Optimization and Engineering
https://doi.org/10.1007/s11081-025-10057-2

RESEARCH ARTICLE

Code generation for solving and differentiating through convex optimization problems

Maximilian Schaller[1] · **Stephen Boyd**[1]

Received: 19 April 2025 / Revised: 11 August 2025 / Accepted: 2 November 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.

Keywords Convex optimization · Differentiable optimization · Code generation

1 Introduction

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

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

✉ Maximilian Schaller
mschall@stanford.edu

[1] Department of Electrical Engineering, Stanford University, Stanford, USA

Published online: 15 December 2025

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 (Boyd and Vandenberghe 2004). 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 (Mattingley and Boyd 2010; Zibulevsky and Elad 2010), machine learning (Murphy 2012; Bishop and Nasrabadi 2006; Zou and Hastie 2005; Tibshirani 1996; Cortes 1995; Hoerl and Kennard 1970; Cox 1958), control systems (Rawlings et al. 2017; Kouvaritakis and Cannon 2016; Wang et al. 2015; Keshavarz and Boyd 2014; Wang and Boyd 2009; Boyd et al. 1994; Boyd and Barratt 1991; Garcia et al. 1989), quantitative finance (Palomar 2025; Boyd et al. 2024; Boyd et al. 2017; Narang 2013; Lobo et al. 2007; Grinold and Kahn 2000; Markowitz 1952), and operations research (Halabian 2019; Bertsekas and Gallager 1992; Bertsekas 1991; Lefever et al. 2016; Bertsimas and Thiele 2004).

1.1 Differentiating through convex optimization problems

In many applications, such as machine learning models with hyper-parameters (Murphy 2012; Tibshirani 1996) and decision models in control or finance with tuning parameters (Agrawal et al. 2020; Boyd et al. 2024), 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, and instead return a reasonable value that ensures that the gradient computation remains functionable and stable (often achieved by picking a subgradient based on simple rules (Paszke et al. 2017). In particular, when $x^\star$ is not unique, or when the mapping is not differentiable, we simply use some reasonable value for the (nonexistent) derivative (corresponding to a similar problem where $x^\star$ is unique and the mapping is differentiable, for more details see Sect. 2). It has been observed that simple gradient (or subgradient) based methods for optimizing parameters are tolerant of these approximations (Bradbury et al. 2018; Paszke et al. 2017; Abadi et al. 2016).

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 (Nobel et al. 2023; Nobel et al. 2024). 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 (Wainwright 2005).

1.2 Autodifferentiation framework

The solution map derivative is much more useful when it is part of an autodifferentiation system such as Bradbury et al. (2018), PyTorch (Paszke et al. 2017), or Tensorflow (Abadi et al. 2016). 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 (Murphy 2012). 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 (Boyd et al. 2017), 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 (Ledoit and Wolf 2008), 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 (Paszke et al. 2017) and Tensorflow (Abadi et al. 2016), as commonly used in backpropagation for machine learning, can *unroll* iterative optimization algorithms and differentiate through all the iterations sequentially. By doing that, they neglect the structure and optimality conditions of, particularly, convex optimization problems.

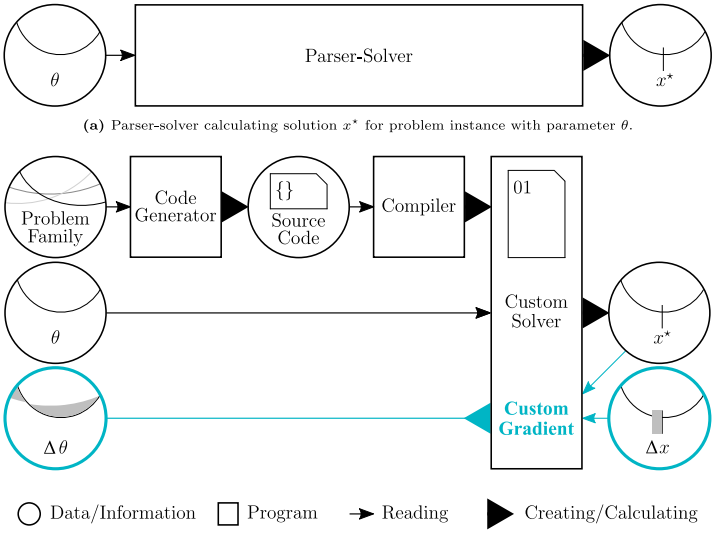
Second, to exploit the structure of a convex optimization problem, one can directly differentiate through its optimality conditions, also referred to as *argmin differenti-*

ation (Agrawal et al. 2019; Amos and Kolter 2017). CVXPYlayers (Agrawal et al. 2019) is a Python package based on prior work on differentiating through a cone program, called diffcp (Agrawal et al. 2019), and provides interfaces to PyTorch and Tensorflow. DiffOpt.jl (Besançon et al. 2023) provides similar functionality in Julia, for differentiating through linear, quadratic, and conic programs. OptNet (Amos and Kolter 2017) addresses quadratic programs and is integrated into PyTorch via the qpth package (Amos and Kolter 2017). Least squares auto-tuning (Barratt and Boyd 2021) 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. JAXopt (Blondel et al. 2022) provides argmin differentiation for various problem classes, including non-convex problems, yet appears to be not maintained any more. The latest version of acados (Frey et al. 2025) performs argmin differentiation for nonlinear programs (Frey et al. 2025). PyEPO (Tang and Khalil 2024) 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 (Tang and Khalil 2024; Elmachtoub and Grigas 2022). 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* (Kotary et al. 2024), 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 (Agrawal et al. 2020, 2019). Examples include Optuna (Akiba et al. 2019), HOLA (Maher et al. 2022), for tuning parameters with respect to some overall metric, and Hyperband (Li et al. 2018), 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 (Feurer and Hutter 2019, Sect. 1.6.2).

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 (Löfberg 2004) and CVX (Grant and Boyd 2014) (in Matlab), CVXPY (Diamond and Boyd 2016) (in Python), which CVXPYlayers is based on, Convex.jl (Udell et al. 2014) and JuMP (Dunning et al. 2017) (in Julia), and CVXR (Fu et al. 2020) (in R). We focus on CVXPY, which also supports the



(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.

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

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 (Boyd and Vandenberghe 2004). 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 the problem is solved (with different parameter values). This simple setting is illustrated in Fig. 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 (Schaller et al. 2022; Mattingley and Boyd 2012; Verschueren et al. 2021).

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 (Banjac et al. 2017; Chari and Açıkmeşe 2025). In this work, we add a program that efficiently computes the gradient of the parameter-solution mapping. The overall workflow is illustrated in Fig. 1b. The compiled solver and differentiator have a number of advantages over parser-solvers. First, by caching canonicalization and exploiting the problem structure (Banjac et al. 2017; Verschueren et al. 2021), 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 (Holzmann 2006).

1.4 Contribution

In this paper, we extend the code generator CVXPYgen (Schaller et al. 2022) 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 (Amos and Kolter 2017) with low-rank updates to the factorization of quasidefinite systems (Davis and Hager 1999, 2005) 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 Sect. 2 we describe, at a high level, how CVXPYgen generates code to differentiate through convex optimization problems. In Sect. 3 we describe a generic system tuning framework that requires differentiating through convex optimization problems and serves as a test bench for the methodology introduced before. In Sect. 4 we assess the performance of the CVXPYgen solvers and differentiators for three realistic system tuning examples, compared to CVXPYlayers. We conclude the paper in Sect. 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, CVXPYgen generates code to solve LPs, QPs, and SOCPs. We extend CVXPYgen to generate allocation-free code for 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 (Diamond and Boyd 2016). 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 (Agrawal et al. 2019). 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 Sect. 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 (Agrawal et al. 2019), *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

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

as used by the OSQP solver (Stellato et al. 2020). 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 (Amos and Kolter 2017). 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 (Carson and Higham 2018; Higham 1997) to correct for the effect of ϵ on the solution.

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

Algorithm 1 Regularized system solve

Initialize P, $A_{\mathcal{C}}$, $\Delta\tilde{x}$

$K_{\mathcal{C}} = \begin{bmatrix} P & A_{\mathcal{C}}^T \\ A_{\mathcal{C}} & 0 \end{bmatrix}$, $K_{\mathcal{C}}^{\epsilon} = K_{\mathcal{C}} + \begin{bmatrix} \epsilon I & 0 \\ 0 & -\epsilon I \end{bmatrix}$, $r = \begin{bmatrix} \Delta\tilde{x} \\ 0 \end{bmatrix}$

$z = (K_{\mathcal{C}}^{\epsilon})^{-1} r$

for $i = 1$ to N^{refine} **do**

  $\delta_r = r - K_{\mathcal{C}} z$

  $\delta_z = (K_{\mathcal{C}}^{\epsilon})^{-1} \delta_r$

  $z \leftarrow z + \delta_z$

end for

$(d_x, d_y) = z$

Ultimately, the gradients in the QP parameters are

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

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 (Vanderbei 1995),

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

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 run algorithm 3 for row/column deletion.

All steps of the row/column addition and deletion 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

Algorithm 2 Row/column addition (variant of algorithm 1 in Davis and Hager (2005))

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 $\bar{L}_{33} \bar{D}_{33} \bar{L}_{33}^T = L_{33} D_{33} L_{33}^T - w w^T$ according to algorithm 5 in Davis and Hager (1999)

Algorithm 3 Row/column deletion (variant of algorithm 2 in Davis and Hager (2005))

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 $\bar{L}_{33} \bar{D}_{33} \bar{L}_{33}^T = L_{33} D_{33} L_{33}^T + w w^T$ according to algorithm 5 in Davis and Hager (1999)

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 (solving a linear system with the coefficient matrix triangular and smaller than the full matrix). When only a few constraints switch their activity, algorithms 2 or 3 need to be executed only a few times, and this procedure is considerably faster than a full re-factorization, which would be of cubic complexity (solving the linear system from scratch). This is demonstrated in Sect. 4 for three practical cases. In the worst case where all constraints switch to active or inactive between two consecutive solves, the complexity approaches the cubic complexity of a full re-factorization, since there will be as many calls to algorithms 2 or 3 as there are rows in A.

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 introduce a generic system tuning framework, as a test bench for the methodology described above. We consider systems of the form

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

where $p \in \mathbf{R}$ is a performance objective and Γ evaluates the system, which includes many solves of the convex optimization problem (1), possibly sequentially. (This means that the function Γ is generally far from convex.) The input to the system is the *design* $\omega \in \Omega \subseteq \mathbf{R}^p$, where Ω is the design space, *i.e.*, the set of admissible

designs. We will 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 (Bertsekas 1997; Calamai and Moré 1987) 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 ω^0, α^0, $k = 0$
2: **repeat**
3: $\quad\hat{\omega} = \Pi(\omega^k - \alpha^k \nabla\Gamma(\omega^k))$ ▷ tentative update
4: $\quad$**if** $\Gamma(\hat{\omega}) < \Gamma(\omega^k)$ **then**
5: $\quad\quad\omega^{k+1} = \hat{\omega}$, $\alpha^{k+1} = \beta\alpha^k$ ▷ accept update and increase step size
6: $\quad$**else**
7: $\quad\quad\alpha^k \leftarrow \alpha^k/\eta$, go to step 3 ▷ shrink step size and re-evaluate
8: $\quad$**end if**
9: $\quad k \leftarrow k + 1$
10: **until** $\|\omega^k - \Pi(\omega^k - \alpha^k\nabla\Gamma(\omega^k))\|_2 \leq \epsilon^{\text{rel}}\|\omega^k\|_2 + \epsilon^{\text{abs}}$

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 (Boyd and Vandenberghe 2004). 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

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

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 Sect. 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 Sect. 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) (Shao 1993; Hastie et al. 2009). 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

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

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 CVXPYgen with CVXPYlayers for system tuning with the framework presented in Sect. 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). As the solutions and gradients obtained with CVXPYgen and CVXPYlayers are virtually the same, we just report the solve and gradient computation times in each case. We run the experiments on an Apple M1 Pro, compiling with Clang at optimization level 3, and report the times averaged over 10 trials each (with very little variability between trials). 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 (Zou and Hastie 2005), which is a sum of ridge (sum squares) (Hoerl and Kennard 1970) and lasso (sum absolute) (Tibshirani 1996) 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 Sect. 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 Sect. 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.

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

```
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)
```

Fig. 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

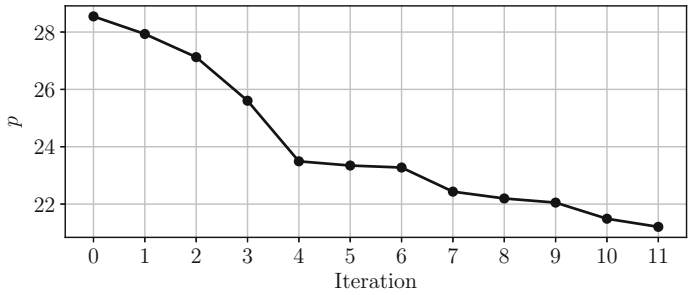


Fig. 3 CV loss over tuning iterations

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 Fig. 3, which also confirms the descent property of the projected gradient descent algorithm 4.

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, reflecting the benefit of generating custom, statically allocated C code for differentiating through the problem. The speed-up for the full tuning loop is reduced due to Python overhead. In particular, both with CVXPYlayers and CVXPYgen, the outer computations of the projected gradient descent algorithm (see Sect. 3.1, Algorithm 4)

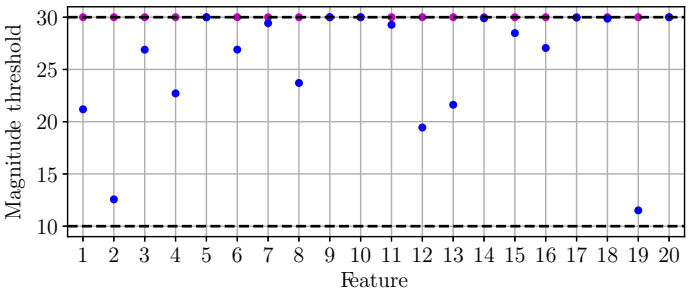


Fig. 4 Thresholds before (magenta) and after tuning (blue). The dashed black lines show the range of possible winsorization thresholds

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

	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

are performed in Python, without any speed-up. The same goes for differentiating through the winsorization step in PyTorch.

4.2 Approximate dynamic programming controller

We investigate controller design with an ADP controller (Wang et al. 2015; Keshavarz and Boyd 2014) 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 Sect. 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 $P^{\mathrm{lqr}} \in \mathbf{S}^n_{++}$ 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.*, (Kwakernaak and Sivan 1972; Pappas et al. 1980; Anderson and Moore 2007).

We will use an approximate dynamic programming (ADP) controller

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

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

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

where the DPP parameter θ consists of $g = L^T A x_t$ and $H = L^T B$. Here, $L = \mathbf{Chol}(P^{\mathrm{lqr}} + Z)$, where $\mathbf{Chol}(\cdot)$ returns the lower Cholesky factor of its argument. In other words, $LL^T = P^{\mathrm{lqr}} + Z$ and L is lower triangular. (When running our projected gradient descent algorithm, we modify L instead of Z and recover $Z = LL^T - P^{\mathrm{lqr}}$ 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 close to 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 $\epsilon^{\mathrm{rel}} = \epsilon^{\mathrm{abs}} = 0.005$.

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 Fig. 5.

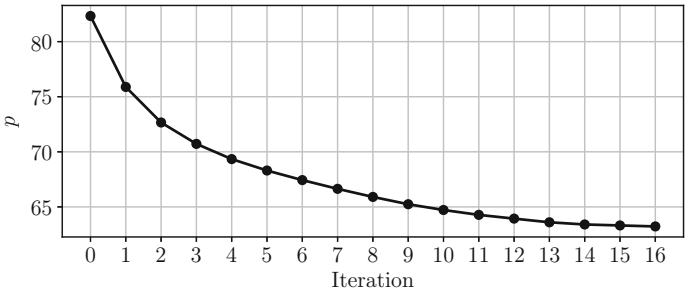


Fig. 5 Control performance over tuning iterations

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

	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

Timing. The gradient computations are sped up compared to CVXPY by a factor of about 40. We obtain a large speed-up, since only few constraints (in the canonical representation) switch between active and inactive from one solve to the next. Intuitively, due to the smooth nature of the dynamical system subject to control, the constraint $\|u\|_\infty \leq 1$ tends to be active for consecutive time steps. This means that there are only few times when the constraint switches activity and rank-1 factorization updates are triggered as described in Sect. 2.3. Including Python overhead, the whole tuning loop is still accelerated by a factor of about 6, as shown in table 2.

4.3 Portfolio optimization

We consider a variant of the classical Markowitz portfolio optimization model (Markowitz 1952) with holding cost for short positions, transaction cost, and a leverage limit (Boyd et al. 2024; Lobo et al. 2007), embedded in a multi-period trading system (Boyd et al. 2017).

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

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

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 Sect. 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

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

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)* (Ledoit and Wolf 2008),

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

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 (Elton et al. 2009).

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 (Brown et al. 1992), 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 Fig. 6.

The values of the tuning parameters are changed to $\gamma^{\text{risk}} \approx 11, \gamma^{\text{hold}} \approx 1, \gamma^{\text{tc}} \approx 1.2$, and $L \approx 1$. Fig. 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.

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.

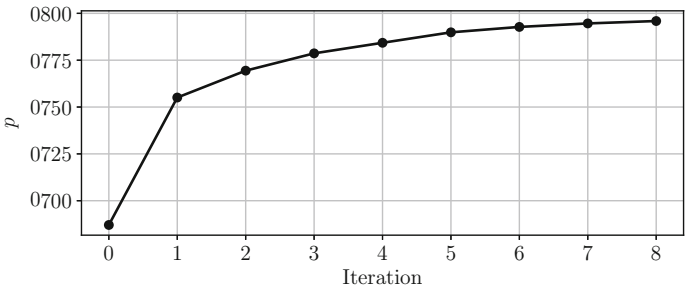


Fig. 6 Sharpe ratio over tuning iterations

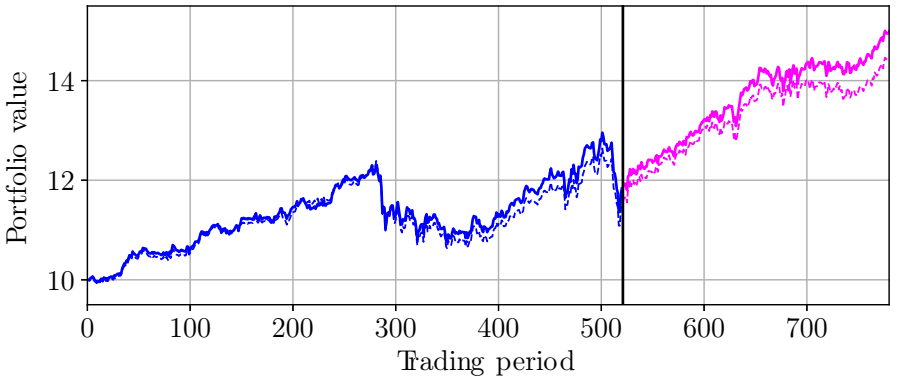


Fig. 7 Portfolio value evolution before (dashed line) and after tuning (solid line). Blue and pink color represent the tuning and testing intervals, respectively

Table 3 Sharpe ratios.

	In-sample	Out-of-sample
Before tuning	0.69	2.21
After tuning	0.80	2.38

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

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

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. Future work may include generating code for differentiating through second-order cone programs, in addition to linear and quadratic programs.

Acknowledgements The authors thank Daniel Cederberg and Kasper Johansson for their helpful suggestions.

Author Contributions Maximilian Schaller led the research project, developed the methodology, performed the analysis, and drafted the manuscript. Stephen Boyd provided supervision, conceptual guidance, and contributed significantly to revising and refining the manuscript. Both authors reviewed and approved the final version.

Funding Not applicable.

Data Availability No datasets were generated or analysed during the current study.

Declarations

Competing interests The authors declare no competing interests.

Ethics approval and consent to participate Not applicable (not a human and/or animal study).

Consent for publication Both authors and their acknowledged colleagues give their consent for publication. The manuscript contains no more individual person's data other than the authors' contact details and the two names in the acknowledgements.

References

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

Agrawal A, Amos B, Barratt S, Boyd S, Diamond S, Kolter Z (2019) Differentiable convex optimization layers. In: Advances in Neural Information Processing Systems (NeurIPS), vol 32

Agrawal A, Barratt S, Boyd S, Busseti E, Moursi W (2019) Differentiating through a cone program. arXiv preprint arXiv:1904.09043

Agrawal A, Barratt S, Boyd S, Stellato B (2020) Learning convex optimization control policies. In: Learning for Dynamics and Control, PMLR, pp 361–373

Akiba T, Sano S, Yanase T, Ohta T, Koyama M (2019) Optuna: A next-generation hyperparameter optimization framework. In: Proceedings of the 25th ACM SIGKDD international conference on knowledge discovery & data mining, pp 2623–2631

Amos B, Kolter Z (2017) OptNet: Differentiable optimization as a layer in neural networks. In: International Conference on Machine Learning, PMLR, pp 136–145

Anderson B, Moore J (2007) Optimal control: Linear quadratic methods. Courier Corporation

Banjac G, Stellato B, Moehle N, Goulart P, Bemporad A, Boyd S (2017) Embedded code generation using the OSQP solver. In: IEEE Conference on Decision and Control, IEEE, pp 1906–1911

Barratt S, Boyd S (2021) Least squares auto-tuning. Eng Optim 53(5):789–810

Bertsekas D (1991) Linear network optimization: Algorithms and codes. MIT press

Bertsekas D, Gallager R (1992) Data networks. Athena Scientific

Bertsekas P (1997) Nonlinear programming. J Operational Res Soc 48(3):334–334

Bertsimas D, Thiele A (2004) 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, Springer, pp 86–100

Besançon M, Dias Garcia J, Legat B, Sharma A (2023) Flexible differentiable optimization via model transformations. INFORMS J Comput 36(2):456–478

Bishop C, Nasrabadi N (2006) Pattern recognition and machine learning. Springer
Blondel M, Berthet Q, Cuturi M, Frostig R, Hoyer S, Llinares-López F, Pedregosa F, Vert JP (2022) Efficient and modular implicit differentiation. Adv Neural Inf Process Syst 35:5230–5242
Boyd S, Barratt C (1991) Linear controller design: Limits of performance. Citeseer
Boyd S, Vandenberghe L (2004) Convex optimization. Cambridge University Press
Boyd S, El Ghaoui L, Feron E, Balakrishnan V (1994) Linear matrix inequalities in system and control theory. SIAM
Boyd S, Busseti E, Diamond S, Kahn R, Koh K, Nystrup P, Speth J (2017) Multi-period trading via convex optimization. Foundations Trends Optimization 3(1):1–76
Boyd S, Johansson K, Kahn R, Schiele P, Schmelzer T (2024) Markowitz portfolio construction at seventy. arXiv preprint arXiv:2401.05080
Bradbury J, Frostig R, Hawkins P, Johnson M, Leary C, Maclaurin D, Necula G, Paszke A, VanderPlas J, Wanderman-Milne S, Zhang Q (2018) JAX: Composable transformations of Python+NumPy programs. http://github.com/jax-ml/jax
Brown S, Goetzmann W, Ibbotson R, Ross S (1992) Survivorship bias in performance studies. Rev Financial Stud 5(4):553–580
Calamai P, Moré J (1987) Projected gradient methods for linearly constrained problems. Math Program 39(1):93–116
Carson E, Higham N (2018) Accelerating the solution of linear systems by iterative refinement in three precisions. SIAM J Sci Comput 40(2):A817–A847
Chari G, Açıkmeşe B (2025) QOCO: A quadratic objective conic optimizer with custom solver generation. arXiv preprint arXiv:2503.12658
Cortes C (1995) Support-vector networks. Mach Learn 20(3):273–297
Cox D (1958) The regression analysis of binary sequences. J R Stat Soc Ser B Stat Methodol 20(2):215–232
Davis T, Hager W (1999) Modifying a sparse Cholesky factorization. SIAM J Matrix Anal Appl 20(3):606–627
Davis T, Hager W (2005) Row modifications of a sparse Cholesky factorization. SIAM J Matrix Anal Appl 26(3):621–639
Diamond S, Boyd S (2016) CVXPY: A Python-embedded modeling language for convex optimization. J Mach Learn Res 17(83):1–5
Dunning I, Huchette J, Lubin M (2017) JuMP: a modeling language for mathematical optimization. SIAM Rev 59(2):295–320
Elmachtoub A, Grigas P (2022) Smart predict, then optimize". Manage Sci 68(1):9–26
Elton E, Gruber M, Brown S, Goetzmann W (2009) Modern portfolio theory and investment analysis. John Wiley & Sons
Feurer M, Hutter F (2019) Hyperparameter optimization. Springer International Publishing
Frey J, BaumgÃĪrtner K, Frison G, Reinhardt D, Hoffmann J, Fichtner L, Gros S, Diehl M (2025) Differentiable nonlinear model predictive control. arXiv preprint arXiv:2505.01353
Fu A, Narasimhan B, Boyd S (2020) CVXR: an r package for disciplined convex optimization. J Stat Softw 94(14):1–34
Garcia C, Prett D, Morari M (1989) Model predictive control: theory and practice - a survey. Automatica 25(3):335–348
Grant M, Boyd S (2014) CVX: Matlab software for disciplined convex programming, version 2.1
Grinold R, Kahn R (2000) Active portfolio management. McGraw Hill, New York
Halabian H (2019) Distributed resource allocation optimization in 5G virtualized networks. IEEE J Sel Areas Commun 37(3):627–642
Hastie T, Tibshirani R, Friedman J (2009) The elements of statistical learning: Data mining, inference, and prediction. Springer
Higham N (1997) Iterative refinement for linear systems and LAPACK. IMA J Numer Anal 17(4):495–509
Hoerl A, Kennard R (1970) Ridge regression: biased estimation for nonorthogonal problems. Technometrics 12(1):55–67
Holzmann G (2006) The power of 10: rules for developing safety-critical code. Computer 39(6):95–99
Keshavarz A, Boyd S (2014) Quadratic approximate dynamic programming for input-affine systems. Int J Robust Nonlinear Control 24(3):432–449
Kotary J, Di Vito V, Cristopher J, Van Hentenryck P, Fioretto F (2024) Learning joint models of prediction and optimization. arXiv preprint arXiv:2409.04898
Kouvaritakis B, Cannon M (2016) Model predictive control. Springer

Kwakernaak H, Sivan R (1972) Linear optimal control systems. Wiley-InterScience, New York
Ledoit O, Wolf M (2008) Robust performance hypothesis testing with the sharpe ratio. J Empir Financ 15(5):850–859
Lefever W, Aghezzaf E, Hadj-Hamou K (2016) A convex optimization approach for solving the single-vehicle cyclic inventory routing problem. Computers & Operations Research 72:97–106
Li L, Jamieson K, DeSalvo G, Rostamizadeh A, Talwalkar A (2018) Hyperband: a novel bandit-based approach to hyperparameter optimization. J Mach Learn Res 18(185):1–52
Lobo M, Fazel M, Boyd S (2007) Portfolio optimization with linear and fixed transaction costs. Ann Oper Res 152:341–365
Löfberg J (2004) YALMIP: A toolbox for modeling and optimization in Matlab. In: IEEE International Conference on Robotics and Automation (ICRA), IEEE, pp 284–289
Maher G, Boyd S, Kochenderfer M, Matache C, Reuter D, Ulitsky A, Yukhymuk S, Kopman L (2022) A light-weight multi-objective asynchronous hyper-parameter optimizer. arXiv preprint arXiv:2202.07735
Markowitz H (1952) Portfolio selection. Journal of Finance 7(1):77–91
Mattingley J, Boyd S (2010) Real-time convex optimization in signal processing. IEEE Signal Process Mag 27(3):50–61
Mattingley J, Boyd S (2012) CVXGEN: a code generator for embedded convex optimization. Optim Eng 13(1):1–27
Murphy K (2012) Machine learning: A probabilistic perspective. MIT press
Narang R (2013) Inside the black box: A simple guide to quantitative and high-frequency trading. John Wiley & Sons
Nobel P, Candès E, Boyd S (2023) Tractable evaluation of stein's unbiased risk estimate with convex regularizers. IEEE Trans Signal Process 71:4330–4341
Nobel P, LeJeune D, Candès E (2024) RandALO: Out-of-sample risk estimation in no time flat. arXiv preprint arXiv:2409.09781
Palomar D (2025) Portfolio Optimization. Cambridge University Press
Pappas T, Laub A, Sandell N (1980) On the numerical solution of the discrete-time algebraic riccati equation. IEEE Trans Autom Control 25(4):631–641
Paszke A, Gross S, Chintala S, Chanan G, Yang E, DeVito Z, Lin Z, Desmaison A, Antiga L, Lerer A (2017) Automatic differentiation in PyTorch. In: NIPS, vol 31
Rawlings J, Mayne D, Diehl M et al (2017) Model predictive control: Theory, computation, and design. Nob Hill Publishing Madison, WI
Schaller M, Banjac G, Diamond S, Agrawal A, Stellato B, Boyd S (2022) Embedded code generation with CVXPY. IEEE Control Systems Letters 6:2653–2658
Shao J (1993) Linear model selection by cross-validation. J Am Stat Assoc 88(422):486–494
Stellato B, Banjac G, Goulart P, Bemporad A, Boyd S (2020) OSQP: an operator splitting solver for quadratic programs. Math Program Comput 12(4):637–672
Tang B, Khalil E (2024) PyEPO: a PyTorch-based end-to-end predict-then-optimize library for linear and integer programming. Math Program Comput 16(3):1–39
Tibshirani R (1996) Regression shrinkage and selection via the lasso. J R Stat Soc Ser B Stat Methodol 58(1):267–288
Udell M, Mohan K, Zeng D, Hong J, Diamond S, Boyd S (2014) Convex optimization in Julia. In: 2014 first workshop for high performance technical computing in dynamic languages, IEEE, pp 18–28
Vanderbei R (1995) Symmetric quasidefinite matrices. SIAM J Optim 5(1):100–113
Verschueren R, Frison G, Kouzoupis D, Frey J, van Duijkeren N, Zanelli A, Novoselnik B, Albin T, Quirynen R, Diehl M (2021) acados – a modular open-source framework for fast embedded optimal control. Mathematical Programming Computation pp 1–37
Wainwright K (2005) Fundamental methods of mathematical economics. McGraw-Hill
Wang Y, Boyd S (2009) Fast model predictive control using online optimization. IEEE Trans Control Syst Technol 18(2):267–278
Wang Y, O'Donoghue B, Boyd S (2015) Approximate dynamic programming via iterated Bellman inequalities. Int J Robust Nonlinear Control 25(10):1472–1496
Zibulevsky M, Elad M (2010) L1–L2 optimization in signal and image processing. IEEE Signal Process Mag 27(3):76–88
Zou H, Hastie T (2005) Regularization and variable selection via the elastic net. J R Stat Soc Ser B Stat Methodol 67(2):301–320