathbb{R})</td>
<td>+</td>
<td>convex</td>
<td>(\uparrow)</td>
</tr>
<tr>
<td>(\text{huber}(x, M = 1))</td>
<td>(\begin{cases} x^2 &amp;</td>
<td>x</td>
<td>\leq M \ 2M</td>
<td>x</td>
<td>- M^2 &amp;</td>
</tr>
<tr>
<td>(\text{inv_pos}(x))</td>
<td>(1/x)</td>
<td>(x &gt; 0)</td>
<td>+</td>
<td>concave</td>
<td>(\uparrow)</td>
</tr>
<tr>
<td>(\text{kl_div}(x, y))</td>
<td>(x \log(x/y) - x + y)</td>
<td>(x &gt; 0, y &gt; 0)</td>
<td>+</td>
<td>convex</td>
<td>none</td>
</tr>
<tr>
<td>(\text{log}(x))</td>
<td>(\log(x))</td>
<td>(x &gt; 0)</td>
<td>±</td>
<td>concave</td>
<td>(\uparrow)</td>
</tr>
<tr>
<td>(\text{log1p}(x))</td>
<td>(\log(x + 1))</td>
<td>(x &gt; -1)</td>
<td>same as (x)</td>
<td>concave</td>
<td>(\uparrow)</td>
</tr>
<tr>
<td>(\text{logistic}(x))</td>
<td>(\log(1 + e^x))</td>
<td>(x \in \mathbb{R})</td>
<td>+</td>
<td>convex</td>
<td>(\uparrow)</td>
</tr>
<tr>
<td>(\text{max elemwise}(x_1, \ldots, x_k))</td>
<td>(\max{x_1, \ldots, x_k})</td>
<td>(x_i \in \mathbb{R})</td>
<td>+</td>
<td>convex</td>
<td>(\uparrow)</td>
</tr>
<tr>
<td>(\text{min elemwise}(x_1, \ldots, x_k))</td>
<td>(\min{x_1, \ldots, x_k})</td>
<td>(x_i \in \mathbb{R})</td>
<td>-</td>
<td>concave</td>
<td>(\uparrow)</td>
</tr>
<tr>
<td>(\text{mul elemwise}(c, x))</td>
<td>(c \times x)</td>
<td>(c \in \mathbb{R})</td>
<td>sign((cx))</td>
<td>affine</td>
<td>depends on (c)</td>
</tr>
<tr>
<td>(\text{neg}(x))</td>
<td>(\max{-x, 0})</td>
<td>(x \in \mathbb{R})</td>
<td>+</td>
<td>convex</td>
<td>(\checkmark)</td>
</tr>
<tr>
<td>(\text{pos}(x))</td>
<td>(\max{x, 0})</td>
<td>(x \in \mathbb{R})</td>
<td>+</td>
<td>convex</td>
<td>(\uparrow)</td>
</tr>
<tr>
<td>(\text{power}(x, 0))</td>
<td>1</td>
<td>(x \in \mathbb{R})</td>
<td>+</td>
<td>constant</td>
<td>(\uparrow)</td>
</tr>
<tr>
<td>(\text{power}(x, 1))</td>
<td>(x)</td>
<td>(x \in \mathbb{R})</td>
<td>+</td>
<td>affine</td>
<td>(\uparrow)</td>
</tr>
<tr>
<td>(\text{power}(x, p))</td>
<td>(x^p)</td>
<td>(x \in \mathbb{R})</td>
<td>+</td>
<td>convex</td>
<td>(\uparrow) for (x \geq 0), (\checkmark) for (x \leq 0)</td>
</tr>
<tr>
<td>(\text{power}(x, p))</td>
<td>(x^p)</td>
<td>(x &gt; 0)</td>
<td>+</td>
<td>convex</td>
<td>(\checkmark)</td>
</tr>
<tr>
<td>(\text{power}(x, p))</td>
<td>(x^p)</td>
<td>(0 &lt; p &lt; 1)</td>
<td>+</td>
<td>concave</td>
<td>(\uparrow)</td>
</tr>
<tr>
<td>(\text{power}(x, p))</td>
<td>(x^p)</td>
<td>(p &gt; 1, p \neq 2, 4, 8, \ldots)</td>
<td>+</td>
<td>convex</td>
<td>(\uparrow) for (x \geq 0), (\checkmark) for (x \leq 0)</td>
</tr>
<tr>
<td>(\text{scalene}(x, \alpha, \beta))</td>
<td>(\alpha \text{pos}(x) + \beta \text{neg}(x))</td>
<td>(x \in \mathbb{R})</td>
<td>+</td>
<td>convex</td>
<td>(\uparrow) for (x \geq 0), (\checkmark) for (x \leq 0)</td>
</tr>
<tr>
<td>(\text{sqrt}(x))</td>
<td>(\sqrt{x})</td>
<td>(x \geq 0)</td>
<td>+</td>
<td>concave</td>
<td>(\uparrow)</td>
</tr>
<tr>
<td>(\text{square}(x))</td>
<td>(x^2)</td>
<td>(x \in \mathbb{R})</td>
<td>+</td>
<td>convex</td>
<td>(\uparrow) for (x \geq 0), (\checkmark) for (x \leq 0)</td>
</tr>
<tr>
<td>Function</td>
<td>Meaning</td>
<td>Domain</td>
<td>Sign</td>
<td>Curvature</td>
<td>Monotonicity</td>
</tr>
<tr>
<td>------------------------------</td>
<td>-----------------------------------</td>
<td>---------------------------------</td>
<td>--------------------------------</td>
<td>-----------</td>
<td>--------------</td>
</tr>
<tr>
<td><code>bmat([[X_{11},..., X_{1q}],...,[X_{p1},..., X_{pq}]]))</code></td>
<td>$\begin{bmatrix} X^{(1,1)} &amp; \cdots &amp; X^{(1,q)} \ \vdots &amp; \ddots &amp; \vdots \ X^{(p,1)} &amp; \cdots &amp; X^{(p,q)} \end{bmatrix}$</td>
<td>$X_{i,j} \in \mathbb{R}^{m_i \times n_j}$</td>
<td>$\text{sign} \left( \sum_{i,j} X_{i,j}^{(i,j)} \right)$</td>
<td>affine</td>
<td>$\nearrow$</td>
</tr>
<tr>
<td><code>conv(c, x)</code></td>
<td>$c \ast x$</td>
<td>$x \in \mathbb{R}^n$</td>
<td>$\text{sign} \left( c_1 x_1 \right)$</td>
<td>affine</td>
<td>depends on $c$</td>
</tr>
<tr>
<td><code>cumsum_axis(X, axis = 1)</code></td>
<td>cumulative sum along given axis.</td>
<td>$X \in \mathbb{R}^{n \times n}$</td>
<td>same as $X$</td>
<td>affine</td>
<td>$\nearrow$</td>
</tr>
<tr>
<td><code>diag(x)</code></td>
<td>$\begin{bmatrix} x_1 \ \vdots \ x_n \end{bmatrix}$</td>
<td>$x \in \mathbb{R}^n$</td>
<td>same as $x$</td>
<td>affine</td>
<td>$\nearrow$</td>
</tr>
<tr>
<td><code>diag(X)</code></td>
<td>$\begin{bmatrix} X_{11} \ \vdots \ X_{nn} \end{bmatrix}$</td>
<td>$X \in \mathbb{R}^{n \times n}$</td>
<td>same as $X$</td>
<td>affine</td>
<td>$\nearrow$</td>
</tr>
<tr>
<td><code>diff(X, k = 1, axis = 1)</code></td>
<td>$k$th order differences (argument $k$ is actually named <code>differences</code> and <code>lag</code> can also be used) along given axis</td>
<td>$X \in \mathbb{R}^{m \times n}$</td>
<td>same as $X$</td>
<td>affine</td>
<td>$\nearrow$</td>
</tr>
<tr>
<td><code>hstack(X_1, ..., X_k)</code></td>
<td>$\begin{bmatrix} X^{(1)} &amp; \cdots &amp; X^{(k)} \end{bmatrix}$</td>
<td>$X^{(i)} \in \mathbb{R}^{m_i \times n_i}$</td>
<td>$\text{sign} \left( \sum_i X^{(i)} \right)$</td>
<td>affine</td>
<td>$\nearrow$</td>
</tr>
<tr>
<td><code>kronecker(C, X)</code></td>
<td>$\begin{bmatrix} C_{11} X &amp; \cdots &amp; C_{1q} X \ \vdots &amp; \ddots &amp; \vdots \ C_{p1} X &amp; \cdots &amp; C_{pq} X \end{bmatrix}$</td>
<td>$X \in \mathbb{R}^{m \times n}$</td>
<td>$\text{sign} \left( C_{11} X_{11} \right)$</td>
<td>affine</td>
<td>depends on $C$</td>
</tr>
<tr>
<td><code>reshape_expr(X, n', m')</code></td>
<td>$X' \in \mathbb{R}^{m' \times n'}$</td>
<td>$X \in \mathbb{R}^{m \times n}$</td>
<td>same as $X$</td>
<td>affine</td>
<td>$\nearrow$</td>
</tr>
<tr>
<td><code>vec(X)</code></td>
<td>$x' \in \mathbb{R}^{mn}$</td>
<td>$X \in \mathbb{R}^{m \times n}$</td>
<td>same as $X$</td>
<td>affine</td>
<td>$\nearrow$</td>
</tr>
<tr>
<td><code>vstack(X_1, ..., X_k)</code></td>
<td>$\begin{bmatrix} X^{(1)} \ \vdots \ X^{(k)} \end{bmatrix}$</td>
<td>$X^{(i)} \in \mathbb{R}^{m_i \times n_i}$</td>
<td>$\text{sign} \left( \sum_i X^{(i)} \right)$</td>
<td>affine</td>
<td>$\nearrow$</td>
</tr>
</tbody>
</table>
row_sums <- sum_entries(X, axis = 1) # Has size (5, 1)
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 a input matrix as one long vector, basically the same as `base::apply` with `MARGIN = c(1, 2)`. The exception is `cumsum_axis` (see Table 5), 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 5. 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, such as `max_elemwise` and `mul_elemwise`, 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])`. This means 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 6, 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(X, 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
Department of Electrical Engineering
David Packard Building
350 Serra Mall
Stanford, CA 94305
E-mail: anqif@stanford.edu
URL: https://web.stanford.edu/~anqif/
Balasubramanian Narasimhan
Department of Biomedical Data Sciences, and
Department of Statistics
Stanford University
390 Serra Mall
Stanford, CA 94305
E-mail: naras@stanford.edu
URL: https://statistics.stanford.edu/~naras/

Stephen Boyd
Department of Electrical Engineering
David Packard Building
350 Serra Mall
Stanford, CA 94305
E-mail: boyd@stanford.edu
URL: https://web.stanford.edu/~boyd/