Generation of Custom Solvers in Rust for Convex Optimization

Hao Zhu[1,2] and Joschka Boedecker[1,2]

[1]IMBIT//BrainLinks-BrainTools
[2]Department of Computer Science, University of Freiburg

June 11, 2026

Abstract

We introduce `cvxgenrust`, an open-source tool for generating custom Rust code that solves families of parameterized convex optimization problems modeled in CVXPY. `cvxgenrust` canonicalizes a problem family, extracts affine maps to Clarabel cone-program data, and generates a specialized Rust crate that updates parameters and calls Clarabel natively at runtime. The generated solver can also be exposed to Python and registered as a custom CVXPY solver. Our code generator supports a wide range of convex optimization problems up to semidefinite programs and exponential-cone problems. Numerical experiments show reduced runtime relative to direct CVXPY solves and performance comparable to CVXPYgen on shared problem classes.

Contents

1 Introduction **3**
- 1.1 Prior and related work . . . 3
- 1.2 Contribution . . . 4
- 1.3 Outline . . . 5

2 `cvxgenrust` **5**
- 2.1 Canonical quadratic cone problems . . . 5
- 2.2 Affine-solve-affine workflow . . . 5
- 2.3 A simple example . . . 7

3 Numerical results **8**

4 Conclusion **8**

5 Appendix: Benchmark problem families **10**
- 5.1 Basic examples . . . 10
- 5.2 Semidefinite programming examples . . . 11
- 5.3 Machine learning examples . . . 11
- 5.4 Control and finance examples . . . 12

1 Introduction

We consider the class of *parameterized convex optimization problems* of the form

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

where $x \in \mathbf{R}^n$ is the optimization variable [AAB+19, SBD+22]. The *family* of problems (1) corresponding to different values of the parameter $\omega \in \mathbf{R}^k$ is called a *problem family*. In this context, we call *a* problem of the form (1) associated with one specific value of ω a *problem instance*. The functions $f_i \colon \mathbf{R}^n \times \mathbf{R}^k \to \mathbf{R}$ for $i = 0, \ldots, m$ are assumed to be convex and $h_i \colon \mathbf{R}^n \times \mathbf{R}^k \to \mathbf{R}$ for $i = 1, \ldots, p$ are assumed to be affine, in the variable x for each fixed ω, so that each problem instance is a convex optimization problem. Parameterized convex optimization problems are ubiquitous in many fields, including machine learning [MB12, AAB+19, BB21], control [SBD+22, MLQH25], and finance [BJK+24, LB24], just to list a few.

In practice, solving a problem instance of the form (1) typically involves two steps. First, the problem instance is *canonicalized* into a standard form, such as a cone program (see §2.1), that can be handled by a generic numerical solver. Second, the canonicalized problem instance is passed to the numerical solver and solved via, *e.g.*, an interior-point method [NN94, BV04]. Although the first problem canonicalization step is usually mathematically tedious, time-consuming, and error-prone, many *domain specific languages* (DSLs) have been developed to automate this step, such as YALMIP [Lof04] and CVX [GB14] for MATLAB, Convex.jl [UMZ+14] for Julia, CVXPY [DB16] for Python, and CVXR [FNB20] for R. These DSLs allow users to express optimization problems in a natural mathematical syntax, and automatically canonicalize them into a standard form that can be solved by generic solvers such as OSQP [SBG+20], SCS [OCPB16], ECOS [DCB13], and Clarabel [GC24].

Most of the DSLs for convex optimization are *parser-solvers* [SBD+22], which means that they parse an instance of the problem (1) and call a generic solver at runtime, for each time the parameter ω changes. This approach is flexible and easy to use, but it can be inefficient when the problem parameter ω changes frequently, as is often the case in real-time embedded applications. In such settings, the parsing and canonicalization steps may become computationally expensive, especially for large-scale problems. Here we address this issue by designing a *code generator*, which parse a given *problem family* and generate custom solvers for it. The generated source code is then compiled into binary libraries that are directly executable and can solve problem instances when the parameter values change, without the need for parsing and canonicalization at runtime.

1.1 Prior and related work

The idea of code generators for embedded convex optimization has been explored in the last several decades. One of the earliest works in this direction is CVXGEN [MB12], which generates custom C code for solving small linear programs (LPs) and convex quadratic programs (QPs) in embedded applications. This software has been used to generate flight code for high-speed onboard convex optimization for precision landing of space rockets by SpaceX [Bla16]. More recently, the CVXPY-based C code generator CVXPYgen [SBD+22]

extends the code generation approach to a much broader class of convex optimization problems, up to second-order cone programs (SOCPs). Custom solvers generated by CVXPYgen are generally faster than those from CVXGEN, and have smaller binary sizes (see [SBD+22] for detailed comparisons). There are several other code generators, which however only support solver generation for a specific problem family, *e.g.*, optimal control problems, including FORCESPRO [Emb26], FORCES NLP [ZDJM20], `acados` [VFK+22], and SCvxPyGen [BCG24].

There are several limitations of the existing code generators for convex optimization. Firstly, existing code generators target C, mainly because C offers native performance, small runtime overhead, and broad support for embedded deployment. In this work, we target Rust in order to retain these systems-level advantages while improving the safety and maintainability of the generated solver code. Rust compiles to efficient native code without a garbage collector, and its ownership model helps prevent memory errors, unintended aliasing, and data races at compile time. These properties make Rust a suitable target for generated solvers used in real-time and embedded applications. Secondly, as far as we know, there is no existing code generator that supports solver generation for semidefinite programs (SDPs) or exponential cone programs, which has become increasingly useful in many applications (see §5.2 and §5.3 for some examples).

1.2 Contribution

This work presents `cvxgenrust`, a code generator for parameterized convex optimization problems that generates custom solvers in Rust. Given a problem family specified in CVXPY, `cvxgenrust` canonicalizes the family once into a standard conic form and generates a Rust crate specialized to that family. At runtime, users only update the problem parameters and call the compiled solver, avoiding repeated parsing and canonicalization.

The generated solver uses Clarabel as its backend conic solver. Since Clarabel is implemented natively in Rust, `cvxgenrust` integrates directly with the Rust ecosystem without foreign-function interfaces or language bindings between the generated solver and the numerical backend. This enables generation of native Rust solver modules that can be used from Rust projects, compiled into binary libraries, or exposed to other languages.

Compared with existing general-purpose code generators for convex optimization problems, `cvxgenrust` targets Rust and supports a broader class of cone programs through Clarabel, including LPs, QPs, SOCPs, SDPs, and exponential cone programs. To the best of our knowledge, `cvxgenrust` is the first code generator for parameterized convex optimization that combines native Rust solver generation with support for semidefinite and exponential cones.

In addition to the Rust module, `cvxgenrust` generates Python bindings that allow the generated solver to be registered as a custom solver in CVXPY. Users can therefore keep the usual CVXPY modeling workflow while reducing runtime overhead for repeated solves of the same problem family.

We evaluate `cvxgenrust` on problem families from several application domains. The numerical results show that generated `cvxgenrust` solvers reduce runtime compared with standard CVXPY workflows and achieve solve times comparable to CVXPYgen on problem classes supported by both tools.

Last but not least, `cvxgenrust` is fully open-source and available at

`https://github.com/dxogrp/cvxgenrust`.

1.3 Outline

The rest of this paper is organized as follows. In §2 we provide a high-level overview of the basic ideas behind `cvxgenrust`, followed by a simple example of how these ideas work and how to use `cvxgenrust` for code generation. In §3 we present numerical results on the performance of `cvxgenrust` on a variety of problem families. (The detailed problem formulations of the compared problem families can be found in the appendix §5.) We conclude the paper in §4 with a summary and discussion of future work.

2 `cvxgenrust`

2.1 Canonical quadratic cone problems

CVXPY allows users to describe convex optimization problems in a high-level mathematical syntax, but numerical solvers operate on canonical forms. For the Clarabel backend used by `cvxgenrust`, each CVXPY problem instance is canonicalized into a quadratic cone program

$$\begin{array}{ll} \text{minimize} & (1/2)\tilde{x}^T P \tilde{x} + q^T \tilde{x} \\ \text{subject to} & A\tilde{x} + s = b \\ & s \in \mathcal{K}, \end{array} \tag{2}$$

where $\tilde{x}$ is the canonical optimization variable, s is the slack variable, and $\mathcal{K}$ is a product cone. The data P, q, A, and b are the canonical numerical data passed to Clarabel, while $\mathcal{K}$ is determined automatically by the atoms and constraints in the original CVXPY model. Here the supported cone components include zero, nonnegative, second-order, positive semidefinite, exponential, and power cones.

2.2 Affine-solve-affine workflow

We could similarly define the data, or parameters, of the canonicalized cone program (2) as $\tilde{\omega} = \{P, q, A, b\}$. The important point for code generation is that, for *disciplined parametrized programming* (DPP) compliant problems [AAB+19], the canonical problem parameters $\tilde{\omega}$ depend affinely on the parameters ω of the original problem (1), and the original problem solution $x^\star$ can be recovered from the canonical solution $\tilde{x}^\star$ via an affine map. In other words, we have

$$\tilde{\omega} = F \begin{bmatrix} \omega \\ 1 \end{bmatrix}, \qquad x^\star = G \begin{bmatrix} \tilde{x}^\star \\ 1 \end{bmatrix},$$

where F and G are sparse matrices depending only on the structure of the problem (1), and can be extracted from the CVXPY-Clarabel canonicalization process.

This *affine-solve-affine* workflow [AAB+19] separates offline code generation from runtime solving. Offline, `cvxgenrust` extracts the canonicalization maps; at runtime, it updates the parameters, reconstructs the cone program, calls Clarabel, and maps the solution back. Figure 1 summarizes this workflow.

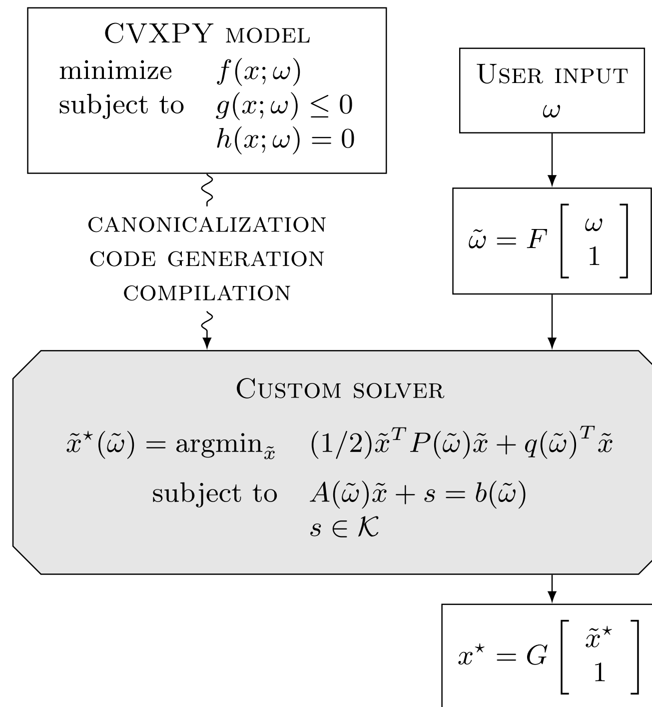

Figure 1 Workflow of `cvxgenrust`. The left part shows the offline code generation process and the right part shows the runtime solving process.

2.3 A simple example

As a minimal example, consider the nonnegative least-squares problem

$$\begin{array}{ll} \text{minimize} & \|Cx - d\|_2^2 \\ \text{subject to} & x \succeq 0, \end{array} \tag{3}$$

with parameters $C \in \mathbf{R}^{m \times n}$ and $d \in \mathbf{R}^m$ ($\succeq$ denotes componentwise inequality). Introducing the residual variable $t = Cx - d$, we can rewrite (3) in the form of (2) with canonical variable $\tilde{x} = \begin{bmatrix} x \\ t \end{bmatrix} \in \mathbf{R}^{n+m}$ and slack variable $s = \begin{bmatrix} s_1 \\ s_2 \end{bmatrix} \in \mathbf{R}^{m+n}$. The corresponding problem data are

$$P = \begin{bmatrix} 0 & 0 \\ 0 & 2I \end{bmatrix}, \qquad q = 0, \qquad A = \begin{bmatrix} C & -I \\ -I & 0 \end{bmatrix}, \qquad b = \begin{bmatrix} d \\ 0 \end{bmatrix},$$

with cone $\mathcal{K} = \{0\}^m \times \mathbf{R}^n_+$ (where the first block enforces $Cx - t = d$, and the second block enforces $x \succeq 0$). In this example, the canonical problem parameters P, q, A, and b are indeed affine in the original problem parameters C and d.

The following CVXPY code generates a Rust solver crate for this problem family.

```python
import cvxpy as cp
import cvxgenrust as cgr

# Define the problem family
x = cp.Variable(n, name="x")
C = cp.Parameter((m, n), name="C")
d = cp.Parameter(m, name="d")

problem = cp.Problem(
    cp.Minimize(cp.sum_squares(C @ x - d)),
    [x >= 0]
)

# Generate the custom solver code
cgr.generate_code(
    problem,
    module_name="nonneg_ls"
)
```

The names assigned to CVXPY parameters and variables are used to generate stable parameter setters and solution extractors. After code generation, the resulting Cargo project can be built and used from Rust. For example, a Rust application can create a generated problem object, set the current values of C and d, solve the problem, and extract the named variable x:

```rust
use nonneg_ls_cgr::CGRProblem;

// Create a new problem instance
let mut problem = CGRProblem::new();

// Update the problem parameters
```

```
problem.set_c(&c_values)?;
problem.set_d(&d_values)?;

// Solve and extract the solution
let solution = problem.solve()?;
let x = problem.extract_x(&solution.x)?;
```

To register and use the generated solver in CVXPY, the generated Python bindings can be used as follows:

```
from nonneg_ls_wrapper.cgr_solver import cgr_solve

# Register the generated solver in CVXPY
problem.register_solve("CGR", cgr_solve)

# Update the problem parameters
C.value = C_value
d.value = d_value

# Solve with the generated solver
problem.solve(
    method="CGR",
    updated_params=["C", "d"]
)
```

3 Numerical results

We benchmarked `cvxgenrust` on the twelve problem families listed in §5, with five problem sizes per family. Each case used three warm-up solves and 1000 timed samples; all methods used Clarabel with the same tolerances where applicable. Experiments ran on Linux with an AMD Ryzen 9 7950X CPU.

We compare direct CVXPY solves, the generated `cvxgenrust` solver called through its Python wrapper, the generated `cvxgenrust` Rust crate called directly, and the corresponding Python and C interfaces generated by CVXPYgen. Note that CVXPYgen is not available for the parametric quadratic-form, SDP, and logistic-regression examples in this benchmark suite.

Figure 2 shows that both generated `cvxgenrust` interfaces reduce runtime relative to direct CVXPY solves, with median speedups of 2.66× through Python and 6.27× through direct Rust calls over the 60 problem-size cases. This improvement comes from avoiding repeated CVXPY parsing and canonicalization after code generation. On the 40 problem-size cases supported by CVXPYgen, the `cvxgenrust` Python and Rust interfaces are close to the corresponding CVXPYgen Python and C interfaces.

4 Conclusion

We presented `cvxgenrust`, a code generator that turns DPP-compliant CVXPY problem families into specialized Rust solver crates. By extracting affine canonicalization maps offline and calling Clarabel natively at runtime, `cvxgenrust` avoids repeated parsing and canonicalization while supporting a broad class of cone programs, including LPs, QPs, SOCPs,

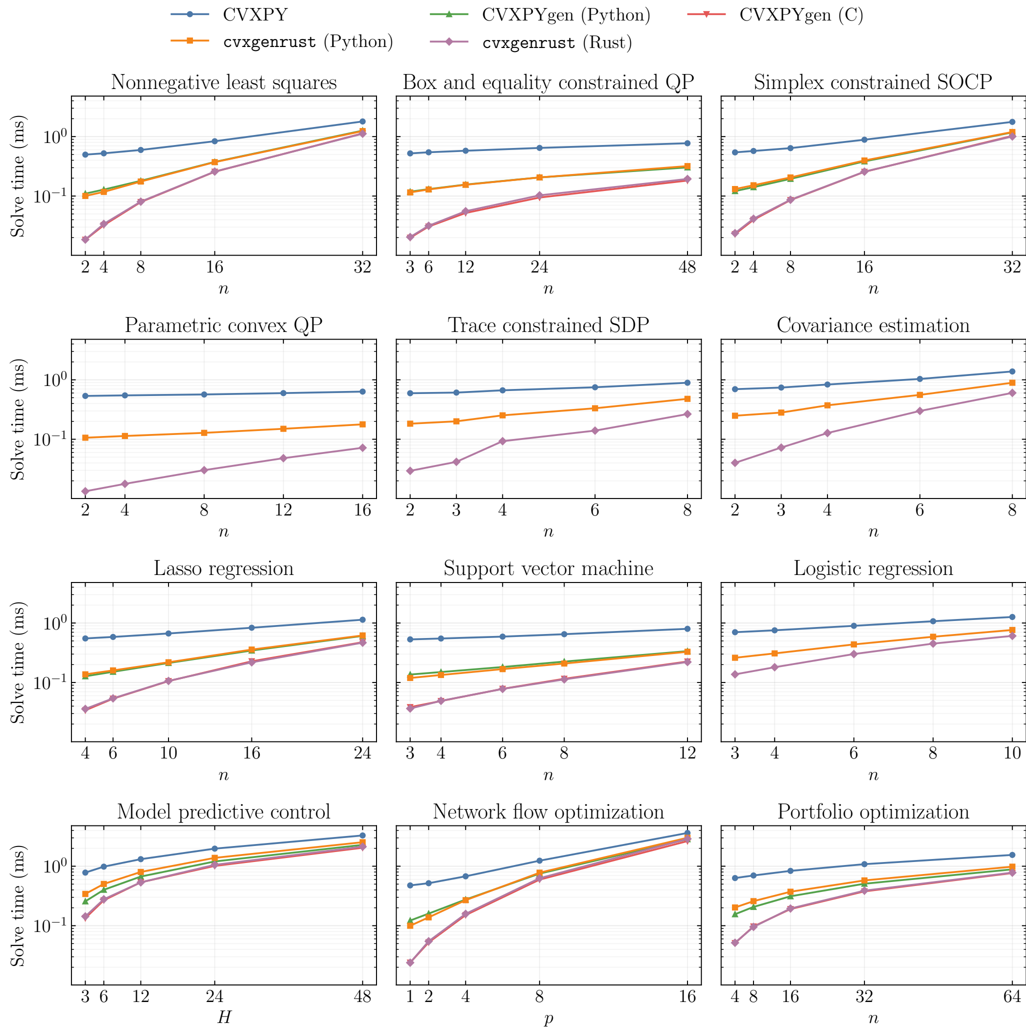

Figure 2 Comparison of solve times for CVXPY, CVXPYgen, and `cvxgenrust`(shown in different colors). Each point represents the median solve time for one problem size over the timed samples.

SDPs, and exponential-cone problems. Numerical experiments show reduced runtime compared with direct CVXPY solves and performance comparable to CVXPYgen on shared problem classes.

Future work involves extending `cvxgenrust` to support differentiating through the generated solvers (*e.g.*, as in [HNB26]), which would enable end-to-end learning of problem parameters in applications such as machine learning and control. Moreover, we could make `cvxgenrust` to support generating explicit solver code for specific problem families, *e.g.*, a parameterized QP [SABB25], which can be solved more efficiently than general cone programs.

5 Appendix: Benchmark problem families

In this section we present the problem families used in the numerical experiments. For the detailed problem dimensions and parameter generation procedures, please refer to the code in our repository

https://github.com/dxogrp/cgr-benchmarks.

5.1 Basic examples

Nonnegative least squares. We consider the problem of finding a nonnegative vector $x \in \mathbf{R}^n$ that minimizes the least squares error with respect to a given matrix $A \in \mathbf{R}^{m \times n}$ and a vector $b \in \mathbf{R}^m$:

$$\begin{array}{ll} \text{minimize} & \|Ax - b\|_2^2 \\ \text{subject to} & x \succeq 0, \end{array}$$

where $\succeq$ represents elementwise inequality. This problem is parameterized by the matrix A and the vector b.

Box and equality constrained QP. We consider the following QP with box and equality constraints:

$$\begin{array}{ll} \text{minimize} & (1/2)\|x - x_0\|_2^2 + q^T x \\ \text{subject to} & Ax = b \\ & 0 \preceq x \preceq 1 \end{array}$$

where $x \in \mathbf{R}^n$ is the optimization variable, and $x_0 \in \mathbf{R}^n$, $q \in \mathbf{R}^n$, $A \in \mathbf{R}^{m \times n}$, and $b \in \mathbf{R}^m$ are parameters of the problem.

Simplex constrained SOCP. We consider the following SOCP with the probabilistic simplex constraint:

$$\begin{array}{ll} \text{minimize} & \|Ax - b\|_2^2 + 0.1\|x\|_2^2 \\ \text{subject to} & \|x\|_2 \leq \rho \\ & x \succeq 0, \quad \mathbf{1}^T x = 1, \end{array}$$

where $x \in \mathbf{R}^n$ is the variable, and $A \in \mathbf{R}^{m \times n}$, $b \in \mathbf{R}^m$, and $\rho \in \mathbf{R}_+$ are the parameters.

Parametric convex QP. We consider the following convex QP with the quadratic coefficient matrix $P \in \mathbf{S}^n_+$ and a vector $q \in \mathbf{R}^n$ as parameters:

$$\begin{array}{ll} \text{minimize} & x^T P x + q^T x \\ \text{subject to} & x \succeq 0, \quad \mathbf{1}^T x \leq 1, \end{array}$$

where $x \in \mathbf{R}^n$ is the variable. Note that `cvxgenrust` allows modeling of this problem family, but it is not yet supported by CVXPYgen [SBD+22].

5.2 Semidefinite programming examples

Trace constrained SDP. We consider the following SDP with equality constraints:

$$\begin{array}{ll} \text{minimize} & \mathbf{tr}(C^T X) \\ \text{subject to} & X \succeq 0, \quad \mathbf{tr}\, X = 1, \end{array}$$

where $X \in \mathbf{S}^n$ is the variable, and $C \in \mathbf{S}^n$ is the parameter. Here the inequality $\succeq$ denotes matrix inequality, *i.e.*, X is constrained to be positive semidefinite.

Covariance estimation. We consider the problem of recovering a covariance matrix $X \in \mathbf{S}^n_+$ from a given noisy observation $S \in \mathbf{S}^n$:

$$\begin{array}{ll} \text{minimize} & \|X - S\|_F^2 + 0.15 \sum_{i \neq j} (X_{ij} - S_{ij})^2 \\ \text{subject to} & X \succeq 0 \\ & X_{ii} = S_{ii}, \quad i = 1, \ldots, n, \end{array}$$

where $\|\cdot\|_F$ denotes the Frobenius norm, X_{ij} and S_{ij} are the (i,j)th entries of X and S, respectively, and the problem is parameterized by the noisy observation S.

5.3 Machine learning examples

Lasso regression. The lasso regression problem is given by

$$\text{minimize} \quad (1/2)\|Ax - b\|_2^2 + \lambda \|x\|_1,$$

where $x \in \mathbf{R}^n$ is the variable, and $A \in \mathbf{R}^{m \times n}$, $b \in \mathbf{R}^m$, and $\lambda \in \mathbf{R}_+$ are the parameters.

Support vector machine. We consider the soft-margin support vector machine problem:

$$\text{minimize} \quad (1/2)\|w\|_2^2 + \textstyle\sum_{i=1}^m (1 - y_i(X_i w + \beta))_+,$$

where $w \in \mathbf{R}^n$ and $\beta \in \mathbf{R}$ are the variables, and $X \in \mathbf{R}^{m \times n}$ and $y \in \{-1, 1\}^m$ are the parameters, where $(u)_+$ denotes the positive part of $u \in \mathbf{R}$.

Logistic regression. The logistic regression problem is given by

$$\text{minimize} \quad (1/m) \textstyle\sum_{i=1}^m \log(1 + \exp(-y_i(X_i w + \beta))) + (\lambda/2)\|w\|_2^2,$$

where $w \in \mathbf{R}^n$ and $\beta \in \mathbf{R}$ are the variables, and $X \in \mathbf{R}^{m \times n}$, $y \in \{-1, 1\}^m$, and $\lambda \in \mathbf{R}_+$ are the parameters. Note that code generation for this problem family requires canonicalizing it into a conic form involving the exponential cone, which is currently not compatible with CVXPYgen.

5.4 Control and finance examples

Model predictive control. We consider the following finite-horizon linear-quadratic tracking problem. Let $x_t \in \mathbf{R}^3$ be the system state at time t for $t = 0, \dots, H$, and let $u_t \in \mathbf{R}^2$ be the control input at time t for $t = 0, \dots, H-1$. Given an initial state $\bar{x}_0 \in \mathbf{R}^3$ and a reference trajectory $r_t \in \mathbf{R}^3$ for $t = 0, \dots, H$, we consider the following optimization problem:

$$\begin{array}{ll}
\text{minimize} & \sum_{t=0}^{H} \|Q^{1/2}(x_t - r_t)\|_2^2 + \sum_{t=0}^{H-1} \|R^{1/2}u_t\|_2^2 + 0.03\|x_H\|_2^2 \\
\text{subject to} & x_0 = \bar{x}_0, \quad x_{t+1} = Ax_t + Bu_t, \quad t = 0, \dots, H-1 \\
& |u_{t+1} - u_t| \leq 0.35, \quad t = 0, \dots, H-2 \\
& |x_{1,t}| \leq 2.5, \quad |x_{2,t}| \leq 2.5, \quad t = 0, \dots, H \\
& |u_t| \leq 1, \quad t = 0, \dots, H-1,
\end{array}$$

where x_t and u_t are the optimization variables. In this example we assume that the system dynamics $A \in \mathbf{R}^{3\times 3}$ and $B \in \mathbf{R}^{3\times 2}$, as well as the quadratic cost matrices $Q \in \mathbf{S}_+^3$ and $R \in \mathbf{S}_+^2$, are constants, so the problem is only parameterized by the initial state $\bar{x}_0$ and the reference trajectory r_t for $t = 0, \dots, H$.

Network flow optimization. We consider a capacitated multi-commodity flow problem on a directed graph with n vertices and m edges. Let p denote the number of commodities, and for each commodity $i = 1, \dots, p$, let $f_i \in \mathbf{R}^m$ represents the flow of the ith commodity on each directed edge, so $f = \begin{bmatrix} f_1 & \cdots & f_p \end{bmatrix}^T \in \mathbf{R}^{p\times m}$ is the commodity-edge flow matrix. The network flow optimization problem is given by

$$\begin{array}{ll}
\text{minimize} & \sum_{i=1}^{p} c_i^T f_i \\
\text{subject to} & f_i \succeq 0, \quad Bf_i = d_i b_i, \quad i = 1, \dots, p \\
& \sum_{i=1}^{p} f_{ij} \leq u_j, \quad j = 1, \dots, m,
\end{array}$$

where $c_i \in \mathbf{R}^m$ is the edge cost vector for the ith commodity, $B \in \mathbf{R}^{n\times m}$ is the vertex-edge incidence matrix of the graph, *i.e.*, $B_{ij} = 1$ if edge j enters vertex i, $B_{ij} = -1$ if edge j leaves vertex i, and $B_{ij} = 0$ otherwise. The vector $d \in \mathbf{R}_+^p$ records the demand for all commodities, and the vector $b_i \in \mathbf{R}^n$ encodes the source and sink of this commodity, which has value 1 at the source vertex, value -1 at the sink vertex, and value 0 elsewhere. Therefore the conservation constraint $Bf_i = d_i b_i$ ensures that the flow of the ith commodity satisfies the demand d_i from the source to the sink. The vector $u \in \mathbf{R}_+^m$ is the edge capacity vector. This problem has variable $f \in \mathbf{R}^{p\times m}$, and is parameterized by $c_1, \dots, c_p \in \mathbf{R}^m$, $d \in \mathbf{R}_+^p$, and $u \in \mathbf{R}_+^m$. The matrix $B \in \mathbf{R}^{n\times m}$ and the vectors $b_1, \dots, b_p \in \mathbf{R}^n$ are assumed to be constants.

Portfolio optimization. We consider the long-only portfolio optimization problem with n assets, given by

$$\begin{array}{ll}
\text{minimize} & \gamma\left(\|F^T w\|_2^2 + \|D^{1/2}w\|_2^2\right) - \mu^T w + 0.02\|w - w^{\text{prev}}\|_1 \\
\text{subject to} & 0 \preceq w \preceq w^{\max}, \quad \mathbf{1}^T w = 1 \\
& \|w - w^{\text{prev}}\|_1 \leq 0.75, \quad Sw \preceq 0.65\mathbf{1}, \quad \mu^T w \geq r^{\text{tar}},
\end{array}$$

where the wealth allocation vector $w \in \mathbf{R}^n$ is the variable. The matrix $F \in \mathbf{R}^{n \times p}$ is the factor loading matrix with p factors, and $D \in \mathbf{S}_+^n$ is the diagonal idiosyncratic risk matrix, so $\gamma \geq 0$ is the risk aversion parameter that controls the trade-off between risk and return in the objective. The vector $\mu \in \mathbf{R}^n$ is the expected return vector, and $w^{\text{prev}} \in \mathbf{R}^n$ is the previous wealth allocation. The matrix $S \in \mathbf{R}^{q \times n}$ encodes the sector information of the assets, where q is the number of sectors and $S_{ij} = 1$ if asset j belongs to sector i and $S_{ij} = 0$ otherwise. The vector $w^{\max} \in \mathbf{R}_+^n$ is the maximum allocation vector, and $r^{\text{tar}} \in \mathbf{R}$ is the target return. We assume that the problem is parameterized by γ, μ, w^{prev}, and r^{tar}, while the matrices F, D, and S, as well as the vector $w^{\max}$, are constants.

Acknowledgments

This work has been funded as part of BrainLinks-BrainTools, which is funded by the Federal Ministry of Economics, Science and Arts of Baden-Württemberg within the sustainability program for projects of the Excellence Initiative II, and CRC/TRR 384 "IN-CODE".

References

[AAB+19] A. Agrawal, B. Amos, S. Barratt, S. Boyd, S. Diamond, and J. Z. Kolter. Differentiable convex optimization layers. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 32. Curran Associates, Inc., 2019.

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

[BCG24] D. Berrah, A. Chapoutot, and P.-L. Garoche. SCvxPyGen: Autocoding SCvx algorithm. In *2024 IEEE 63rd Conference on Decision and Control (CDC)*, pages 5086–5093. IEEE, 2024.

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

[Bla16] L. Blackmore. Autonomous precision landing of space rockets. In *Frontiers of Engineering: Reports on Leading-Edge Engineering from the 2016 Symposium*, pages 33–41. The National Academies Press, 2016.

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

[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.

[DCB13] A. Domahidi, E. Chu, and S. Boyd. ECOS: An SOCP solver for embedded systems. In *European Control Conference*, pages 3071–3076. IEEE, 2013.

[Emb26] Embotech AG. FORCES Professional, 2026. Available at https://embotech.com/forcespro.

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

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

[GC24] P. J. Goulart and Y. Chen. Clarabel: An interior-point solver for conic programs with quadratic objectives. *arXiv*, 2405.12762, 2024.

[HNB26] Q. Healey, P. Nobel, and S. Boyd. Differentiating through a quadratic cone program. *Optimization Letters*, 2026. Preprint on *arXiv:2508.17522*.

[LB24] E. Luxenberg and S. Boyd. Portfolio construction with Gaussian mixture returns and exponential utility via convex optimization. *Optimization and Engineering*, 25:555–574, 2024.

[Lof04] J. Lofberg. YALMIP: A toolbox for modeling and optimization in MATLAB. In *Proceedings of the IEEE International Symposium on Computed Aided Control Systems Design*, pages 294–289. IEEE, 2004.

[MB12] J. Mattingley and S. Boyd. CVXGEN: A code generator for embedded convex optimization. *Optimization and Engineering*, 13:1–27, 2012.

[MLQH25] Y. Ma, Y. Liu, K. Qu, and M. Hutter. Learning accurate whole-body throwing with high-frequency residual policy and pullback tube acceleration. In *2025 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 1771–1778. IEEE, 2025.

[NN94] Y. Nesterov and A. Nemirovskii. *Interior-Point Polynomial Algorithms in Convex Programming*. Studies in Applied and Numerical Mathematics. Society for Industrial and Applied Mathematics, 1994.

[OCPB16] B. O'Donoghue, E. Chu, N. Parikh, and S. Boyd. Conic optimization via operator splitting and homogeneous self-dual embedding. *Journal of Optimization Theory and Applications*, 169(3):1042–1068, 2016.

[SABB25] M. Schaller, D. Arnström, A. Bemporad, and S. Boyd. Automatic generation of explicit quadratic programming solvers. *arXiv*, 2506.11513, 2025.

[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.

[UMZ+14] M. Udell, K. Mohan, D. Zeng, J. Hong, S. Diamond, and S. Boyd. Convex optimization in Julia. In *Proceedings of the Workshop for High Performance Technical Computing in Dynamic Languages*, pages 18–28, 2014.

[VFK+22] R. Verschueren, G. Frison, D. Kouzoupis, J. Frey, N. van Duijkeren, A. Zanelli, B. Novoselnik, T. Albin, R. Quirynen, and M. Diehl. `acados` — a modular open-source framework for fast embedded optimal control. *Mathematical Programming Computation*, 14:147–183, 2022.

[ZDJM20] A. Zanelli, A. Domahidi, J. Jerez, and M. Morari. FORCES NLP: an efficient implementation of interior-point methods for multistage nonlinear nonconvex programs. *International Journal of Control*, 93(1):13–29, 2020.