Code Generation for Solving and Differentiating through Convex Optimization Problems

Maximilian Schaller and Stephen Boyd

April 18, 2025

Abstract

We introduce custom code generation for parametrized convex optimization problems that supports evaluating the derivative of the solution with respect to the parameters, *i.e.*, differentiating through the optimization problem. We extend the open source code generator CVXPYgen, which itself extends CVXPY, a Python-embedded domain-specific language with a natural syntax for specifying convex optimization problems, following their mathematical description. Our extension of CVXPYgen adds a custom C implementation to differentiate the solution of a convex optimization problem with respect to its parameters, together with a Python wrapper for prototyping and desktop (non-embedded) applications. We give three representative application examples: Tuning hyper-parameters in machine learning; choosing the parameters in an approximate dynamic programming (ADP) controller; and adjusting the parameters in an optimization based financial trading engine via back-testing, *i.e.*, simulation on historical data. While differentiating through convex optimization problems is not new, CVXPYgen is the first tool that generates custom C code for the task, and increases the computation speed by about an order of magnitude in most applications, compared to CVXPYlayers, a general-purpose tool for differentiating through convex optimization problems.

1 Introduction

A convex optimization problem, parametrized by $\theta \in \Theta \subseteq \mathbf{R}^d$, can be written as

$$\begin{array}{ll} \text{minimize} & f_0(x,\theta) \\ \text{subject to} & f_i(x,\theta) \leq 0, \quad i=1,\ldots,m \\ & A(\theta)x = b(\theta), \end{array} \tag{1}$$

where $x \in \mathbf{R}^n$ is the optimization variable, f_0 is the objective function to be minimized, which is convex in x, and $f_1,\ldots,f_m$ are inequality constraint functions that are convex in x [BV04].

The parameter θ specifies data that can change, but is constant and given (or chosen) when we solve an instance of the problem. We refer to the parametrized problem (1) as a *problem family*; when we specify a fixed value of $\theta \in \Theta$, we refer to it as a *problem instance*. We let $x^\star$ denote an optimal point for problem (1), assuming it exists. To emphasize its dependence on θ, we write it as $x^\star(\theta)$.

Convex optimization is used in many domains, including signal and image processing [MB10, ZE10], machine learning [Mur12, BN06, ZH05, Tib96, Cor95, HK70, Cox58], control systems [RMD$^+$17, KC16, WOB15, KB14, WB09, BEGFB94, BB91, GPM89], quantitative finance [Pal25, BJK$^+$24, BBD$^+$17, Nar13, LFB07, GK00, Mar52], and operations research [Hal19, BG92, Ber91, LAHH16, BT04].

1.1 Differentiating through convex optimization problems

In many applications we are interested in the sensitivity of the solution $x^\star$ with respect to the parameter θ. We will assume there is a unique solution for parameters near θ, and that the mapping from θ to $x^\star$ is differentiable with Jacobian $\partial x^\star / \partial \theta \in \mathbf{R}^{n \times d}$, evaluated at θ. We then have

$$\Delta x^\star \approx \frac{\partial x^\star}{\partial \theta} \Delta \theta,$$

where $\Delta \theta$ is the change in θ, and $\Delta x^\star$ is the resulting change in $x^\star$.

We make a few comments on our assumptions. First, the solution $x^\star$ need not be unique, and so does not define a function from θ to $x^\star$. Even when the solution is unique for each parameter value, the mapping from θ to $x^\star$ need not be differentiable. Following universal practice in machine learning, we simply ignore these issues. When $x^\star$ is not unique, or when the mapping is not differentiable, we simply use some reasonable value for the (nonexistent) derivative. It has been observed that simple gradient (or subgradient) based methods for optimizing parameters are tolerant of these approximations.

Approximating the change in solution with a change in parameters can be useful by itself in some applications. As an example, consider a machine learning problem where we fit the parameters of a model to data by minimizing the sum of a convex loss function over the given training data. Considering the training data as a parameter, and the solution as the estimated model parameters, the Jacobian above gives us the sensitivity of each model parameter with respect to the training data. In particular, these sensitivities are sometimes used to compute a risk estimate for the learned model parameters [NCB23, NLC24]. As another example, suppose we model some economic variables (*e.g.*, consumption, demand for products, trades) as maximizing a concave utility function that depends on parameters. The Jacobian here directly gives us an approximation of the change in demand (say) when the utility parameters change [Wai05].

1.2 Autodifferentiation framework

The solution map derivative is much more useful when it is part of an autodifferentiation system such as JAX [BFH$^+$18], PyTorch [PGC$^+$17], or Tensorflow [ABC$^+$16]. We consider a scalar function that is described by its compution graph, which can include standard operations and functions, as well the solution of one or more convex optimization problems. We can compute the gradient of this function automatically, and this can be used for applications such as tuning or optimizing the performance of a system. We give a few simple generic examples here.

In machine learning, we fit model parameters (also called weights or coefficents) using convex optimization, but we may have other hyper-parameters (*e.g.*, that scale regularization terms) that we would like to tune to get good performance on an unseen, out-of-sample test data set [Mur12]. The scalar function that we differentiate is the loss function computed with test data, and we differentiate with respect to the hyper-parameters of the machine learning model. A similar situation occurs in finance, where the actual trades to execute are determined by solving a parameterized problem [BBD$^+$17], which also contains a number of hyper-parameters that set limits on the portfolio or trading, or scale objective terms, and our goal is to obtain good performance on a simulation that uses historical data, *i.e.*, a back-test. In this case, the scalar function that we optimize might be a metric like the realized portfolio return or the Sharpe ratio [LW08], and we differentiate with respect to the hyper-parameters of the portfolio construction model.

1.3 Related work

Differentiating through convex optimization problems. There are two main classes of methods to differentiate the mapping from problem parameters to the solution. First, autodifferentiation software like PyTorch [PGC$^+$17] and Tensorflow [ABC$^+$16], as commonly used in backpropagation for machine learning, would differentiate through all instructions of an iterative optimization algorithm. While these tools are commonly used in the deep learning domain, they neglect the structure and optimality conditions of, particularly, convex optimization problems.

Second, as less of a brute-force approach, one can directly differentiate through the optimality conditions of a convex optimization problem, also referred to as *argmin differentiation* [ABB$^+$19, AK17]. CVXPYlayers [AAB$^+$19] is based on prior work on differentiating through a cone program, called diffcp [ABB$^+$19], and provides interfaces to PyTorch and Tensorflow. OptNet [AK17] addresses quadratic programs. Least squares auto-tuning [BB21] is a specialized hyper-parameter tuning framework for least squares problems, *i.e.*, a subclass of quadratic programs, with parametrized problem data. It admits flexible tuning objectives and tuning regularizers (called hyper-hyper-parameters), and computes their gradient with respect to the parameters of the least squares problem. PyEPO [TK24] combines various autodifferentiation and argmin differentiation tools into one software suite.

Tuning systems that involve convex optimization. Differentiating through a convex problem is useful in a broad array of applications sometimes called predict-then-optimize. In these applications we make some forecast or prediction (using convex optimization or other machine learning models), and then take some action (using convex optimization), and we are interested in the gradient of parameters appearing both in the predictor and the action policy [TK24, EG22]. In some cases, the prediction and optimization steps are fused into one black-box model that maps features to actions, called *learning to optimize from features* [KDVC$^+$24], which avoids differentiating through an optimization problem. This approximation is not necessary when it is possible to differentiate through the optimization problem in an easy and fast way, which motivates this work.

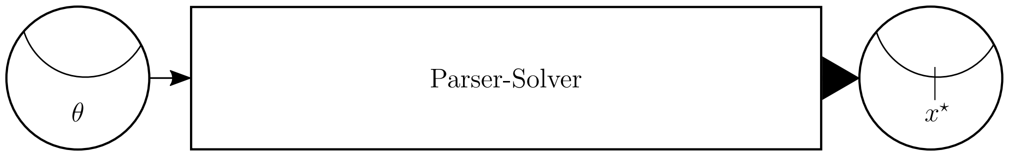
We also mention that when the dimension of θ is small enough, the parameters can be optimized using zero-order or derivative-free methods, which do not require the gradient of the overall metric with respect to the parameters [ABBS20, AAB$^+$19]. Examples include Optuna [ASY$^+$19], HOLA [MBK$^+$22], for tuning parameters with respect to some overall metric, and Hyperband [LJD$^+$18], which dynamically allocates resources for efficient hyper-parameter search in machine learning. Even when a tuning problem can be reasonably carried out using derivative-free methods, the ability to evaluate the gradient can give faster convergence with fewer evaluations.

Domain-specific languages for convex optimization. Argmin differentiation tools like CVXPYlayers admit convex optimization problems that are specified in a domain-specific language (DSL). Such systems allow the user to specify the functions f_i and A and b, in a simple format that closely follows the mathematical description of the problem. Examples include YALMIP [LÖ4] and CVX [GB14] (in Matlab), CVXPY [DB16] (in Python), which CVXPYlayers is based on, Convex.jl [UMZ$^+$14] and JuMP [DHL17] (in Julia), and CVXR [FNB20] (in R). We focus on CVXPY, which also supports the declaration of parameters, enabling it to specify problem families, not just problem instances.

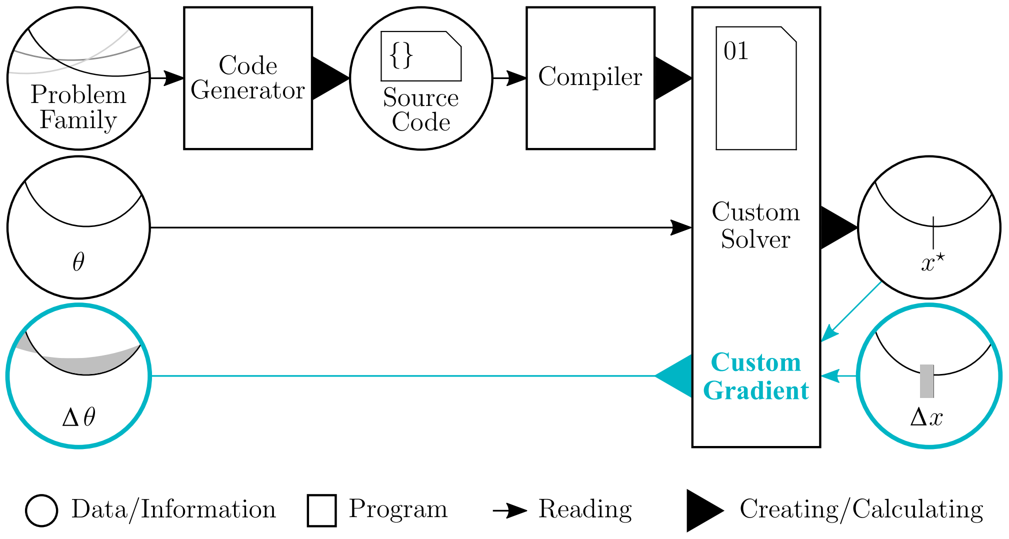
DSLs parse the problem description and translate (canonicalize) it to an equivalent problem that is suitable for a solver that handles some generic class of problems, such as linear programs (LPs), quadratic programs (QPs), second-order cone programs (SOCPs), semidefinite programs (SDPs), and others such as exponential cone programs [BV04]. Our work focuses on LPs and QPs. After the canonicalized problem is solved, a solution of the original problem is retrieved from a solution of the canonicalized problem.

It is useful to think of the whole process as a function that maps θ, the parameter that specifies the problem instance, into $x^\star$, an optimal value of the variable. With a DSL, this process consists of three steps. First the original problem description is canonicalized to a problem in some standard (canonical) form; then the canonicalized problem is solved using a solver; and finally, a solution of the original problem is retrieved from a solution of the canonicalized problem. When differentiating through the problem, the sequence of canonicalization, canonical solving, and retrieval is reversed. Reverse retrieval is followed by canonical differentiating and reverse canonicalization.

Most DSLs are organized as *parser-solvers*, which carry out the canonicalization each time



(a) Parser-solver calculating solution $x^\star$ for problem instance with parameter θ.



(b) Source code generation for problem family, followed by compilation to custom solver and custom gradient computation (new, signified with blue color). The compiled solver computes a solution $x^\star$ to the problem instance with parameter θ. The compiled differentiator computes the gradient $\Delta\theta$ given Δx.

Figure 1: Comparison of convex optimization problem parsing and solving/differentiating approaches.

the problem is solved (with different parameter values). This simple setting is illustrated in figure 1a.

Code generation for convex optimization. We are interested in applications where we solve many instances of the problem, possibly in an embedded application with hard real-time constraints, or a non-embedded application with limited compute. For such applications, a *code generator* makes more sense.

A code generator takes as input a description of a problem family, and generates specialized source code for that specific family. That source code is then compiled, and we have an efficient solver for the specific family. In this work, we add a program that efficiently computes the gradient of the parameter-solution mapping. The overall workflow is illustrated in figure 1b. The compiled solver and differentiator have a number of advantages over parser-solvers. First, by caching canonicalization and exploiting the problem structure, the compiled solver and the compiled differentiator are faster. Second, the compiled solver and in some applications also the compiled differentiator can be deployed in embedded systems, fulfilling rules for safety-critical code [Hol06].

1.4 Contribution

In this paper, we extend the code generator CVXPYgen [SBD+22] to produce source code for differentiating the parameter-solution mapping of convex optimization problems that can be reduced to QPs. We allow for the use of any canonical solver that is supported by CVXPYgen, including conic solvers. We combine existing theory on differentiating through the optimality conditions of a QP [AK17] with low-rank updates to the factorization of quasidefinite systems [DH99, DH05] to enable very fast repeated differentiation. Along with the generated C code, we compile two Python interfaces, one for use with CVXPY and one for use with CVXPYlayers. To the best of our knowledge, CVXPYgen is the first code generator for convex optimization that supports differentiation.

We give three examples, tuning the hyper-parameters and feature engineering parameters of a machine learning model, tuning the controller weights of an approximate dynamic programming controller, and adjusting the parameters in a financial trading engine. CVXPYgen accelerates these applications by around an order of magnitude.

1.5 Outline

The remainder of this paper is structured as follows. In §2 we describe, at a high level, how CVXPYgen generates code to differentiate through convex optimization problems. In §3 we describe the generic system tuning framework that runs the CVXPYgen solvers and differentiators, and in §4, we illustrate how we use it for three realistic examples that compare the performance of CVXPYgen to CVXPYlayers. We conclude the paper in §5.

2 Differentiating with CVXPYgen

CVXPYgen is an open-source code generator, based on the Python-embedded domain-specific language CVXPY. While CVXPY treats all typical conic programs and CVXPYgen generates code to solve LPs, QPs, and SOCPs, we focus on differentiating through problems that can be reduced to a QP, *i.e.*, LPs and QPs.

2.1 Disciplined parametrized programming

The CVXPY language uses disciplined convex programming (DCP) to allow for modeling instructions that are very close to the mathematical problem description and to verify convexity in a systematic way [DB16]. Disciplined parametrized programming (DPP) is an extension to the DCP rules for modeling convex optimization problems. While a DCP problem is readily canonicalized, a DPP problem is readily canonicalized with *affine* mappings from the user-defined parameters to the canonical parameters. Similarly, the mapping from a canonical solution back to a solution to the user-defined problem is affine for DPP problems [AAB+19]. DPP imposes mild restrictions on how parameters enter the problem expressions. In short, if all parameters enter the problem expressions in an affine way, the problem is DPP. We model all example problems in §4 DPP and illustrate standard modifications to make DCP problems DPP. Details on the DCP and DPP rules can be found at `https://www.cvxpy.org`.

2.2 Differentiating through parametrized problems

Differentiating through a DPP problem consists of three steps: affine parameter canonicalization, canonical solving, and affine solution retrieval,

$$\tilde{\theta} = C\theta + c, \qquad \tilde{x}^\star = \mathcal{S}(\tilde{\theta}), \qquad x^\star = R\tilde{x}^\star + r,$$

where θ is the user-defined parameter, C and R are sparse matrices, and $\mathcal{S}(\cdot)$ is the canoncial solver. We mark the canonical parameter and solution with a tilde.

In this work, we want to propagate a gradient in terms of the current solution, called Δx, to a gradient in the parameters $\Delta\theta$. This mapping is symmetric to the solution mapping [AAB+19], *i.e.*,

$$\Delta\tilde{x} = R^T \Delta x, \qquad \Delta\tilde{\theta} = (\mathsf{D}^T\mathcal{S})(\Delta\tilde{x}; \tilde{x}^\star, \tilde{\theta}), \qquad \Delta\theta = C^T\Delta\tilde{\theta},$$

and we can simply re-use the descriptions of R and C that CVXPYgen has already extracted for solving the problem. The following section explains how we (re)compute the canonical derivative $(\mathsf{D}^T\mathcal{S})(\Delta\tilde{x}; \tilde{x}^\star, \tilde{\theta})$ efficiently.

2.3 Differentiating through canonical solver

We focus on differentiating through LPs and QPs, *i.e.*, problems that can be reduced to the QP standard form

$$\begin{array}{ll} \text{minimize} & (1/2)\tilde{x}^T P\tilde{x} + q^T\tilde{x} \\ \text{subject to} & l \leq A\tilde{x} \leq u, \end{array}$$

as used by the OSQP solver [SBG+20]. The variable is $\tilde{x} \in \mathbf{R}^{\tilde{n}}$ and all other symbols are parameters. The objective is parametrized with $P \in \mathbf{S}_+^{\tilde{n}}$, where $\mathbf{S}_+^{\tilde{n}}$ is the set of symmetric positive semidefinite matrices, and $q \in \mathbf{R}^{\tilde{n}}$. The constraints are parametrized with $A \in \mathbf{R}^{m\times\tilde{n}}$, $l \in \mathbf{R}^m \cup \{-\infty\}$, and $u \in \mathbf{R}^m \cup \{\infty\}$. If an entry of l or u is $-\infty$ or ∞, respectively, it means there is no constraint. A pair of equal entries $l_i = u_i$ represents an equality constraint.

We closely follow the approach that is used in OptNet [AK17]. We denote by $A_{\mathcal{C}}$ the row slice of A that contains all rows A_i for which the lower or upper constraint is active at optimality, *i.e.*, it holds that $A_i\tilde{x} = l_i$ or $A_i\tilde{x} = u_i$ (or both, in the case of an equality constraint). We omit the superscript $\star$ for brevity. We set $b_{\mathcal{C}}$ to contain the entries of l or u at the active constraints indices, *i.e.*, $A_{\mathcal{C}}\tilde{x} = b_{\mathcal{C}}$. Then, the solution is characterized by the KKT system

$$\begin{bmatrix} P & A_{\mathcal{C}}^T \\ A_{\mathcal{C}} & 0 \end{bmatrix} \begin{bmatrix} \tilde{x} \\ \tilde{y}_{\mathcal{C}} \end{bmatrix} = \begin{bmatrix} -q \\ b_{\mathcal{C}} \end{bmatrix}, \tag{2}$$

where $\tilde{y}_{\mathcal{C}}$ is the slice of the dual variable corresponding to the active constraints. Note that we use the sign of $\tilde{y}$ to determine constraint activity. (The first block row of system (2) corresponds to stationarity of the Lagrangian, and the second block row corresponds to primal feasibility.)

We take the differential of (2) and re-group the terms as

$$\begin{bmatrix} P & A_{\mathcal{C}}^T \\ A_{\mathcal{C}} & 0 \end{bmatrix} \begin{bmatrix} \mathrm{d}\tilde{x} \\ \mathrm{d}\tilde{y}_{\mathcal{C}} \end{bmatrix} = \begin{bmatrix} -\mathrm{d}P\tilde{x} - \mathrm{d}A_{\mathcal{C}}^T\tilde{y}_{\mathcal{C}} - \mathrm{d}q \\ -\mathrm{d}A_{\mathcal{C}}\tilde{x} + \mathrm{d}b_{\mathcal{C}} \end{bmatrix}.$$

We introduce

$$\begin{bmatrix} d_x \\ d_y \end{bmatrix} = \begin{bmatrix} P & A_{\mathcal{C}}^T \\ A_{\mathcal{C}} & 0 \end{bmatrix}^{-1} \begin{bmatrix} \Delta\tilde{x} \\ 0 \end{bmatrix}, \tag{3}$$

where the righthand side is evaluated using algorithm 1. To avoid singularity of the linear system, we regularize the matrix diagonal with a small $\epsilon > 0$, solve the regularized system and add N^{refine} steps of iterative refinement [CH18, Hig97] to correct for the effect of ϵ on the solution.

Algorithm 1 Regularized system solve

```
1: Initialize $P$, $A_{\mathcal{C}}$, $\Delta\tilde{x}$
2: $K_{\mathcal{C}} = \begin{bmatrix} P & A_{\mathcal{C}}^T \\ A_{\mathcal{C}} & 0 \end{bmatrix}$,   $K_{\mathcal{C}}^{\epsilon} = K_{\mathcal{C}} + \begin{bmatrix} \epsilon I & 0 \\ 0 & -\epsilon I \end{bmatrix}$,   $r = \begin{bmatrix} \Delta\tilde{x} \\ 0 \end{bmatrix}$
3: $z = (K_{\mathcal{C}}^{\epsilon})^{-1} r$
4: for $i = 1$ to $N^{\text{refine}}$ do
5:     $\delta_r = r - K_{\mathcal{C}} z$
6:     $\delta_z = (K_{\mathcal{C}}^{\epsilon})^{-1} \delta_r$
7:     $z \leftarrow z + \delta_z$
8: end for
9: $(d_x, d_y) = z$
```

The regularization strength $\epsilon = 10^{-6}$ and $N^{\text{refine}} = 3$ iterations of iterative refinement work well in most practical cases.

Ultimately, the gradients in the QP parameters are

$$\Delta P = -(1/2)(d_x\tilde{x}^T + \tilde{x}d_x^T), \qquad \Delta q = -d_x, \qquad \Delta A_{\mathcal{C}} = -(d_y\tilde{x}^T + \tilde{y}_{\mathcal{C}}d_x^T), \qquad \Delta b_{\mathcal{C}} = d_y.$$

We copy the rows of $\Delta A_{\mathcal{C}}$ and $\Delta b_{\mathcal{C}}$ into the corresponding rows of ΔA and Δb, respectively, and set all other entries of ΔA and Δb to zero.

Low-rank updates to factorization of linear system. For the quasidefinite matrix $K_{\mathcal{C}}^{\epsilon}$ used in algorithm 1, there always exists an LDL-factorization [Van95],

$$L_{\mathcal{C}} D_{\mathcal{C}} L_{\mathcal{C}}^T = K_{\mathcal{C}}^{\epsilon} = \begin{bmatrix} P + \epsilon I & A_{\mathcal{C}}^T \\ A_{\mathcal{C}} & -\epsilon I \end{bmatrix}.$$

If the entries of P or A change, we perform a full re-factorization. Otherwise, we use the fact that the factors $L_{\mathcal{C}}$ and $D_{\mathcal{C}}$ change with the set of active constraints, denoted by $\mathcal{C}$. For the constraints that switch from inactive to active or vice-versa, we perform a sequence of rank-1 updates to $L_{\mathcal{C}}$ and $D_{\mathcal{C}}$.

We re-write the LDL-factorization as

$$\begin{bmatrix} L_{11} & 0 & 0 \\ \bar{l}_{12}^T & 1 & 0 \\ L_{31} & \bar{l}_{32} & \bar{L}_{33} \end{bmatrix} \begin{bmatrix} D_{11} & 0 & 0 \\ 0 & \bar{d}_{22} & 0 \\ 0 & 0 & \bar{D}_{33} \end{bmatrix} \begin{bmatrix} L_{11}^T & \bar{l}_{12} & L_{31}^T \\ 0 & 1 & \bar{l}_{32}^T \\ 0 & 0 & \bar{L}_{33}^T \end{bmatrix} = \begin{bmatrix} K_{11} & \bar{k}_{12} & K_{31}^T \\ \bar{k}_{12}^T & \bar{k}_{22} & \bar{k}_{32}^T \\ K_{31} & \bar{k}_{32} & K_{33} \end{bmatrix},$$

where lowercase symbols marked with a bar denote row/column combinations that are added or deleted. Uppercase symbols marked with a bar are sub-matrices that will be altered due to the addition or deletion.

For a constraint that switches from inactive to active, we add the respective row/column combination to $K_{\mathcal{C}}^{\epsilon}$ and run algorithm 2. If a constraint switches from active to inactive, we

Algorithm 2 Row/column addition (variant of algorithm 1 in [DH05])

```
1: Solve the lower triangular system $L_{11}D_{11}\bar{l}_{12} = \bar{k}_{12}$ for $\bar{l}_{12}$
2: $\bar{d}_{22} = \bar{k}_{22} - \bar{l}_{12}^T D_{11}\bar{l}_{12}$
3: $\bar{l}_{32} = (\bar{k}_{32} - L_{31}D_{11}\bar{l}_{12})/\bar{d}_{22}$
4: $w = \bar{l}_{32}(-\bar{d}_{22})^{1/2}$
5: Perform the rank-1 downdate $\bar{L}_{33}\bar{D}_{33}\bar{L}_{33}^T = L_{33}D_{33}L_{33}^T - ww^T$ according to algorithm 5
   in [DH99]
```

run algorithm 3 for row/column deletion. All steps of the row/column addition and deletion

Algorithm 3 Row/column deletion (variant of algorithm 2 in [DH05])

```
1: $w = \bar{l}_{32}(-\bar{d}_{22})^{1/2}$
2: $\bar{l}_{12} = 0$
3: $\bar{d}_{22} = 1$
4: $\bar{l}_{32} = 0$
5: Perform the rank-1 update $\bar{L}_{33}\bar{D}_{33}\bar{L}_{33}^T = L_{33}D_{33}L_{33}^T + ww^T$ according to algorithm 5
   in [DH99]
```

algorithms operate on the sparse matrices $K_{\mathcal{C}}^{\epsilon}$ and $L_{\mathcal{C}}$ stored in compressed sparse column format. The diagonal matrix $D_{\mathcal{C}}$ is stored as an array of diagonal entries. Note that $(-\bar{d}_{22})^{1/2}$ is always real because we run algorithms 2 and 3 only for row/column combinations that are in the lower and right parts of $K_{\mathcal{C}}^{\epsilon}$ (where $A_{\mathcal{C}}$ changes), for which the diagonal entries of $D_{\mathcal{C}}$ are all negative by quasidefiniteness of $K_{\mathcal{C}}^{\epsilon}$.

It is important to note that all steps, including step 5 in both algorithms, are of at most quadratic complexity, whereas a full re-factorization would be of cubic complexity. When only a few constraints switch their activity, this procedure is considerably faster than full re-factorizations. This is demonstrated in §4 for three practical cases. In the worst case where all constraints switch to active or inactive between two consecutive solves, the complexity returns to cubic.

Open source code and full documentation for CVXPYgen and its differentiation feature is available at

`https://pypi.org/project/cvxpygen`.

3 System tuning framework

We present a generic tuning method for systems of the form

$$p = \Gamma(\omega)$$

where $p \in \mathbf{R}$ is a performance objective, Γ evaluates the system, which includes many solves of the convex optimization problem (1), possibly sequentially, and $\omega \in \Omega \subseteq \mathbf{R}^p$ is a *design*, where Ω is the design space, *i.e.*, the set of admissible designs. Then, we describe in detail what Γ and ω are, for two important classes of system tuning.

3.1 A generic tuning method

We compute the gradient $\nabla\Gamma(\omega)$ using the chain rule and our ability to differentiate through DPP problems. We use $\nabla\Gamma(\omega)$ in a simple projected gradient method [Ber97, CM87] to optimize the design ω.

We use Euclidean projections onto the design space Ω, denoted by $\Pi(\cdot)$, and a simple line search to guarantee that the algorithm is a descent method. If the performance is improved with the current step size, we use it for the current iteration and increase it by a constant factor $\beta > 1$ for the next iteration. Otherwise, we repeatedly shrink the step size by a constant factor $\eta > 1$ until the performance is improved. The simple generic design method we use is given in algorithm 4.

Algorithm 4 Projected gradient descent

```
1: Initialize $\omega^0$, $\alpha^0$, $k = 0$
2: repeat
3:     $\hat\omega = \Pi(\omega^k - \alpha^k \nabla\Gamma(\omega^k))$                ▷ tentative update
4:     if $\Gamma(\hat\omega) < \Gamma(\omega^k)$ then
5:         $\omega^{k+1} = \hat\omega$, $\alpha^{k+1} = \beta\alpha^k$        ▷ accept update and increase step size
6:     else
7:         $\alpha^k \leftarrow \alpha^k/\eta$, go to step 3             ▷ shrink step size and re-evaluate
8:     end if
9:     $k \leftarrow k + 1$
10: until $\|\omega^k - \Pi(\omega^k - \alpha^k \nabla\Gamma(\omega^k))\|_2 \leq \epsilon^{\rm rel}\|\omega^k\|_2 + \epsilon^{\rm abs}$
```

Note that algorithm 4 assumes that $\Gamma(\omega)$ is to be minimized. If $\Gamma(\omega)$ is to be maximized, replace it with $-\Gamma(\omega)$.

Initialization. We initialize ω^0 to a value that is typical for the respective application and α^0 with the clipped Polyak step size

$$\alpha^0 = \min\left\{\frac{\Gamma(\omega^0) - \hat{p}}{\|\nabla\Gamma(\omega^0)\|_2^2}, 1\right\},$$

where $\hat{p}$ is an estimate for the optimal value of the performance objective. We clip the step size at 1 to avoid too large initial steps due to local concavity. The algorithm is not particularly dependent on the line search parameters β and η. Reasonable choices are, *e.g.*, $\beta = 1.2$ and $\eta = 1.5$.

Stopping criterion. We stop the algorithm as soon as the termination criterion

$$\|\omega^k - \Pi(\omega^k - \alpha^k \nabla\Gamma(\omega^k))\|_2 \leq \epsilon^{\text{rel}}\|\omega^k\|_2 + \epsilon^{\text{abs}}$$

with $\epsilon^{\text{rel}}, \epsilon^{\text{abs}} > 0$ is met. This is also referred to as the *projected gradient* being small. When Γ is convex, this corresponds to the first-order optimality condition, *i.e.*, the gradient $\nabla\Gamma(\omega^k)$ lying in (or close to) the normal cone to Ω at ω^k [BV04]. Depending on the application, the stopping tolerances ϵ^{rel} and ϵ^{abs} might range between 10^{-2} and 10^{-6}.

3.2 Tuning hyper-parameters of machine learning models

We call the data points used for *training* a machine learning model $(z_1, y_1), \ldots, (z_N, y_N) \in \mathcal{D}$, where each data point consists of features z_i and output y_i. For the design ω of the machine learning model, we consider any hyper-parameters, including pre-processing parameters that determine how the data $(z_1, y_1), \ldots, (z_N, y_N)$ is modified before fitting the model.

For any choice of ω, we find the model weights $\beta \in \mathbf{R}^n$ as

$$\beta^\star(\omega) = \underset{\beta}{\text{argmin}}\ \frac{1}{N}\sum_{i=1}^{N} \ell(z_i, y_i, \beta, \omega) + r(\beta, \omega),$$

where $\ell : \mathcal{D} \times \mathbf{R}^n \times \Omega \to \mathbf{R}$ is the training loss function and $r : \mathbf{R}^n \times \Omega \to \mathbf{R}$ is the regularizer. While the entries of β are oftentimes referred to as *model parameters* in the machine learning literature, we call them *weights* to make clear that they enter the above optimization problem as variables (and not as parameters of the optimization problem). Both ℓ and r are parametrized by the design ω. In the case of ℓ, the design ω might enter in terms of pre-processing parameters like thresholds for processing outliers. In the case of r, the design ω might enter as the scaling of the regularization term. In the remainder of this work, we consider ℓ and r that are convex and quadratic, admitting the differentiation method described in §2.

We choose ω to minimize the validation loss

$$p = \Gamma(\omega) = \frac{1}{N^{\text{val}}}\sum_{i=1}^{N^{\text{val}}} \ell^{\text{val}}(z_i^{\text{val}}, y_i^{\text{val}}, \beta^\star(\omega), \omega),$$

where $\ell^{\text{val}} : \mathcal{D} \times \mathbf{R}^n \times \Omega \to \mathbf{R}$ is the validation loss function, with validation data $(z_i^{\text{val}}, y_i^{\text{val}})$ that is different from and ideally uncorrelated with the training data. Here, ℓ^{val} need not be convex (or quadratic), since we use the projected gradient method described in §3.1. Note that the design ω enters ℓ^{val} both directly (for example as pre-processing parameters) and through the optimal model parameters $\beta^\star(\omega)$.

If ℓ^{val} is the squared error (between data and model output), then the alternative performance objective

$$\bar{p} = \bar{\Gamma}(\omega) = \left(\frac{1}{N^{\text{val}}} \sum_{i=1}^{N^{\text{val}}} \ell^{\text{val}}(z_i^{\text{val}}, y_i^{\text{val}}, \beta^\star(\omega), \omega) \right)^{1/2},$$

is more meaningful, as it resembles the root mean square error (RMSE).

Cross validation. For better generalization of the optimized ω to unseen data, we can employ cross validation (CV) [Sha93, HTF09]. We split the set of data points into J partitions or *folds* (typically equally sized) and train the model J times. Every time, we take $J-1$ folds as training data and 1 fold as validation data. We compute the performance p_j as described above, where the subscript j denotes that the validation data $(z_i^{\text{val}}, y_i^{\text{val}})$ is that of the jth fold. Then, we average these over all folds as

$$p^{\text{CV}} = \frac{1}{J} \sum_{j=1}^{J} p_j(\omega).$$

This is usually referred to as the cross validation loss. Similarly, if ℓ^{val} is the squared error, we compute the cross-validated RMSE $\bar{p}^{\text{CV}}$ by averaging all $\bar{p}_j$.

3.3 Tuning the weights of convex optimization control policies

We consider a convex optimization control policy (COCP) that determines a control input $u \in \mathcal{U} \subseteq \mathbf{R}^m$ that is applied to a dynamical system with state $x \in \mathcal{X} \subseteq \mathbf{R}^n$, by solving the convex optimization problem

$$u = \phi(x; \omega) = \underset{u \in \mathcal{U}}{\operatorname{argmin}}\, \ell(x, u, \omega).$$

Here, $\phi : \mathcal{X} \times \Omega \to \mathcal{U}$ is a family of control policies, parametrized by ω, and x is the current measurement (or estimate) of the state of the dynamical system. The loss function $\ell : \mathcal{X} \times \mathcal{U} \times \Omega \to \mathbf{R}$ is parametrized by the design ω, which might involve controller weights, for example.

We choose ω to minimize the closed-loop loss

$$p = \Gamma(\omega) = \ell^{\text{cl}}(x_0, \omega),$$

where the loss function $\ell^{\text{cl}} : \mathcal{X} \times \Omega \to \mathbf{R}$ involves a simulation or real experiment starting from the initial state x_0 and using the control policy $u = \phi(x; \omega)$.

4 Numerical experiments

In this section we present three numerical examples, comparing the solve and differentiation speed of CVXPYgen with CVXPYlayers, for system tuning with the framework presented in §3.1. In all three cases we take OSQP as the canonical solver for CVXPYgen and Clarabel as the canonical solver for CVXPYlayers (since it only supports conic solvers). The code that was used for the experiments is available at

https://github.com/cvxgrp/cvxpygen.

4.1 Elastic net regression with winsorized features

We consider a linear regression model with elastic net regularization [ZH05], which is a sum of ridge (sum squares) [HK70] and lasso (sum absolute) [Tib96] regularization, each of which has a scaling parameter. In addition, we clip or *winsorize* each feature at some specified level to mitigate the problem of feature outliers. The clipping levels for each feature are also parameters.

We consider m data points with one observation and n features each. We start with a set of observations $y_i \in \mathbf{R}$ and raw features $z_i \in \mathbf{R}^n$, where $z_{i,j}$ is the jth feature for the ith observation, subject to outliers. We obtain the winsorized features $x_i \in \mathbf{R}^n$ by clipping the n components at winsorization levels $w \in \mathbf{R}^n_{++}$ (part of the design) as

$$x_{i,j}(z_{i,j}; w) = \min\{\max\{z_{i,j}, -w_j\}, w_j\}, \quad i = 1, \ldots, m, \quad j = 1, \ldots, n. \tag{4}$$

The training loss function and regularizer as in §3.2 are

$$\ell(z_i, y_i, \beta, \omega) = (x_i(z_i; w)^T \beta - y_i)^2, \quad r(\beta, \omega) = \lambda \|\beta\|_2^2 + \gamma \|\beta\|_1,$$

where $\lambda \geq 0$ and $\gamma \geq 0$ are the the ridge and lasso regularization factors, respectively. Since the model performance depends mainly on the orders of magnitude of λ and γ, we write them as $\lambda = 10^\mu$ and $\gamma = 10^\nu$.

Together with the winsorization levels w, the design vector becomes $\omega = (w, \mu, \nu) \in \mathbf{R}^n_{++} \times \mathbf{R}^2$. We allow to clip z_i between 1 and 3 standard deviations and search the elastic net weights across 7 orders of magnitude. We assume that the entries of z_i are approximately centered and scaled, *i.e.*, have zero mean and standard deviation 1. The design space becomes

$$\Omega = [1, 3]^n \times [-3, 3]^2.$$

The validation loss function ℓ^{val} is identical to the training loss function ℓ and the performance objective is the cross-validated RMSE $\bar{p}^{\text{CV}}$ from §3.2.

Code generation. Figure 2 shows how to generate code for this problem. The problem is modeled with CVXPY in lines 5–9. Code is generated with CVXPYgen in line 12, where we use the `gradient=True` option to generate code for computing gradients through the problem. After importing the CVXPYlayers interface in line 16, it is passed to the `Cvxpylayer` constructor in line 17 through the `custom_method` keyword. The performance objective is computed in line 18 and differentiated in line 19.

```python
import cvxpy as cp
from cvxpygen import cpg

# model problem
beta = cp.Variable(n, name='beta')
X = cp.Parameter((m-m//J, n), name='X')
l = cp.Parameter(nonneg=True, name='l')
g = cp.Parameter(nonneg=True, name='g')
prob = cp.Problem(cp.Minimize(cp.sum_squares(X @ beta - y)
    + l * cp.sum_squares(beta) + g * cp.norm(beta, 1)), [])

# generate code
cpg.generate_code(prob, gradient=True)

# use CVXPYlayers interface
from cvxpylayers.torch import Cvxpylayer
from cpg_code.cpg_solver import forward, backward
layer = Cvxpylayer(prob, parameters=[X,l,g], variables=[beta],
    custom_method=(forward, backward))
p = Gamma(w, mu, nu)  # involves beta_solution = layer(...)
p.backward()
print(w.grad, mu.grad, nu.grad)
```

Figure 2: Code generation and CVXPYlayers interface for elastic net example. The integers `m`, `n`, and `J`, the constant `y`, the function `Gamma`, and the `torch` tensors `w`, `mu`, and `nu` are pre-defined.

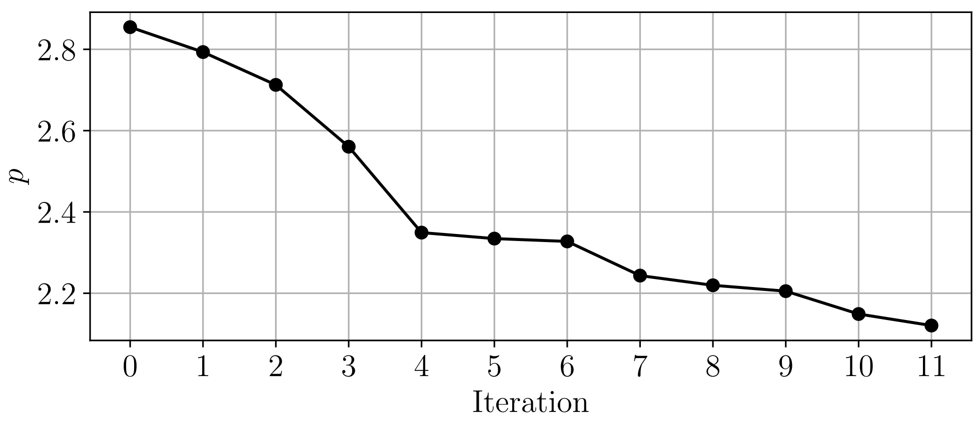


Figure 3: CV loss over tuning iterations.

Data generation. We take $m = 100$ data points, $n = 20$ features, and $J = 10$ CV folds. For every fold, we reserve $m/J = 10$ data points for validation and use the other 90 data points for training. We generate the features $\bar{z}_i$ without outliers by sampling from the Gaussian $\mathcal{N}(0, 1)$. Then, we sample $\bar{\beta} \sim \mathcal{N}(0, I)$ and set noisy labels $y_i = \bar{z}_i^T \bar{\beta} + \xi_i$ with $\xi_i \sim \mathcal{N}(0, 0.01)$. Afterwards, we simulate feature outliers due to, *e.g.*, data capturing errors. For every feature, we randomly select $m/10 = 10$ indices and increase the magnitude of the respective entries of $\bar{z}_i$ to a value $\sim \mathcal{U}[2, 4]$, *i.e.*, between 2 and 4 standard deviations, and save them in z_i. We estimate the optimal cross-validated RMSE as $\hat{p} = 0.1$ corresponding to the standard deviation of the noise ξ (if it was known). This is a very optimistic estimate, since it implies that the data is outlier-free after winsorization. The tuning parameters are initialized to $\omega^0 = (w, \mu, \nu)^0 = (3 \cdot \mathbf{1}, 0, 0)$. We set the termination tolerances to $\epsilon^{\text{rel}} = \epsilon^{\text{abs}} = 10^{-3}$.

Results. The projected gradient method terminates after 11 steps with a reduction of the cross-validated RMSE from 2.85 to 2.12, as shown in figure 3. Figure 4 shows the tuned winsorization thresholds w and we obtain $\lambda \approx 0.68$ and $\gamma \approx 0.80$.

Timing. Table 1 shows that the speed-up factor for the gradient computations is about 5. The speed-up for the full tuning loop is reduced due to Python overhead.

	Full tuning	Solve and Gradient	Gradient
CVXPYlayers	1.684 sec	1.570 sec	0.033 sec
CVXPYgen	0.696 sec	0.490 sec	0.007 sec

Table 1: Computation times with CVXPY and CVXPYgen for the elastic net example.

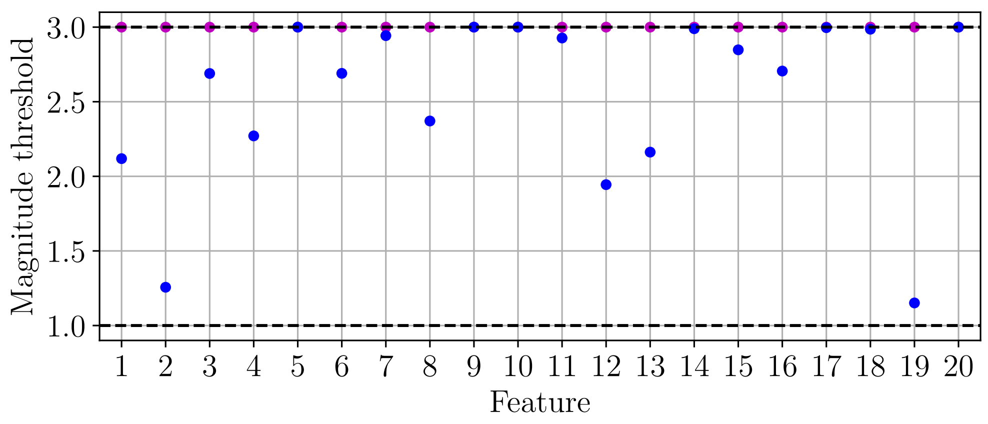


Figure 4: Thresholds before (magenta) and after tuning (blue). The dashed black lines show the range of possible winsorization thresholds.

4.2 Approximate dynamic programming controller

We investigate controller design with an ADP controller [WOB15, KB14] where the state and input cost parameters are subject to tuning.

We are given the discrete-time dynamical system

$$x_{t+1} = Ax_t + Bu_t + w_t, \quad t = 0, 1, \ldots, \tag{5}$$

with state $x_t \in \mathbf{R}^n$, input $u \in \mathbf{R}^m$ limited as $\|u\|_\infty \leq 1$, and state disturbance w_t, where w_t are unknown, but assumed IID $\mathcal{N}(0, W)$, with W known. The matrices $A \in \mathbf{R}^{n\times n}$ and $B \in \mathbf{R}^{n\times m}$ are the given state transition and input matrices, respectively.

We seek a state feedback controller $u_t = \phi(x_t)$ that guides the state x_t to zero while respecting the constraint $\|u\|_\infty \leq 1$. We judge a controller ϕ by the metric

$$J = \lim_{T\to\infty} \frac{1}{T} \mathbf{E} \sum_{t=0}^{T-1} \left(x_t^T Q x_t + u_t^T R u_t\right),$$

where $Q \in \mathbf{S}^n_+$ and $R \in \mathbf{S}^m_{++}$ are given. We assume that the matrix A contains no unstable eigenvalues with magnitude beyond 1, such that J is guaranteed to exist. Corresponding to §3.3, we will take our performance metric as

$$p = \Gamma(\omega) = \ell^{\mathrm{cl}}(x_0, \omega) = \sum_{t=0}^{T-1} \left(x_t^T Q x_t + u_t^T R u_t\right),$$

where T is large and fixed, and w_t are sampled from $\mathcal{N}(0, W)$. Note that all $x_1, \ldots, x_{T-1}$ and $u_0, \ldots, u_{T-1}$ are fully determined by the initial state x_0, the controller $\phi(x; \omega)$, and the system dynamics (5).

When the input constraint is absent, we can find the optimal controller (*i.e.*, the one that minimizes J) using dynamic programming, by minimizing a convex quadratic function,

$$(Ax_t + Bu)^T P^{\mathrm{lqr}} (Ax_t + Bu) + u^T R u,$$

where the matrix $P^{\text{lqr}} \in \mathbf{S}^n_{++}$ is the solution of the algebraic Riccati equation (ARE) for discrete time systems. The minimizer is readily obtained analytically, with u a linear function of the state x_t. See, *e.g.*, [KS72, PLS80, AM07].

We will use an approximate dynamic programming (ADP) controller

$$\phi(x_t;\omega) = \underset{u\in\mathcal{U}}{\operatorname{argmin}}\, \ell(x_t,u,\omega) = \underset{\|u\|_\infty\le 1}{\operatorname{argmin}} (Ax_t+Bu)^T(P^{\text{lqr}}+Z)(Ax_t+Bu) + u^T R u.$$

The controller is designed by $\omega = Z$ with $\Omega = \mathbf{S}^n_+$. The state x_t is another parameter and the matrices A, B, P^{lqr}, and R are constants.

The quadratic form in the objective makes the above formulation non-DPP. We render the problem DPP as

$$\phi(x_t;\omega) = \underset{\|u\|_\infty\le 1}{\operatorname{argmin}} \|g+Hu\|_2^2 + u^T R u,$$

where the DPP parameter θ consists of $g = L^T A x_t$ and $H = L^T B$. Here, $L = \mathbf{Chol}(P^{\text{lqr}}+Z)$, where $\mathbf{Chol}(\cdot)$ returns the lower Cholesky factor of its argument. In other words, $LL^T = P^{\text{lqr}}+Z$ and L is lower triangular. (When running our projected gradient descent algorithm, we modify L instead of Z and recover $Z = LL^T - P^{\text{lqr}}$ at the end of the tuning).

Data generation. We choose $n = 6$ states and $m = 3$ inputs. We consider an open-loop system $A = \mathbf{diag}\, a$ with stable and unstable modes sampled from $[0.99, 1.00]$. The entries of the input matrix B are sampled from $[-0.01, 0.01]$. We initialize the state at the origin and simulate for $T = 1000$ steps with noise covariance $W = 0.1^2 I$. The *true* state and input cost matrices are $Q = R = I$, respectively, which we use to compute P^{lqr} as the solution to the ARE. We initialize $\omega^0 = Z^0 = 0$ and estimate the optimal control performance $\hat{p}$ by running the simulation with the input constraint of the controller removed. We use $\epsilon^{\text{rel}} = \epsilon^{\text{abs}} = 0.005$.

Results. The projected gradient descent algorithm terminates after 16 gradient steps with a reduction of the control objective from 8.23 to 6.32, as shown in figure 5.

Timing. The gradient computations are sped up compared to CVXPY by a factor of about 40. Including Python overhead, the whole tuning loop is still sped up by a factor of about 6, as shown in table 2.

	Full tuning	Solve and gradient	Gradient
CVXPYlayers	107.2 sec	101.9 sec	15.7 sec
CVXPYgen	18.3 sec	10.5 sec	0.4 sec

Table 2: Computation times with CVXPY and CVXPYgen for the ADP controller tuning example.

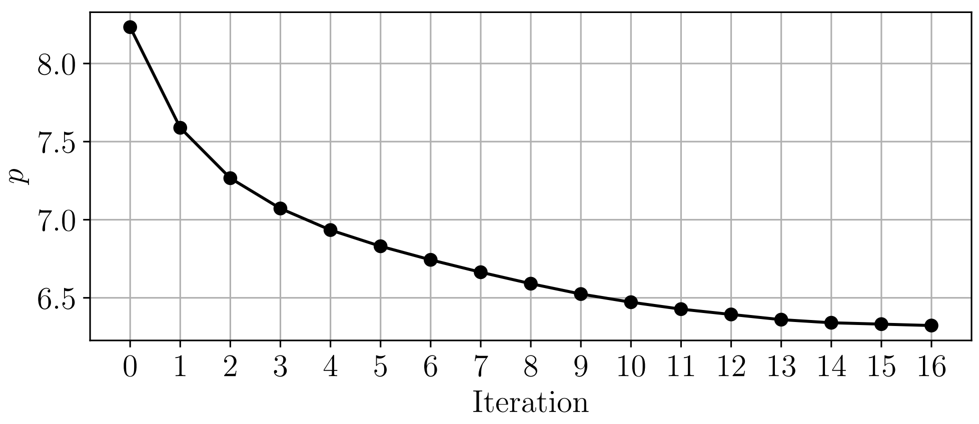


Figure 5: Control performance over tuning iterations.

4.3 Portfolio optimization

We consider a variant of the classical Markowitz portfolio optimization model [Mar52] with holding cost for short positions, transaction cost, and a leverage limit [BJK+24, LFB07], embedded in a multi-period trading system [BBD+17].

We want to find a fully invested portfolio of holdings in N assets. The holdings are represented relative to the total portfolio value, in terms of weights $w \in \mathbf{R}^N$ with $\mathbf{1}^T w = 1$. The expected portfolio return is $\mu^T w$, with esimated returns $\mu \in \mathbf{R}^N$. The variance or risk of the portfolio return is $w^T \Sigma w$, with estimated asset return covariance $\Sigma \in \mathbf{S}^N_{++}$. We assume that μ and Σ are pre-computed, which we will detail later. We approximate the cost of holding short positions as $\kappa^{\text{hold}} \mathbf{1}^T w_-$, where κ^{hold} describes the (equal) cost of holding a short position in any asset. Subscript "$-$" denotes the negative part, *i.e.*, $w_- = \max\{-w, 0\}$. We approximate the transaction cost as $\kappa^{\text{tc}} \|w - w^{\text{pre}}\|_1$, where κ^{tc} describes the cost of trading any asset and w^{pre} is the pre-trade portfolio. We solve

$$\begin{array}{ll} \text{maximize} & \mu^T w - \gamma^{\text{risk}} w^T \Sigma w - \gamma^{\text{hold}} \kappa^{\text{hold}} \mathbf{1}^T w_- - \gamma^{\text{tc}} \kappa^{\text{tc}} \|\Delta w\|_1 \\ \text{subject to} & \mathbf{1}^T w = 1, \quad \|w\|_1 \leq L, \quad \Delta w = w - w^{\text{pre}}, \end{array} \tag{6}$$

where $w, \Delta w \in \mathbf{R}^N$ are the variables. We introduced the variable Δw, also referred to as the *trade vector*, to prevent products of parameters and render the problem DPP. Problem (6) can also be seen as a convex optimization control policy as described in §3.3, where the expected returns μ (updated once per trading period) and the previous portfolio w^{pre} are the state x and the trade Δw is the input u. Our design consists of the leverage limit L and the aversion factors $\gamma^{\text{risk}}, \gamma^{\text{hold}}, \gamma^{\text{tc}} > 0$ for risk, holding cost, and short-selling cost, respectively. Since the model performance depends primarily on the orders of magnitude of these factors, we write them as

$$\gamma^{\text{risk}} = 10^{\nu^{\text{risk}}}, \quad \gamma^{\text{hold}} = 10^{\nu^{\text{hold}}}, \quad \gamma^{\text{tc}} = 10^{\nu^{\text{tc}}},$$

and tune $\omega = (L, \nu^{\text{risk}}, \nu^{\text{hold}}, \nu^{\text{tc}})$, restricted to the design space

$$\Omega = [1, 2] \times [-3, 3]^3.$$

We keep the risk Σ and costs κ^{hold} and κ^{tc} constant.

We evaluate the performance of the model via a back-test over h trading periods. After solving problem (6) at a given period, we trade to $w^\star$, pay short-selling cost $\kappa^{\text{hold}}\mathbf{1}^T w_-^\star$ and transaction cost $\kappa^{\text{tc}}\|w^\star - w^{\text{pre}}\|_1$, experience the returns r_t, and re-invest the full portfolio value. Hence, the total portfolio value evolves as

$$V_{t+1} = V_t(1 + r_t^T w^\star) - \kappa^{\text{hold}}\mathbf{1}^T w_-^\star - \kappa^{\text{tc}}\|w^\star - w^{\text{pre}}\|_1.$$

The pre-trade portfolio for the following trading period is re-balanced as

$$w^{\text{pre}} = w^\star \circ (1 + r_t) \cdot V_t / V_{t+1}$$

and the portfolio realized return at period t is

$$R_t = (V_{t+1} - V_t)/V_t.$$

We consider the average return and portfolio risk,

$$\bar{R} = (1/h)\sum_{t=1}^{h} R_t, \quad \sigma = \left((1/h)\sum_{t=1}^{h} R_t^2\right)^{1/2},$$

respectively, and annualize them as

$$\bar{R}^{\text{ann}} = h^{\text{ann}}\bar{R}, \quad \sigma^{\text{ann}} = (h^{\text{ann}})^{1/2}\sigma,$$

where h^{ann} is the number of trading periods per year. We take their ratio as performance metric, the so-called *Sharpe ratio (SR)* [LW08],

$$p = \text{SR} = \bar{R}^{\text{ann}}/\sigma^{\text{ann}} = (h^{\text{ann}})^{1/2}\bar{R}/\sigma.$$

Data generation. We consider three adjacent intervals of trading periods. First, we use a *burn-in* interval to compute the estimate for the expected returns at later time periods and to compute the constant risk estimate. Second we take a *tune* interval to perform parameter optimization. Third, we use a *test* interval to evaluate the final parameter choice out-of-sample. We denote the lengths of the three intervals by h^{burnin}, h^{tune}, and h^{test}, respectively.

We compute the expected return μ_t for the holdings at time t as the back-looking moving average of historical returns r_t with window size h^{burnin}. To compute the constant risk estimate, we first compute the empirical covariance $\hat{\Sigma}$ of returns over the burn-in interval. Then, we fit the standard factor model

$$\Sigma = FF^T + D$$

to $\hat{\Sigma}$, where $F \in \mathbf{R}^{N\times K}$ is the factor loading matrix and the diagonal matrix $D \in \mathbf{S}_{++}^N$ stores the variance of the idiosyncratic returns [EGBG09].

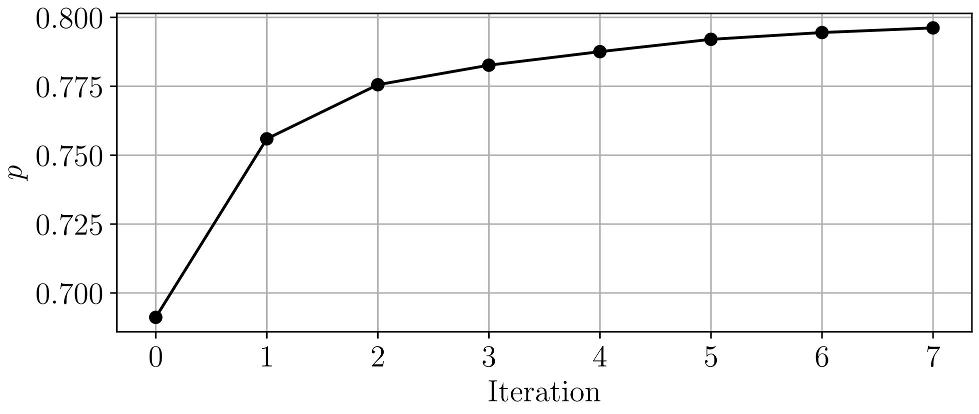


Figure 6: Sharpe ratio over tuning iterations.

We consider $N = 25$ stock assets, chosen randomly from the S&P 500, where historical return data is available from 2016–2019. While this clearly imposes survivership bias [BGIR92], the point of this experiment is not to find realistic portfolios but rather to assess the numerical performance of CVXPYgen. We choose $K = 5$ factors, $h^{\text{burnin}} = 260$, $h^{\text{tune}} = 520$, $h^{\text{test}} = 260$, and we take the number of trading periods per year as $h^{\text{ann}} = 260$, *i.e.*, we trade once a day. In other words, we use data from the year 2016 as burn-in interval for estimating μ_t and to estimate Σ. We tune the model with data from the years 2017 and 2018 and test it with data from the year 2019. We fix $\kappa^{\text{hold}} = \kappa^{\text{tc}} = 0.001$. We estimate the optimal Sharpe ratio to be roughly $\hat{p} = 1$. We initialize the design vector as $\omega^0 = (L, \nu^{\text{risk}}, \nu^{\text{hold}}, \nu^{\text{tc}})^0 = (1, 0, 0, 0)$. We set the termination tolerances to $\epsilon^{\text{rel}} = \epsilon^{\text{abs}} = 0.03$.

Results. The projected gradient descent algorithm terminates after 7 iterations and improves the Sharpe ratio by a bit more than 0.1, from 0.69 to 0.80, where it is saturating, as shown in figure 6.

The values of the tuning parameters are changed to $\gamma^{\text{risk}} \approx 10$, $\gamma^{\text{hold}} \approx 1$, $\gamma^{\text{tc}} \approx 1.2$, and $L \approx 1$. Figure 7 contains the portfolio value over the trading periods used for tuning and out-of-sample, before and after tuning, respectively.

Table 3 contains the respective Sharpe ratios. While the tuning interval appears to be a difficult time period with large drawdown in the middle and the end of the interval, the Sharpe ratio is improved out-of-sample from an already high level.

	In-sample	Out-of-sample
Before tuning	0.69	2.21
After tuning	0.80	2.39

Table 3: Sharpe ratios.

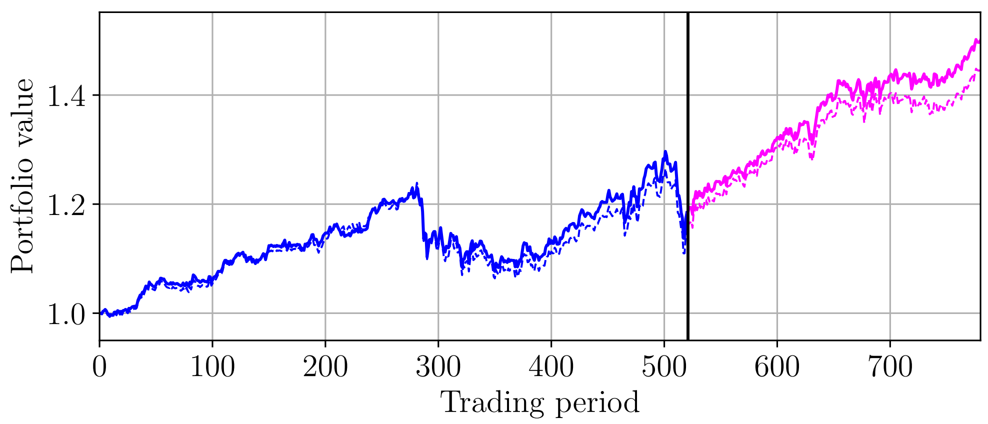


Figure 7: Portfolio value evolution before (dashed line) and after tuning (solid line). Blue and pink color represent the tuning and testing intervals, respectively.

Timing. Table 4 shows the solve and differentiation times. The gradient computations are sped up by a factor of about 10. Including Python overhead, the overall tuning loop is sped up by a factor of about 3.

	Full tuning	Solve and gradient	Gradient
CVXPYlayers	61 sec	57 sec	23 sec
CVXPYgen	21 sec	17 sec	2 sec

Table 4: Computation times with CVXPY and CVXPYgen for the portfolio optimization example.

5 Conclusions

We have added new functionality to the code generator CVXPYgen for differentiating through parametrized convex optimization problems. Users can model their problem in CVXPY with instructions close to the math, and create an efficient implementation of the gradient computation in C, by simply setting an additional keyword argument of the CVXPYgen code generation method. Our numerical experiments show that the gradient computations are sped up by around one order of magnitude for typical use cases.

References

[AAB+19] A. Agrawal, B. Amos, S. Barratt, S. Boyd, S. Diamond, and Z. Kolter. Differentiable convex optimization layers. In *Advances in Neural Information Processing Systems (NeurIPS)*, volume 32, 2019.

[ABB+19] A. Agrawal, S. Barratt, S. Boyd, E. Busseti, and W. Moursi. Differentiating through a cone program. *arXiv preprint arXiv:1904.09043*, 2019.

[ABBS20] A. Agrawal, S. Barratt, S. Boyd, and B. Stellato. Learning convex optimization control policies. In *Learning for Dynamics and Control*, pages 361–373. PMLR, 2020.

[ABC$^+$16] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, et al. TensorFlow: a system for large-scale machine learning. In *12th USENIX symposium on operating systems design and implementation (OSDI 16)*, pages 265–283, 2016.

[AK17] B. Amos and Z. Kolter. OptNet: Differentiable optimization as a layer in neural networks. In *International Conference on Machine Learning*, pages 136–145. PMLR, 2017.

[AM07] B. Anderson and J. Moore. *Optimal control: linear quadratic methods.* Courier Corporation, 2007.

[ASY$^+$19] T. Akiba, S. Sano, T. Yanase, T. Ohta, and M. Koyama. Optuna: A next-generation hyperparameter optimization framework. In *Proceedings of the 25th ACM SIGKDD international conference on knowledge discovery & data mining*, pages 2623–2631, 2019.

[BB91] S. Boyd and C. Barratt. *Linear controller design: limits of performance.* Citeseer, 1991.

[BB21] S. Barratt and S. Boyd. Least squares auto-tuning. *Engineering Optimization*, 53(5):789–810, 2021.

[BBD$^+$17] S. Boyd, E. Busseti, S. Diamond, R. Kahn, K. Koh, P. Nystrup, and J. Speth. Multi-period trading via convex optimization. *Foundations and Trends in Optimization*, 3(1):1–76, 2017.

[BEGFB94] S. Boyd, L. El Ghaoui, E. Feron, and V. Balakrishnan. *Linear matrix inequalities in system and control theory.* SIAM, 1994.

[Ber91] D. Bertsekas. *Linear network optimization: algorithms and codes.* MIT press, 1991.

[Ber97] P. Bertsekas. Nonlinear programming. *Journal of the Operational Research Society*, 48(3):334–334, 1997.

[BFH$^+$18] J. Bradbury, R. Frostig, P. Hawkins, M. Johnson, C. Leary, D. Maclaurin, G. Necula, A. Paszke, J. VanderPlas, S. Wanderman-Milne, and Q. Zhang. JAX: composable transformations of Python+NumPy programs, 2018.

[BG92] D. Bertsekas and R. Gallager. *Data networks.* Athena Scientific, 1992.

[BGIR92] S. Brown, W. Goetzmann, R. Ibbotson, and S. Ross. Survivorship bias in performance studies. *The Review of Financial Studies*, 5(4):553–580, 1992.

[BJK$^+$24] S. Boyd, K. Johansson, R. Kahn, P. Schiele, and T. Schmelzer. Markowitz portfolio construction at seventy. *arXiv preprint arXiv:2401.05080*, 2024.

[BN06] C. Bishop and N. Nasrabadi. *Pattern recognition and machine learning.* Springer, 2006.

[BT04] D. Bertsimas and A. Thiele. A robust optimization approach to supply chain management. In *Integer Programming and Combinatorial Optimization: 10th International IPCO Conference, New York, NY, USA, June 7-11, 2004. Proceedings 10*, pages 86–100. Springer, 2004.

[BV04] S. Boyd and L. Vandenberghe. *Convex Optimization.* Cambridge University Press, 2004.

[CH18] E. Carson and N. Higham. Accelerating the solution of linear systems by iterative refinement in three precisions. *SIAM Journal on Scientific Computing*, 40(2):A817–A847, 2018.

[CM87] P. Calamai and J. Moré. Projected gradient methods for linearly constrained problems. *Mathematical programming*, 39(1):93–116, 1987.

[Cor95] C. Cortes. Support-vector networks. *Machine Learning*, 20(3):273–297, 1995.

[Cox58] D. Cox. The regression analysis of binary sequences. *Journal of the Royal Statistical Society Series B: Statistical Methodology*, 20(2):215–232, 1958.

[DB16] S. Diamond and S. Boyd. CVXPY: A Python-embedded modeling language for convex optimization. *Journal of Machine Learning Research*, 17(83):1–5, 2016.

[DH99] T. Davis and W. Hager. Modifying a sparse Cholesky factorization. *SIAM Journal on Matrix Analysis and Applications*, 20(3):606–627, 1999.

[DH05] T. Davis and W. Hager. Row modifications of a sparse Cholesky factorization. *SIAM Journal on Matrix Analysis and Applications*, 26(3):621–639, 2005.

[DHL17] I. Dunning, J. Huchette, and M. Lubin. JuMP: a modeling language for mathematical optimization. *SIAM Review*, 59(2):295–320, 2017.

[EG22] A. Elmachtoub and P. Grigas. Smart “predict, then optimize”. *Management Science*, 68(1):9–26, 2022.

[EGBG09] E. Elton, M. Gruber, S. Brown, and W. Goetzmann. *Modern portfolio theory and investment analysis.* John Wiley & Sons, 2009.

[FNB20] A. Fu, B. Narasimhan, and S. Boyd. CVXR: an R package for disciplined convex optimization. *Journal of Statistical Software*, 94(14):1–34, 2020.

[GB14] M. Grant and S. Boyd. CVX: Matlab software for disciplined convex programming, version 2.1, 2014.

[GK00] R. Grinold and R. Kahn. *Active portfolio management.* McGraw Hill New York, 2000.

[GPM89] C. Garcia, D. Prett, and M. Morari. Model predictive control: theory and practice – a survey. *Automatica*, 25(3):335–348, 1989.

[Hal19] H. Halabian. Distributed resource allocation optimization in 5G virtualized networks. *IEEE Journal on Selected Areas in Communications*, 37(3):627–642, 2019.

[Hig97] N. Higham. Iterative refinement for linear systems and LAPACK. *IMA Journal of Numerical Analysis*, 17(4):495–509, 1997.

[HK70] A. Hoerl and R. Kennard. Ridge regression: Biased estimation for nonorthogonal problems. *Technometrics*, 12(1):55–67, 1970.

[Hol06] G. Holzmann. The power of 10: Rules for developing safety-critical code. *Computer*, 39(6):95–99, 2006.

[HTF09] T. Hastie, R. Tibshirani, and J. Friedman. *The elements of statistical learning: data mining, inference, and prediction.* Springer, 2009.

[KB14] A. Keshavarz and S. Boyd. Quadratic approximate dynamic programming for input-affine systems. *International Journal of Robust and Nonlinear Control*, 24(3):432–449, 2014.

[KC16] B. Kouvaritakis and M. Cannon. *Model predictive control.* Springer, 2016.

[KDVC$^+$24] J. Kotary, V. Di Vito, J. Cristopher, P. Van Hentenryck, and F. Fioretto. Learning joint models of prediction and optimization. *arXiv preprint arXiv:2409.04898*, 2024.

[KS72] H. Kwakernaak and R. Sivan. *Linear optimal control systems.* Wiley-InterScience New York, 1972.

[LÖ4] J. Löfberg. YALMIP: a toolbox for modeling and optimization in Matlab. In *IEEE International Conference on Robotics and Automation (ICRA)*, pages 284–289. IEEE, 2004.

[LAHH16] W. Lefever, E. Aghezzaf, and K. Hadj-Hamou. A convex optimization approach for solving the single-vehicle cyclic inventory routing problem. *Computers & Operations Research*, 72:97–106, 2016.

[LFB07] M. Lobo, M. Fazel, and S. Boyd. Portfolio optimization with linear and fixed transaction costs. *Annals of Operations Research*, 152:341–365, 2007.

[LJD$^+$18] L. Li, K. Jamieson, G. DeSalvo, A. Rostamizadeh, and A. Talwalkar. Hyperband: A novel bandit-based approach to hyperparameter optimization. *Journal of Machine Learning Research*, 18(185):1–52, 2018.

[LW08] O. Ledoit and M. Wolf. Robust performance hypothesis testing with the Sharpe ratio. *Journal of Empirical Finance*, 15(5):850–859, 2008.

[Mar52] H. Markowitz. Portfolio selection. *Journal of Finance*, 7(1):77–91, 1952.

[MB10] J. Mattingley and S. Boyd. Real-time convex optimization in signal processing. *IEEE Signal Processing Magazine*, 27(3):50–61, 2010.

[MBK$^+$22] G. Maher, S. Boyd, M. Kochenderfer, C. Matache, D. Reuter, A. Ulitsky, S. Yukhymuk, and L. Kopman. A light-weight multi-objective asynchronous hyper-parameter optimizer. *arXiv preprint arXiv:2202.07735*, 2022.

[Mur12] K. Murphy. *Machine learning: a probabilistic perspective.* MIT press, 2012.

[Nar13] R. Narang. *Inside the black box: a simple guide to quantitative and high-frequency trading.* John Wiley & Sons, 2013.

[NCB23] P. Nobel, E. Candès, and S. Boyd. Tractable evaluation of Stein's unbiased risk estimate with convex regularizers. *IEEE Transactions on Signal Processing*, 71:4330–4341, 2023.

[NLC24] P. Nobel, D. LeJeune, and E. Candès. RandALO: Out-of-sample risk estimation in no time flat. *arXiv preprint arXiv:2409.09781*, 2024.

[Pal25] D. Palomar. *Portfolio Optimization.* Cambridge University Press, 2025.

[PGC$^+$17] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer. Automatic differentiation in PyTorch. In *NIPS*, volume 31, 2017.

[PLS80] T. Pappas, A. Laub, and N. Sandell. On the numerical solution of the discrete-time algebraic riccati equation. *IEEE Transactions on Automatic Control*, 25(4):631–641, 1980.

[RMD$^+$17] J. Rawlings, D. Mayne, M. Diehl, et al. *Model predictive control: theory, computation, and design.* Nob Hill Publishing Madison, WI, 2017.

[SBD$^+$22] M. Schaller, G. Banjac, S. Diamond, A. Agrawal, B. Stellato, and S. Boyd. Embedded code generation with CVXPY. *IEEE Control Systems Letters*, 6:2653–2658, 2022.

[SBG$^+$20] B. Stellato, G. Banjac, P. Goulart, A. Bemporad, and S. Boyd. OSQP: an operator splitting solver for quadratic programs. *Mathematical Programming Computation*, 12(4):637–672, 2020.

[Sha93] J. Shao. Linear model selection by cross-validation. *Journal of the American statistical Association*, 88(422):486–494, 1993.

[Tib96] R. Tibshirani. Regression shrinkage and selection via the lasso. *Journal of the Royal Statistical Society Series B: Statistical Methodology*, 58(1):267–288, 1996.

[TK24] B. Tang and E. Khalil. PyEPO: A PyTorch-based end-to-end predict-then-optimize library for linear and integer programming. *Mathematical Programming Computation*, 16(3):1–39, 2024.

[UMZ$^+$14] M. Udell, K. Mohan, D. Zeng, J. Hong, S. Diamond, and S. Boyd. Convex optimization in Julia. In *2014 first workshop for high performance technical computing in dynamic languages*, pages 18–28. IEEE, 2014.

[Van95] R. Vanderbei. Symmetric quasidefinite matrices. *SIAM Journal on Optimization*, 5(1):100–113, 1995.

[Wai05] K. Wainwright. *Fundamental methods of mathematical economics*. McGraw-Hill, 2005.

[WB09] Y. Wang and S. Boyd. Fast model predictive control using online optimization. *IEEE Transactions on Control Systems Technology*, 18(2):267–278, 2009.

[WOB15] Y. Wang, B. O'Donoghue, and S. Boyd. Approximate dynamic programming via iterated Bellman inequalities. *International Journal of Robust and Nonlinear Control*, 25(10):1472–1496, 2015.

[ZE10] M. Zibulevsky and M. Elad. L1-L2 optimization in signal and image processing. *IEEE Signal Processing Magazine*, 27(3):76–88, 2010.

[ZH05] H. Zou and T. Hastie. Regularization and variable selection via the elastic net. *Journal of the Royal Statistical Society Series B: Statistical Methodology*, 67(2):301–320, 2005.