

# CVXPY 1.9: Recent Advances in Optimization Modeling Software

WILLIAM ZHANG, Polytechnique Montreal and Gridmatic, Canada

PARTH NOBEL, Optimal Intellect, USA

ARYAMAN JEENDGAR, Technical University of Munich and International Computer Science Institute, Germany

RILEY MURRAY, Sandia National Laboratories, USA

PHILIPP SCHIELE, Citadel, USA

STEVEN DIAMOND, Optimal Intellect, USA

CVXPY is a Python-embedded domain-specific language for convex optimization that lets users express problems in mathematical notation while the system verifies convexity and reduces valid programs to solver-ready form. This paper reports on the major advances from versions 1.1 through 1.9. These include a unified conic quadratic program (CQP) standard form for canonicalization; a stacked-slices backend that accelerates parameterized programs; first-class support for N-dimensional expressions; explicit sparsity for variables; support for multiple variable attributes; cones/atoms relevant to quantum information theory; and the introduction of disciplined nonlinear programming (DNLP). We outline the design, algorithms, and modeling consequences of these features.

CCS Concepts: • **Mathematics of computing** → **Solvers**; *Mathematical software performance*; **Continuous optimization**; **Integer programming**.

Additional Key Words and Phrases: convex optimization, domain-specific languages, conic programming, Python

## 1 Introduction

CVXPY began in 2013 as a Pythonic modular implementation of the modeling principles of CVX [22] in MATLAB. A fundamental design decision of CVXPY is the separation between modeling, canonicalization, and solver interfaces [14]. Modeling in CVXPY is based on a grammar called disciplined convex programming (DCP), which allows for systematic verification of convexity. CVXPY version 1.0 introduced a general reduction-based rewriting system for canonicalization [4], which enabled grammars such as disciplined geometric programming (DGP) [3] and disciplined quasiconvex programming (DQCP) [2], each being reduced to DCP-compatible forms. CVXPY version 1.1 added disciplined parametrized programming (DPP) [1], which allows users to efficiently update canonicalized programs under a restricted grammar and enables differentiable programs and code generation (CVXPYlayers [1], CVXPYgen [27, 28]). Other grammars which focus on different problem classes have been implemented as separate standalone packages. Examples include DCCP, DMCP, DCSP, and DSP [6, 29–31]. CVXPY 1.9 added support for disciplined nonlinear programming (DNLP), a new syntax for specifying nonlinear programs [12].

The rest of the paper is structured as follows: §2 presents relevant standard forms while §3 presents canonicalization, the process of converting between these standard forms. Finally, the paper concludes with a wide range of new canonicalization and modeling features in §4 and §5.

---

Authors' Contact Information: William Zhang, Polytechnique Montreal and Gridmatic, Montreal, Canada, william-2.zhang@polymtl.ca; Parth Nobel, Optimal Intellect, San Francisco, California, USA, ptnobel@alumni.stanford.edu; Aryaman Jeendgar, Technical University of Munich and International Computer Science Institute, Heilbronn, Baden-Württemberg, Germany, aryaman.jeendgar@tum.de; Riley Murray, Sandia National Laboratories, Livermore, California, USA, rjmurr@sandia.gov; Philipp Schiele, Citadel, New York, New York, USA, phschiele@gmail.com; Steven Diamond, Optimal Intellect, Mountain View, California, USA, diamond@cs.stanford.edu.

## 2 Standard Forms

CVXPY is a compiler between functional convex programming, a user-friendly representation of convex optimization problems, and conic quadratic programming (and its special cases), a solver-friendly representation of convex optimization problems. Recognizing that users frequently want to solve similar problems, CVXPY supports parameterization, enabling reuse of compilation.

### 2.1 Functional Convex Programming

When a user specifies a problem in CVXPY, the first step is to verify that the problem is convex. Users specify problems—by providing expression trees for each function—as the standard “functional” form for convex optimization

$$\begin{aligned} & \text{minimize} && f_0(x; \theta) \\ & \text{subject to} && f_i(x; \theta) \leq 0, \quad i = 1, \dots, m_1, \\ & && g_i(x; \theta) = 0, \quad i = 1, \dots, m_2, \end{aligned} \tag{1}$$

with variable  $x \in \mathbf{R}^n$  and parameter vector  $\theta \in \mathbf{R}^p$  [9]. When  $\theta$  is fixed, we refer to (1) as a problem instance, otherwise we refer to (1) as a problem family.

Theorem 2.1 provides a sufficient criterion for verifying convexity of the functional form. The process of applying this convexity theorem on every node of the expression tree is known as disciplined convex programming (DCP). It is disciplined in the sense that it restricts the choice of representation of mathematical expressions. For example  $\sqrt{x^2}$  is not DCP, whereas  $|x|$  is. In addition to per-argument monotonicities, CVXPY uses sign information to extend the set of valid compositions. Without signed DCP,  $f(g(x)) = (|x|)^2$  is invalid since  $f(t) = t^2$  is not monotone on  $\mathbf{R}$ . With signed DCP, it is valid because  $f$  is non-decreasing on  $\mathbf{R}_+$  and the image of  $g$  is  $\mathbf{R}_+$ .

**THEOREM 2.1 (CONVEX COMPOSITION RULE).**  *$h(f_1(x), f_2(x), \dots, f_k(x))$  is convex when  $h$  is convex and for each  $i \in \{1, \dots, k\}$  the following holds:*

- *$h$  is non-decreasing in argument  $i$  and  $f_i$  is convex, or*
- *$h$  is non-increasing in argument  $i$  and  $f_i$  is concave, or*
- *$f_i$  is affine.*

*The same results hold for concave functions by swapping any mention of convex with concave (and vice versa). Affine expressions require  $h$  and all of its arguments to be affine [22].*

### 2.2 Conic Quadratic Programming

Quadratic programs generalize linear programs by allowing quadratic objective functions. Cone programs generalize linear programs by abstracting constraints of the form “ $b - Ax \succeq 0$ ” to “ $b - Ax \in \mathcal{K}$ ”, where  $\mathcal{K}$  is a convex cone.<sup>1</sup> We say *conic quadratic program* in reference to a convex optimization problem with conic constraints and a linear or quadratic objective.

*Remark.* Conic quadratic programming is not a major generalization of conic programming. If we allow the use of the second-order cone, then a conic quadratic program can have its objective linearized at the expense of one additional variable and one additional constraint with dimension 1 plus the rank of the quadratic objective matrix. Nevertheless, retaining the quadratic objective can be useful from an algorithmic standpoint.

<sup>1</sup>That is,  $\mathcal{K}$  is a convex set such that  $\theta x \in \mathcal{K}$  for all  $x \in \mathcal{K}$  and all  $\theta \geq 0$ .

Specialized solvers exist for conic quadratic programs [13, 21, 25], cone programs [7, 15], and quadratic programs [32]. CVXPY interfaces with any such solver by first canonicalizing to a conic quadratic program

$$\begin{aligned} & \text{minimize} && \frac{1}{2}x^T Px + q^T x \\ & \text{subject to} && b - Ax \in \mathcal{K}, \end{aligned} \tag{2}$$

where  $x \in \mathbf{R}^n$  is the variable,  $\mathcal{K} \subseteq \mathbf{R}^m$  is a closed convex cone, and the *problem data* are  $P \in \mathbf{S}_+^n$ ,  $q \in \mathbf{R}^n$ ,  $A \in \mathbf{R}^{m \times n}$ , and  $b \in \mathbf{R}^m$ . CVXPY supports problems where the cone is a product of the zero cone ( $\{0\}$ ), the nonnegative cone ( $\mathbf{R}_+$ ), the second-order cone ( $\mathcal{K}_{\text{soc},n} = \{(t, x) \mid \|x\|_2 \leq t, t \in \mathbf{R}, x \in \mathbf{R}^{n-1}\}$ ), the power cone ( $\mathcal{K}_{\text{pow},\alpha} = \{(x, y, z) \mid x \geq 0, y \geq 0, x^\alpha y^{1-\alpha} \geq |z|\}$ ), the exponential cone ( $\mathcal{K}_{\text{exp}} = \text{cl}\{(x, y, z) \mid y > 0, y \exp(x/y) \leq z\}$ ), and the cone of positive semidefinite matrices ( $\mathbf{S}_+$ ). Previously, CVXPY used two standard forms, cone programs, for which  $P = 0$ , and quadratic programs for which  $\mathcal{K}$  is a product of only the zero cone and the nonnegative cone. Further elaboration on unifying the two standard forms is in §4.1.

### 2.3 Parameterized Conic Quadratic Programming

In parameterized conic quadratic programming,  $P, q, A, b$  in (2) are functions of a parameter  $\theta$ . DPP is a grammar that guarantees parametrized convex programs can be reduced to affine-solver-affine (ASA) form. ASA form is a representation of an optimization problem such that  $P, q, A, b$  are affine functions of  $\theta$  and the solution of the original problem is an affine function of the solution to the canonicalized problem.

DPP consists of the same composition rule as DCP but with additional restrictions on expressions involving parameters. A parameter is a symbolic constant with known properties (*e.g.*, sign and sparsity) and unknown numeric value. The following canonicalization theorem shows that DPP-compliant programs can be reduced to ASA form.

**THEOREM 2.2 (CANONICALIZATION MAP).** *The canonicalization map for a disciplined parametrized program can be represented by sparse tensors  $Q \in \mathbf{R}^{n \times (n+1) \times (p+1)}$  and  $R \in \mathbf{R}^{m \times (n+1) \times (p+1)}$ , where  $m$  is the dimension of the constraints. Letting  $\tilde{\theta} \in \mathbf{R}^{p+1}$  denote the concatenation of  $\theta$  and the scalar 1, the quadratic cone problem data can be obtained as*

$$[P \quad q] = \sum_{i=1}^{p+1} Q[:, :, i] \cdot \tilde{\theta}_i, \quad \text{and} \quad [A \quad b] = \sum_{i=1}^{p+1} R[:, :, i] \cdot \tilde{\theta}_i. \tag{3}$$

The proof can be adapted from [1].

## 3 Canonicalization

Canonicalization converts a verified DCP/DPP model into a cone program via a sequence of reductions [4, 22]. The front end verifies convexity by parsing the user-specified expression tree. Then, reductions—*e.g.*, negating the objective of maximization problems and turning variable attributes into explicit constraints—are applied. The last symbolic reduction is to expand nonlinear atoms to their epigraph form, resulting in an affine expression tree. Subsequently, a numerical reduction reduces the affine expression tree into the sparse tensors from Theorem 2.2. Finally, solver-specific transformations are applied to match the form required by the solvers' interfaces. At each reduction, the inverse transformations are also recorded, allowing the solution of the standard form problem to be mapped back to the user-specified problem.

As an illustrative example, adapted from [1], consider the following optimization problem

$$\begin{aligned} & \text{minimize} && \|Fx - g\|_2 + \mu \|x\|_2 \\ & \text{subject to} && x \geq 0 \end{aligned} \tag{4}$$

with variable  $x \in \mathbf{R}^n$  and parameters  $F \in \mathbf{R}^{m \times n}$ ,  $g \in \mathbf{R}^m$ , and  $\mu \geq 0$ . This problem formulation given by the following CVXPY code conforms to the DPP grammar:

```
x = cp.Variable(n)
F = cp.Parameter((m, n))
g = cp.Parameter(m)
mu = cp.Parameter(nonneg=True)
objective = cp.Minimize(cp.norm(F @ x - g) + mu * cp.norm(x))
prob = cp.Problem(objective, [x >= 0])
```

For brevity, all code snippets omit the definition of dimension parameters (e.g.,  $n$  and  $m$  in this example) and the imports of cvxpy and numpy.

In the following subsections we describe in greater detail the canonicalization procedure for problem (4). In the first phase, a front end takes a human-readable specification of a program and converts it to an intermediate representation; both compilers and rewriting systems often use abstract syntax trees as their intermediate representations [4, 5].

### 3.1 Expression Tree Representation

The first canonicalization step is to verify that the objective and constraint expressions of a CVXPY problem conform to the specified rule set. If a problem does not conform to the rule set, CVXPY raises an error message. This phase of the canonicalization is known as *parsing*. Throughout the parsing, an intermediate tree representation, also known as an *abstract syntax tree* (AST), is formed. In the AST, each parent node represents an atomic function, called an *atom*, whose children nodes are the arguments to the atom. The leaf nodes of the tree are variables, parameters, or constants. A DPP syntax tree for problem (4) is shown in Figure 1.

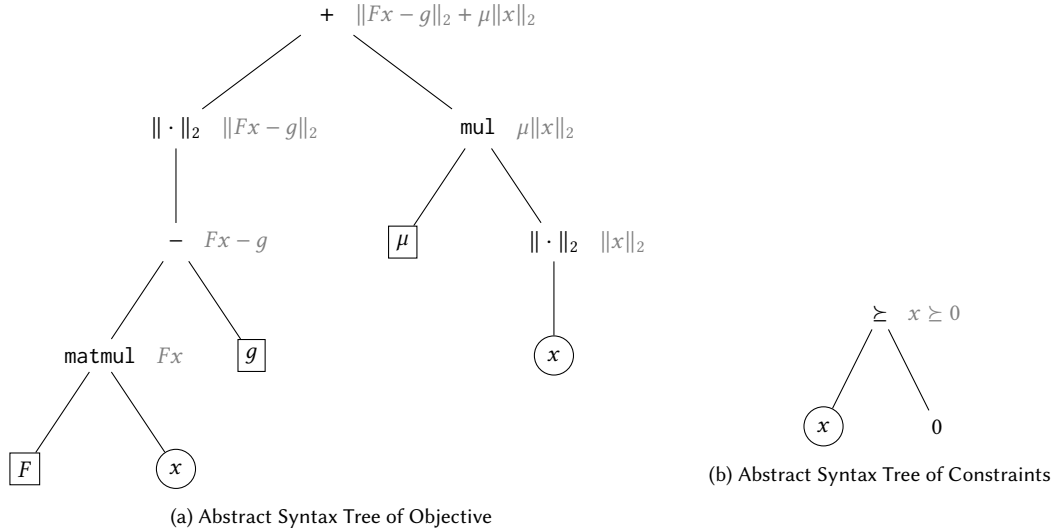


Fig. 1. AST Representation. Users write expressions in Python code that is immediately parsed into an AST, shown here. DPP verification is then run on the AST to ensure canonicalization will be possible on the problem and to verify convexity. Variables are labeled with a circle, and parameters with a square. Leaf nodes that refer to the same parameter or variable are deduplicated in practice (not shown here).

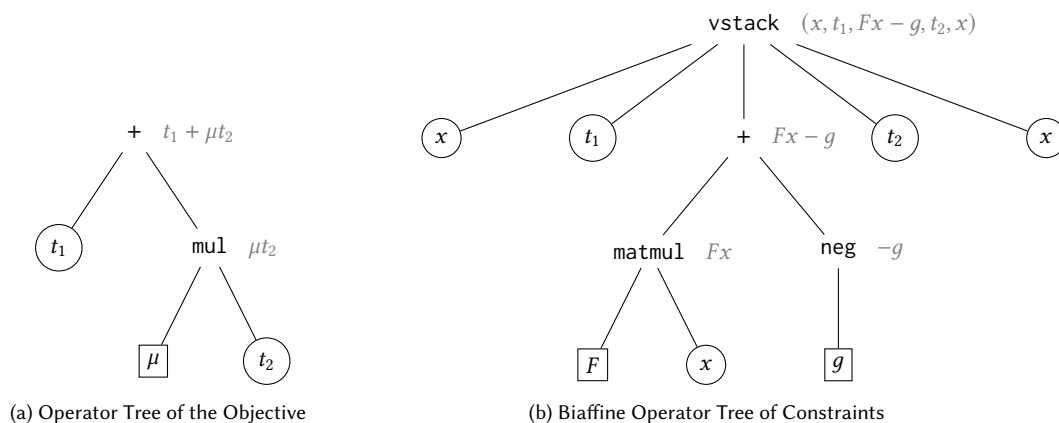


Fig. 2. Biaffine Operator Trees. After the application of reductions, the constraints and the gradient with respect to the variables of the objective are biaffine functions. As with the AST representation, duplicate leaves are deduplicated in practice, which is not shown here. For readability, we show the objective rather than its gradient.

### 3.2 Reductions

Following the parsing and DPP verification, reductions are applied to transform the problem into a cone program. In DPP, each atom has a *canonicalization* that replaces nonlinear functions with an epigraph variable and auxiliary constraints of the form  $f(x, t, \theta) \in \mathcal{K}$  where  $t$  is the epigraph variable,  $f$  is a biaffine function of the variables  $(x, t)$  and the parameters  $\theta$ , and  $\mathcal{K}$  is a convex cone. One illustrative canonicalization replaces the  $\|z_i\|_2$  expressions in the objective with a new epigraph variable  $t_i$  and imposes the epigraph constraint in its conic form

$$(t_i, z_i) \in \mathcal{K}_{\text{soc}}. \quad (5)$$

### 3.3 Biaffine Operator Tree Representation

Following the reductions, the problem has been reduced to an equivalent parameterized convex cone program:

$$\begin{aligned} &\text{minimize} && t_1 + \mu t_2 \\ &\text{subject to} && x \in \mathbf{R}_+^n \\ &&& (t_1, Fx - g) \in \mathcal{K}_{\text{soc}, m+1} \\ &&& (t_2, x) \in \mathcal{K}_{\text{soc}, n+1} \end{aligned} \quad (6)$$

with variables  $t_1, t_2, x$ . We represent the biaffine function in the constraints and the gradient of the objective (which is an affine function since the objective is either quadratic or linear) as trees of biaffine operators, see Figure 2.

### 3.4 Matrix Stuffing Representation

Finally, the biaffine operator trees are materialized as sparse tensors. This step is called *matrix stuffing* and is done by the *canonicalization backend*. Canonicalization backends apply a depth-first walk on the tree to generate the sparse tensors  $Q$  and  $R$  from Theorem 2.2. See Figure 3.

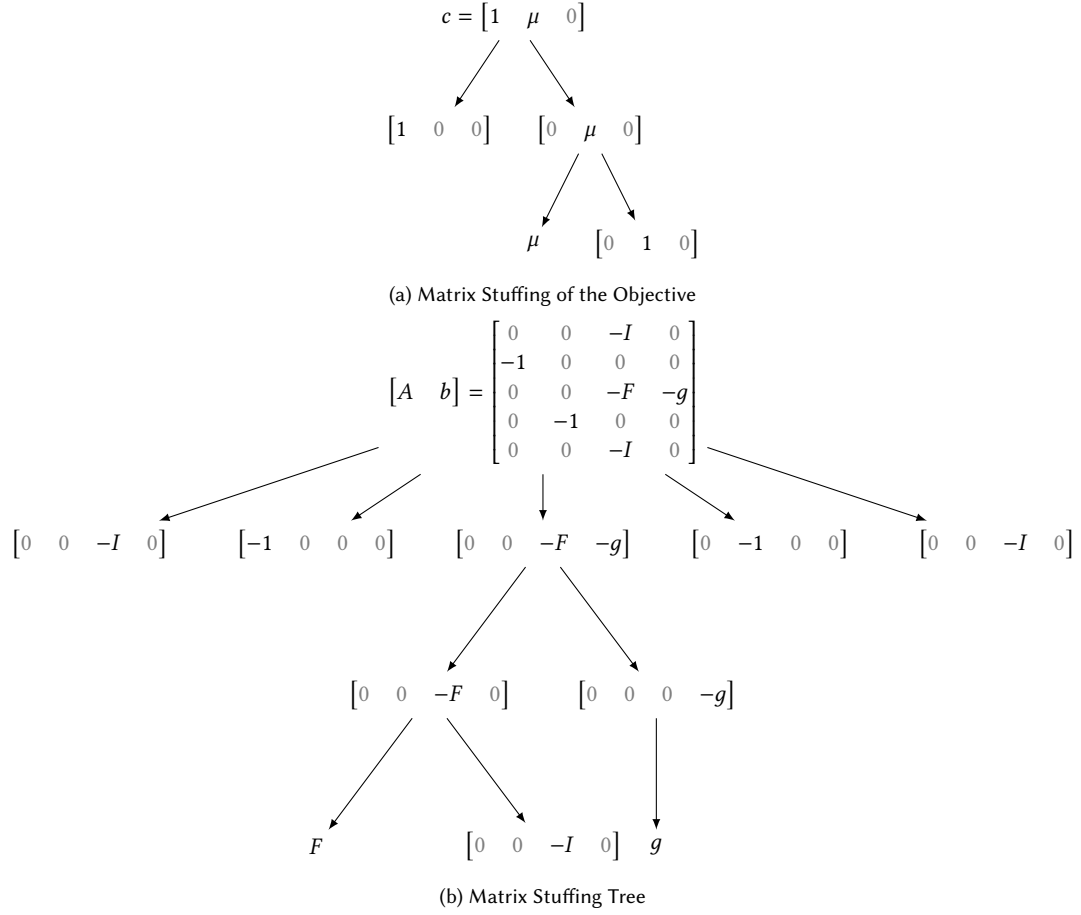


Fig. 3. Matrix Stuffing Representation. Matrix stuffing generates a 3D tensor. To depict these 3D tensors here, we leave the parameter values symbolic. Compare this figure with Figure 2, which shows the operations being applied at each node.

## 4 New Canonicalization Features

### 4.1 Quadratic Cone Program Standard Form

When CVXPY was originally designed, it created two canonicalization pathways: one for cone programs,

$$\begin{aligned} & \text{minimize} && c^T x \\ & \text{subject to} && Ax + s = b \\ & && s \in \mathcal{K}, \end{aligned}$$

and one for quadratic programs,

$$\begin{aligned} & \text{minimize} && \frac{1}{2} x^T P x + q^T x \\ & \text{subject to} && \ell \preceq Ax \preceq u. \end{aligned}$$

Recognizing the opportunity of recent solver advancements [21, 25], we combined our two standard forms into a single canonicalization pathway that generalizes both forms.

We now canonicalize to

$$\begin{aligned} & \text{minimize} && \frac{1}{2}x^T Px + q^T x \\ & \text{subject to} && Ax + s = b \\ & && s \in \mathcal{K}. \end{aligned}$$

The reduction to a quadratic cone program is a modification of the reduction to a linear cone program. Instead of expanding all nonlinear atoms into an affine objective and conic constraints, quadratic-representable nonlinear atoms are expanded into a quadratic objective and conic constraints.

Most atoms with quadratic objectives in their expansion can be expressed purely as a quadratic term  $x^T Px$ , with no conic constraints. A few functions have more interesting expansions, such as the huber function

$$\text{huber}(x, M) = \begin{cases} x^2 & |x| \leq M \\ 2M|x| - M^2 & |x| > M. \end{cases}$$

The huber function can be rewritten as

$$\begin{aligned} & \text{minimize} && t_1^2 + 2M|t_2| \\ & \text{subject to} && t_1 + t_2 = x, \end{aligned}$$

where  $t_1$  and  $t_2$  are auxiliary variables.

CVXPY users may still specify that a problem be canonicalized as a cone program with a linear objective rather than with a quadratic objective.

## 4.2 Disciplined Nonlinear Programming

In version 1.9, CVXPY allows users to specify nonlinear programs (NLP) of the form

$$\begin{aligned} & \text{minimize} && f(x) \\ & \text{subject to} && c(x) = 0 \\ & && \ell \leq x \leq u, \end{aligned} \tag{7}$$

where  $\ell \in \mathbf{R}^n$  and  $u \in \mathbf{R}^n$  are given variable bounds and  $f : \mathbf{R}^n \rightarrow \mathbf{R}$  and  $c : \mathbf{R}^n \rightarrow \mathbf{R}^m$  are twice differentiable functions that can be nonconvex. In addition, CVXPY supports combining nonsmooth convex and concave functions with smooth functions, subject to restrictions on how nonsmooth functions can be used. As in DCP, nonsmooth functions are relaxed in a lossless way to a form that can be handled by NLP solvers. CVXPY currently supports the open-source solvers IPOPT [35] and UNO [33], and the commercial solvers KNITRO [11] and COPT [20]. For more details on the DNLP syntax, grammar rules, and canonicalization, see [12].

## 4.3 Stacked-Slices Backend

In the hopes of replacing our deprecated C++-based backend CVXCORE, we developed a SciPy .sparse-based backend. To represent three-dimensional sparse tensors, the initial Python backend held per-parameter slices as lists of sparse matrices, which incurred Python-loop overhead for parameters of large size. The new *stacked-slices* architecture vertically concatenates slices and applies the linear maps via Kronecker identities. For a tensor  $Q \in \mathbf{R}^{a \times b \times (p+1)}$  whose slices correspond to the augmented parameter vector  $\tilde{\theta} = (\theta, 1) \in \mathbf{R}^{p+1}$ , let  $Q_i = Q[:, :, i] \in \mathbf{R}^{a \times b}$  and let  $\mathcal{S}(Q) \in \mathbf{R}^{(p+1)a \times b}$  denote the vertical stack of the slices  $Q_i$ . If  $L \in \mathbf{R}^{c \times a}$  is a linear map applied to each slice and  $I_{p+1} \in \mathbf{R}^{(p+1) \times (p+1)}$  is the identity matrix over the parameter and constant slices, then

$$\mathcal{S}(LQ) = (I_{p+1} \otimes L) \mathcal{S}(Q).$$

This change removes most Python loops over lists of slices and speeds up canonicalization significantly in many cases. A continuous asv benchmark suite tracks performance on standard workloads in finance, SDP, and differentiable settings [16].

In version 1.8, a variant of the stacked-slices backend was introduced that uses a custom N-dimensional sparse COO representation in place of SciPy.sparse. The COO representation is more memory efficient, particularly for problems with large numbers of parameters. CVXPY defaults to the COO backend for DPP problems with more than 1000 parameters.

## 5 New Modeling Features

This section covers recent improvements to the modeling abilities of CVXPY: support for N-dimensional expressions, a sparsity attribute, and multiple attributes. This section also reviews CVXPY’s features for modeling in quantum information theory; the implementation of these features also enabled approximate solution of mixed-integer exponential-cone programs (and mixed-integer geometric programs) with mixed-integer SOCP solvers.

### 5.1 N-Dimensional Expressions

Starting from version 1.6, CVXPY began supporting N-dimensional (N-d) expressions, intended to match the syntax of ndarrays from NumPy [23]. N-dimensional data structures—also referred to as *tensors*—are used widely across the scientific Python ecosystem (e.g., xarray [24], PyTorch [26], and JAX [10]). Previously, expressions involving more than two dimensions had to be implemented by hand, either through nested lists of variables or custom indexing logic for a stacked variable.

As the standard form requires a single vector variable, N-dimensional expressions must be flattened by the backend. CVXPY defaults internally to the Fortran order, or column-major order, for flattening higher-dimensional arrays. This process is identical to how we previously handled matrices.

#### 5.1.1 Sum Operator.

*Example.* Consider an expression  $\mathcal{X}$  with shape  $(2, 2, 2)$  being summed over a single axis  $s = \{1\}$ . The user considers  $\mathcal{X}$  as a vector of matrices:

$$\mathcal{X} = \left[ \begin{bmatrix} x_{111} & x_{112} \\ x_{121} & x_{122} \end{bmatrix}, \begin{bmatrix} x_{211} & x_{212} \\ x_{221} & x_{222} \end{bmatrix} \right]$$

The user expects to see the output of the sum operator  $\mathcal{Y}$  to be a matrix:

$$\mathcal{Y} = \begin{bmatrix} y_{11} & y_{12} \\ y_{21} & y_{22} \end{bmatrix} = \begin{bmatrix} x_{111} + x_{121} & x_{112} + x_{122} \\ x_{211} + x_{221} & x_{212} + x_{222} \end{bmatrix}.$$

In the CVXPY implementation, both expressions are stored in their vectorized form and the sum operator reduces to the following sparse matrix:

	$x_{111}$	$x_{211}$	$x_{121}$	$x_{221}$	$x_{112}$	$x_{212}$	$x_{122}$	$x_{222}$
$y_{11}$	1	0	1	0	0	0	0	0
$y_{21}$	0	1	0	1	0	0	0	0
$y_{12}$	0	0	0	0	1	0	1	0
$y_{22}$	0	0	0	0	0	1	0	1

### 5.1.2 Indexing Operator.

*Example.* Consider an arbitrary expression with shape  $(3, 2, 2)$  and the slice operation  $\mathcal{X}[0 : 2, :, 1]$ , which selects the first two slices along axis 0, all elements along axis 1, and index 1 along axis 2. First, we present the internal representation of the expression. The original expression is of the following form:

$$\mathcal{X} = \left[ \begin{bmatrix} x_{111} & x_{112} \\ x_{121} & x_{122} \end{bmatrix}, \begin{bmatrix} x_{211} & x_{212} \\ x_{221} & x_{222} \end{bmatrix}, \begin{bmatrix} x_{311} & x_{312} \\ x_{321} & x_{322} \end{bmatrix} \right].$$

The indexing operation  $\mathcal{X}[0:2, :, 1]$  produces a tensor  $\mathcal{Y}$  of shape  $(2, 2)$ :

$$\mathcal{Y} = \begin{bmatrix} y_{11} & y_{12} \\ y_{21} & y_{22} \end{bmatrix} = \begin{bmatrix} x_{112} & x_{122} \\ x_{212} & x_{222} \end{bmatrix}.$$

Internally, CVXPY represents the indexing operation as a sparse matrix that maps the vectorization of the input tensor to the vectorization of the selected subtensor:

	$x_{111}$	$\dots$	$x_{112}$	$x_{212}$	$x_{312}$	$x_{122}$	$x_{222}$	$x_{322}$
$y_{11}$	0	0	1	0	0	0	0	0
$y_{21}$	0	0	0	1	0	0	0	0
$y_{12}$	0	0	0	0	0	1	0	0
$y_{22}$	0	0	0	0	0	0	1	0

## 5.2 Sparsity Attribute

In version 1.6, CVXPY introduced a sparsity attribute for variables, which allows users to specify the nonzero entries of a variable. The sparsity attribute avoids storing unnecessary variables and provides performance improvements in both memory use and computation time during the canonicalization and solve steps. The syntax for specifying the sparse indices is the same as SciPy's `sparse.coo_array` format [34].

*Example.* Consider an arbitrary expression with shape  $(2, 2, 3)$  that has only three nonzero entries, at indices  $(0, 1, 0)$ ,  $(1, 0, 1)$ , and  $(1, 1, 2)$ . The user sees an expression of the form

$$\left[ \begin{bmatrix} 0 & 0 & 0 \\ z_1 & 0 & 0 \end{bmatrix}, \begin{bmatrix} 0 & z_2 & 0 \\ 0 & 0 & z_3 \end{bmatrix} \right]$$

Using the sparsity attribute, CVXPY can internally represent this expression with a compact vector of three entries  $\mathcal{Z} = (z_1, z_2, z_3)$ . When instantiating the leaves of the matrix stuffing tree, instead of using an identity matrix to represent

the  $\mathcal{Z}$  tensor, we use the following sparse matrix:

	$z_1$	$z_2$	$z_3$
$x_{000}$	0	0	0
$x_{100}$	0	0	0
$x_{010}$	1	0	0
$x_{110}$	0	0	0
$x_{001}$	0	0	0
$x_{101}$	0	1	0
$\vdots$	0	0	0
$x_{112}$	0	0	1

### 5.2.1 Defining a Sparse Variable.

```
# Create an upper triangular sparse variable
X = cp.Variable((10, 10), sparsity=np.triu_indices(n=10))
```

Another way to define a sparse variable is to use `np.where` on problem-specific data.

```
# Define problem data (adapt to your use case)
data = np.random.randn(10, 10)
# Create a sparse variable given condition on data
x = cp.Variable((10, 10), sparsity=np.where(data > 0.5))
```

Finally, you can also define the sparsity attribute manually. The input to the attribute needs to conform to the index format of `numpy.where` and `numpy.nonzero`.

```
# Create a sparse variable manually
# The two tuples correspond respectively to row and column indices
x = cp.Variable((3, 3), sparsity=[(0, 1), (0, 2)])
# This is equivalent to calling np.where(data == 1) on the following matrix
# data = np.array([[1, 0, 0],
#                  [0, 0, 1],
#                  [0, 0, 0]])
```

## 5.3 Multiple Attributes

In version 1.7, CVXPY began supporting multiple attributes for variables. CVXPY's variable attributes can be viewed as implicit constraints that provide useful sign and monotonicity information to the DCP analyzer. In addition, due to the large potential computational cost, projections for validating parameter inputs with multiple attributes are disabled. The following examples show some combinations of variable attributes that are supported in CVXPY 1.9.

### 5.3.1 Sparse Integer Variable.

```
x = cp.Variable((2, 2), sparsity=[(0, 1), (0, 1)], integer=True)
prob = cp.Problem(cp.Minimize(cp.sum(x)), [x >= -5.5])
prob.solve()
np.testing.assert_allclose(prob.value, -10)
```

```
# Although the lower bound on x is -5.5, the nonzero entries
# of x are set to -5 due to the integer attribute
# .value is a dense NumPy array; .value_sparse returns a COO array
np.testing.assert_allclose(x.value, np.array([[ -5,  0], [ 0, -5]]))
```

### 5.3.2 Bounded Integer Variable.

```
x = cp.Variable((2, 2), integer=True, bounds=[-1.5, 2])
prob = cp.Problem(cp.Minimize(cp.sum(x)), [])
prob.solve()
np.testing.assert_allclose(prob.value, -4)
# Although the lower bound is -1.5, each entry of x
# is set to -1 due to the integer attribute
np.testing.assert_allclose(x.value, np.array([[ -1, -1], [-1, -1]]))
```

### 5.3.3 Bounded Sparse Variable.

```
x = cp.Variable((3, 3), sparsity=[(0, 1), (1, 2)], bounds=[-3.5, 7])
prob = cp.Problem(cp.Minimize(cp.sum(x)), [])
prob.solve()
np.testing.assert_allclose(prob.value, -7)
# Each nonzero entry of x is set to the lower bound of -3.5
np.testing.assert_allclose(x.value, np.array([[0, -3.5, 0],
                                             [0, 0, -3.5],
                                             [0, 0, 0.0]]))
```

## 5.4 Quantum Information Modeling

CVXPY has added support for some key cones and atoms relevant for modeling problems arising in quantum information theory, such as calculating channel capacities or entanglement measures. These can broadly be categorized in two groups.

*Linear Algebra and Tensor Product Spaces.* Tensor product spaces of complex matrices are ubiquitous in quantum information theory. The familiar Kronecker product (available in CVXPY as the `kron` atom) is used to construct a matrix in a tensor product space  $W = U \otimes V$  given matrices in the generating spaces  $U$  and  $V$ .

When  $U$  and  $V$  are spaces of square matrices we arrive at important generalizations of the matrix trace and the matrix transpose. These generalizations can be defined in terms of orthonormal bases  $B_U$  and  $B_V$  for  $U$  and  $V$ , which generate the basis  $B_W = \{u \otimes v : u \in B_U, v \in B_V\}$  for  $W$ .

- The *partial trace* with respect to  $U$  is the function  $\text{Tr}_U : W \rightarrow V$  that maps  $u \otimes v \in B_W$  to  $\text{Tr}(u)v$ , and is then extended to all of  $W$  by  $\mathbb{C}$ -linearity; the partial trace with respect to  $V$  is the operator  $\text{Tr}_V : W \rightarrow U$  where  $\text{Tr}_V(u \otimes v) = \text{Tr}(v)u$ , again extended to all of  $W$  by linearity.
- The *partial transpose* (with respect to  $U$  or  $V$ ) is defined analogously. We emphasize that the partial transpose must use the *transpose*, not the *conjugate transpose*. The conjugate transpose is not a  $\mathbb{C}$ -linear operation.

CVXPY does not provide a first-class abstraction for vector spaces (tensor product, or otherwise). One implicitly works with tensor product spaces simply by keeping a list of dimensions of the generating spaces. In the context of quantum information, the generating spaces are called *subsystems*.

*Quantum Information Atoms.* These atoms implement key concave and convex functions from quantum information theory, which often form the objective function of an optimization problem. Matrices provided to these atoms must be Hermitian positive semidefinite (PSD).

- (1) `von_neumann_entr`. The von Neumann entropy,  $H(\rho) = -\text{Tr}(\rho \log \rho)$ , is the quantum mechanical analogue of Shannon entropy. The `von_neumann_entr` atom is concave.
- (2) `quantum_rel_entr`. The quantum relative entropy,  $D(\rho||\tau) = \text{Tr}(\rho(\log \rho - \log \tau))$ , is the quantum analogue of the Kullback-Leibler (KL) divergence. It is a non-negative measure of *distinguishability* between two quantum states  $\rho, \tau$  and  $D(\rho||\tau) = 0$  only if  $\rho = \tau$ . The quantum relative entropy is jointly convex in its arguments  $(\rho, \tau)$ .
- (3) `quantum_cond_entr`. Quantum conditional entropy is defined for a state  $\rho \in U \otimes V$  and a subsystem  $U$  or  $V$ . Conditioning with respect to  $V$ , it is equal to  $H(\rho) - H(\text{Tr}_V(\rho))$ . This is concave in  $\rho$ , but is not naturally DCP. Representing its hypograph requires quantum relative entropy.

We now demonstrate two quantum information problems that can be modeled with combinations of the atoms described above. These problems are borrowed from [18].

**Relative Entropy of Entanglement (REE).** The state of a two-qubit quantum register is described by a trace-1 Hermitian PSD matrix (a *density matrix*) of order 4. Such a state  $\rho$  is called *separable* if it can be expressed as a convex combination of tensor product states,  $\rho_0 \otimes \rho_1$ , where  $(\rho_0, \rho_1)$  are density matrices of order 2. States that are not separable are called *entangled*. The REE of a state  $\rho$  is the minimum quantum relative entropy achievable between  $\rho$  and a separable state  $\tau$  when we optimize over  $\tau$ . For a two-qubit register, we can compute  $\rho$ 's REE by optimizing over  $\tau$  that satisfy the *positive partial transpose* condition. It makes no difference whether the partial transpose is over the first or second qubit's subsystem.

```
# Define hypothetical input to the problem
U = np.random.randn(4, 4) + 1j * np.random.randn(4, 4)
psd = U @ U.T.conj()
rho = psd / np.trace(psd)
# Build and solve the problem
tau = cp.Variable((4, 4), hermitian=True)
func = cp.quantum_rel_entr(rho, tau) / np.log(2)
cons = [cp.partial_transpose(tau, [2, 2], 1) >> 0]
cons += [tau >> 0, cp.trace(tau) == 1]
prob = cp.Problem(cp.Minimize(func), cons)
prob.solve()
```

Listing 1. Computing the Relative Entropy of Entanglement (REE) in CVXPY

**Entanglement-Assisted Classical Capacity.** We try to calculate the fastest rate of transferring classical information (bits, 0s & 1s) over a noisy quantum channel  $\Phi$  when both the sender and receiver share an entangled state. This is quantified by using the *mutual information*,  $I(\rho, \Phi)$ , which can be implemented within CVXPY by a combination of the atoms `hermitian_wrap`, `quantum_cond_entr`, `von_neumann_entr`, and `partial_trace`.

```

# Define Phi by Stinespring rep., Phi(rho) = Tr_2(C rho C^\dagger)
C = np.array([
    [1, 0],
    [0, np.sqrt(0.01)],
    [0, np.sqrt(0.99)],
    [0, 0],
])
# Build and solve the CVXPY model
rho = cp.Variable((2, 2), hermitian=True)
sigma = cp.hermitian_wrap( C @ rho @ C.T.conj() )
Phi_rho = cp.hermitian_wrap(cp.partial_trace(sigma, [2, 2], 1))
I_rho_Phi = cp.quantum_cond_entr(sigma, [2, 2], 0)
I_rho_Phi = (I_rho_Phi + cp.von_neumann_entr(Phi_rho)) / np.log(2)
obj = cp.Maximize(I_rho_Phi)
prob = cp.Problem(obj, [rho >> 0, cp.trace(rho) == 1])
prob.solve()

```

Listing 2. Entanglement-Assisted Classical Capacity of a Quantum Channel in CVXPY

## 5.5 Approximating Logs and Exponentials

In §5.4, we outlined the various tools that CVXPY offers for modeling problems in quantum information theory. This subsection discusses the machinery behind these atoms and points out some compelling corollary uses.

*Operator Relative Entropy.* The operator relative entropy cone for order- $n$  Hermitian matrices,  $\mathbf{H}^n$ , is given by

$$K_{\text{re}}^n = \text{cl} \left\{ (X, Y, T) \in \mathbf{H}_{++}^n \times \mathbf{H}_{++}^n \times \mathbf{H}^n \mid \sqrt{X} \log(\sqrt{X} Y^{-1} \sqrt{X}) \sqrt{X} \preceq T \right\}.$$

The square roots and log in the display above are understood in the sense of spectral functions.

CVXPY’s `OpRelEntrConeQuad` class provides approximate representations of  $K_{\text{re}}^n$  using the techniques in the foundational work by Fawzi, Saunderson, and Parrilo [19]. The approximations, denoted  $K_{m,k}^n$  for integer parameters  $m, k$ , are constructed using operator monotonicity of certain rational functions and Gauss-Legendre quadrature. The main representation result, [19, Theorem 3], requires  $m + k + 1$  auxiliary Hermitian matrices and  $(m + k)$  PSD constraints of sizes  $2n \times 2n$  each. The parameter  $m$  controls the number of quadrature nodes and  $k$  controls the number of iterative square roots applied to the argument to improve the approximation’s accuracy, especially for values far from 1. The use of these approximate cones for quantum information modeling is described in [18].

`RelEntrConeQuad`. The operator relative entropy approximation scheme when  $n = 1$  provides a way to approximate the exponential cone. We introduced `RelEntrConeQuad` to handle this special case, with utility functions to map to and from the native `ExpCone`. The  $2 \times 2$  PSD constraints for `RelEntrConeQuad` are canonicalized into equivalent second-order cone constraints.

This canonicalization has two potential uses. The first is to open up the possibility of using SOCP solvers in place of less common exponential cone solvers. This is particularly valuable for mixed-integer problems, in light of open-source mixed-integer SOCP solvers such as SCIP [8]. The second potential use is in improving convergence for solvers that support the exponential cone directly.

*Market Equilibrium with Only Second-Order Cones.* In Listing 3, we show a market equilibrium problem with continuously divisible goods and linear utilities. This problem can be formulated as an exponential cone program [17].

```

# Valuations (V), budgets (b), and utilities.
buyers, items = 400, 200
V = 10 ** np.random.uniform(-2, 2, (buyers, items))
b = 10 ** np.random.uniform(-2, 1, buyers)
b /= np.sum(b)
allocations = cp.Variable((buyers, items), nonneg=True)
utilities = cp.sum(cp.multiply(V, allocations), axis=1)
# Maximize conic form of Nash Social Welfare
aux_var = cp.Variable(buyers)
aux_con = cp.ExpCone(aux_var, np.ones(buyers), utilities)
objective = cp.Maximize(b @ aux_var)
constraints = [
    aux_con.as_quad_approx(m=3, k=3), # Try dropping "as_quad_approx"
    cp.sum(allocations, axis=0) <= 1,
]
prob = cp.Problem(objective, constraints)
prob.solve(solver='CLARABEL', verbose=True)

```

Listing 3. Approximating an Exponential Cone Program to Help Solve a Market Equilibrium Problem

The code below builds the conic form directly, and shows how one can elect to replace the exponential cone with the `RelEntrConeQuad` approximation. The approximate form of the problem is more tractable for many open-source solvers, including Clarabel.

## 6 Conclusion

CVXPY has sought to improve its performance, to have its many features be increasingly cross-compatible, and to expand its modeling capabilities for new domains. We aim to finish the deprecation and removal of CVXCORE as we seek to further improve performance. We hope to develop a stable serialization format and to improve our infeasibility debugging support. Our sister library CVXPYlayers has greatly improved and expanded our GPU support in ways that will further the applicability of CVXPY across domains and fields.

## Acknowledgments

We thank Gridmatic for funding the development of N-dimensional arrays. We thank Google for Google Summer of Code. We thank NumFOCUS for facilitating our applications to GSoC as well as a developer fund to improve CVXPY's website. Finally, we would like to thank the numerous CVXPY contributors and solver interface maintainers.

Parth Nobel was supported in part by the National Science Foundation Graduate Research Fellowship Program under Grant No. DGE-1656518. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

## References

- [1] Akshay Agrawal, Brandon Amos, Shane Barratt, Stephen Boyd, Steven Diamond, and J. Zico Kolter. 2019. Differentiable convex optimization layers. In *Advances in Neural Information Processing Systems*. Curran Associates, Inc., Red Hook, NY, USA, 9558–9570.
- [2] Akshay Agrawal and Stephen Boyd. 2020. Disciplined quasiconvex programming. *Optimization Letters* 14 (2020), 1643–1657. doi:10.1007/s11590-020-01561-8
- [3] Akshay Agrawal, Steven Diamond, and Stephen Boyd. 2019. Disciplined geometric programming. *Optimization Letters* 13, 5 (2019), 961–976.

- [4] Akshay Agrawal, Robin Verschueren, Steven Diamond, and Stephen Boyd. 2018. A rewriting system for convex optimization problems. *Journal of Control and Decision* 5, 1 (2018), 42–60.
- [5] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. 2006. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison-Wesley Longman Publishing Co., Inc., USA.
- [6] Alnur Ali, J. Zico Kolter, Steven Diamond, and Stephen Boyd. 2015. Disciplined convex stochastic programming: a new framework for stochastic optimization. In *Proceedings of the Thirty-First Conference on Uncertainty in Artificial Intelligence (Amsterdam, Netherlands) (UAI'15)*. AUA Press, Arlington, Virginia, USA, 62–71.
- [7] MOSEK ApS. 2025. *MOSEK Optimization Suite*. MOSEK ApS. <https://docs.mosek.com/latest/intro/index.html>
- [8] Suresh Bolusani, Mathieu Besançon, Ksenia Bestuzheva, Antonia Chmiela, João Dionísio, Tim Donkiewicz, Jasper van Doornmalen, Leon Eifler, Mohammed Ghannam, Ambros Gleixner, Christoph Graczyk, Katrin Halbig, Ivo Hedtke, Alexander Hoen, Christopher Hojny, Rolf van der Hulst, Dominik Kamp, Thorsten Koch, Kevin Kofler, Jurgen Lentz, Julian Manns, Gioni Mexi, Erik Mühmer, Marc E. Pfetsch, Franziska Schlösser, Felipe Serrano, Yuji Shinano, Mark Turner, Stefan Vigerske, Dieter Weninger, and Lixing Xu. 2024. *The SCIP Optimization Suite 9.0*. ZIB-Report 24-02-29. Zuse Institute Berlin. <https://nbn-resolving.org/urn:nbn:de:0297-zib-95528>
- [9] Stephen Boyd and Lieven Vandenbergh. 2004. *Convex Optimization*. Cambridge University Press, Cambridge, UK.
- [10] James Bradbury, Roy Frostig, Peter Hawkins, Matthew James Johnson, Chris Leary, Dougal Maclaurin, George Necula, Adam Paszke, Jake VanderPlas, Skye Wanderman-Milne, and Qiao Zhang. 2018. JAX: composable transformations of Python+NumPy programs. <http://github.com/google/jax>
- [11] Richard H Byrd, Jorge Nocedal, and Richard A Waltz. 2006. KNITRO: An integrated package for nonlinear optimization. In *Large-Scale Nonlinear Optimization*. Springer, Boston, MA, USA, 35–59.
- [12] Daniel Cederberg, William Zhang, Parth Nobel, and Stephen Boyd. 2026. Disciplined Nonlinear Programming. arXiv:2606.02896 [math.OC] <https://arxiv.org/abs/2606.02896>
- [13] Yuwen Chen, Danny Tse, Parth Nobel, Paul Goulart, and Stephen Boyd. 2024. CuClarabel: GPU Acceleration for a Conic Optimization Solver. arXiv:2412.19027 [math.OC] <https://arxiv.org/abs/2412.19027>
- [14] Steven Diamond and Stephen Boyd. 2016. CVXPY: A Python-embedded modeling language for convex optimization. *Journal of Machine Learning Research* 17, 83 (2016), 1–5. Based on CVXPY version 0.3.
- [15] Alexander Domahidi, Eric Chu, and Stephen Boyd. 2013. ECOS: An SOCP solver for embedded systems. In *2013 European Control Conference (ECC)*. IEEE, Zurich, Switzerland, 3071–3076.
- [16] Michael Droettboom, Pauli Virtanen, and asv Developers. 2025. airspeed velocity (asv): A simple Python benchmarking tool with web-based reporting. <https://github.com/airspeed-velocity/asv> Accessed: 2025-06-22.
- [17] E. Eisenberg. 1961. Aggregation of Utility Functions. *Management Science* 7, 4 (July 1961), 337–350. doi:10.1287/mnsc.7.4.337
- [18] Hamza Fawzi and Omar Fawzi. 2018. Efficient optimization of the quantum relative entropy. *Journal of Physics A: Mathematical and Theoretical* 51, 15 (March 2018), 154003. doi:10.1088/1751-8121/aab285
- [19] Hamza Fawzi, James Saunderson, and Pablo A. Parrilo. 2018. Semidefinite Approximations of the Matrix Logarithm. *Foundations of Computational Mathematics* 19, 2 (March 2018), 259–296. doi:10.1007/s10208-018-9385-0
- [20] D. Ge, Q. Huangfu, Z. Wang, J. Wu, and Y. Ye. 2022. Cardinal Optimizer (COPT) user guide. <https://guide.coap.online/copt/en-doc>.
- [21] Paul J. Goulart and Yuwen Chen. 2024. Clarabel: An interior-point solver for conic programs with quadratic objectives. arXiv:2405.12762 [math.OC] <https://arxiv.org/abs/2405.12762>
- [22] Michael Grant. 2004. *Disciplined Convex Programming*. Ph. D. Dissertation. Stanford University.
- [23] Charles R. Harris, K. Jarrod Millman, Stéfan J. van der Walt, Ralf Gommers, Pauli Virtanen, David Courmepau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fernández del Río, Mark Wiebe, Pearu Peterson, Pierre Gérard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant. 2020. Array programming with NumPy. *Nature* 585, 7825 (Sept. 2020), 357–362. doi:10.1038/s41586-020-2649-2
- [24] Stephan Hoyer and Joe Hamman. 2017. xarray: N-D labeled Arrays and Datasets in Python. *Journal of Open Research Software* 5, 1 (Apr 2017), 10. doi:10.5334/jors.148
- [25] Brendan O’Donoghue. 2021. Operator splitting for a homogeneous embedding of the linear complementarity problem. arXiv:2004.02177 [math.OC] <https://arxiv.org/abs/2004.02177>
- [26] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Yang, Zach DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. arXiv:1912.01703 [cs.LG] <https://arxiv.org/abs/1912.01703>
- [27] M. Schaller, G. Banjac, S. Diamond, A. Agrawal, B. Stellato, and S. Boyd. 2022. Embedded code generation with CVXPY. *IEEE Control Systems Letters* 6 (2022), 2653–2658.
- [28] Maximilian Schaller and Stephen Boyd. 2025. Code generation for solving and differentiating through convex optimization problems. arXiv:2504.14099 [math.OC] <https://arxiv.org/abs/2504.14099>
- [29] Philipp Schiele, Eric Luxenberg, and Stephen Boyd. 2024. Disciplined Saddle Programming. *Transactions on Machine Learning Research* (2024), 1–25. <https://openreview.net/forum?id=KhMLfeloUm>

- [30] Xinyue Shen, Steven Diamond, Yuantao Gu, and Stephen Boyd. 2016. Disciplined convex-concave programming. In *2016 IEEE 55th Conference on Decision and Control (CDC)*. IEEE, Las Vegas, NV, USA, 1009–1014. doi:10.1109/CDC.2016.7798400
- [31] Xinyue Shen, Steven Diamond, Madeleine Udell, Yuantao Gu, and Stephen Boyd. 2017. Disciplined multi-convex programming. In *2017 29th Chinese Control And Decision Conference (CCDC)*. IEEE, Chongqing, China, 895–900. doi:10.1109/CCDC.2017.7978647
- [32] B. Stellato, G. Banjac, P. Goulart, A. Bemporad, and S. Boyd. 2020. OSQP: an operator splitting solver for quadratic programs. *Mathematical Programming Computation* 12, 4 (2020), 637–672. doi:10.1007/s12532-020-00179-2
- [33] Charlie Vanaret and Sven Leyffer. 2024. Implementing a unified solver for nonlinearly constrained optimization. arXiv:2406.13454 [math.OC] <https://arxiv.org/abs/2406.13454> To appear in *Mathematical Programming Computation*.
- [34] Pauli Virtanen, Ralf Gommers, Travis E. Oliphant, Matt Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski, Pearu Peterson, Warren Weckesser, Jonathan Bright, Stéfan J. van der Walt, Matthew Brett, Joshua Wilson, K. Jarrod Millman, Nikolay Mayorov, Andrew R. J. Nelson, Eric Jones, Robert Kern, Eric Larson, C J Carey, İlhan Polat, Yu Feng, Eric W. Moore, Jake VanderPlas, Denis Laxalde, Josef Perktold, Robert Cimrman, Ian Henriksen, E. A. Quintero, Charles R. Harris, Anne M. Archibald, António H. Ribeiro, Fabian Pedregosa, Paul van Mulbregt, Aditya Vijaykumar, Alessandro Pietro Bardelli, Alex Rothberg, Andreas Hilboll, Andreas Kloeckner, Anthony Scopatz, Antony Lee, Ariel Rokem, C. Nathan Woods, Chad Fulton, Charles Masson, Christian Häggström, Clark Fitzgerald, David A. Nicholson, David R. Hagen, Dmitrii V. Pasechnik, Emanuele Olivetti, Eric Martin, Eric Wieser, Fabrice Silva, Felix Lenders, Florian Wilhelm, G. Young, Gavin A. Price, Gert-Ludwig Ingold, Gregory E. Allen, Gregory R. Lee, Hervé Audren, Irvin Probst, Jörg P. Dietrich, Jacob Silterra, James T Webber, Janko Slavič, Joel Nothman, Johannes Buchner, Johannes Kulick, Johannes L. Schönberger, José Vinicius de Miranda Cardoso, Joscha Reimer, Joseph Harrington, Juan Luis Cano Rodríguez, Juan Nunez-Iglesias, Justin Kuczynski, Kevin Tritz, Martin Thoma, Matthew Newville, Matthias Kümmerer, Maximilian Bolingbroke, Michael Tartre, Mikhail Pak, Nathaniel J. Smith, Nikolai Nowaczyk, Nikolay Shebanov, Oleksandr Pavlyk, Per A. Brodtkorb, Perry Lee, Robert T. McGibbon, Roman Feldbauer, Sam Lewis, Sam Tygier, Scott Sievert, Sebastiano Vigna, Stefan Peterson, Surhud More, Tadeusz Pudlik, Takuya Oshima, Thomas J. Pingel, Thomas P. Robitaille, Thomas Spura, Thouis R. Jones, Tim Cera, Tim Leslie, Tiziano Zito, Tom Krauss, Utkarsh Upadhyay, Yaroslav O. Halchenko, and Yoshiki Vázquez-Baeza. 2020. SciPy 1.0: fundamental algorithms for scientific computing in Python. *Nature Methods* 17, 3 (Feb. 2020), 261–272. doi:10.1038/s41592-019-0686-2
- [35] A. Wächter and L. Biegler. 2006. On the implementation of an interior-point filter line-search algorithm for large-scale nonlinear programming. *Mathematical Programming* 106, 1 (2006), 25–57. doi:10.1007/s10107-004-0559-y