Automatic Generation of Explicit Quadratic Programming Solvers

Maximilian Schaller Daniel Arnström Alberto Bemporad
Stephen Boyd

June 12, 2025

Abstract

We consider a family of convex quadratic programs in which the coefficients of the linear objective term and the righthand side of the constraints are affine functions of a parameter. It is well known that the solution of such a parametrized quadratic program is a piecewise affine function of the parameter. The number of (polyhedral) regions in the solution map can grow exponentially in problem size, but when the number of regions is moderate, a so-called explicit solver is practical. Such a solver computes the coefficients of the affine functions and the linear inequalities defining the polyhedral regions offline; to solve a problem instance online it simply evaluates this explicit solution map. Potential advantages of an explicit solver over a more general purpose iterative solver can include transparency, interpretability, reliability, and speed. In this paper we describe how code generation can be used to automatically generate an explicit solver from a high level description of a parametrized quadratic program. Our method has been implemented in the open-source software CVXPYgen, which is part of CVXPY, a domain specific language for general convex optimization.

Contents

1 Introduction

1.1 Parametric convex optimization

A parametric convex optimization problem can be written as

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

where $x \in \mathbf{R}^n$ is the variable and $\theta \in \Theta \subseteq \mathbf{R}^p$ is the parameter, *i.e.*, data that is given and known whenever (1) is solved. The objective function f_0 and the inequality constraint functions f_i, $i = 1,\ldots,m$, are convex in x and the equality constraint functions h_i, $i = 1,\ldots,q$, are affine in x, for any given value of $\theta \in \Theta$ [BV04]. We refer to a solution of (1) as $x^\star(\theta)$ to emphasize its dependence on the parameter θ. Here we neglect that $x^\star$ might not be unique or might not exist, and refer to the mapping from θ to x as the solution map of the parametrized problem.

Convex optimization is used in various domains, including control systems [BBM17, RMD+17, KC16, KB14, WB09, BM07, BB91, GPM89], signal and image processing [CP16, CP11, MB10, ZE10], and quantitative finance [Pal25, BJK+24, BBD+17, Nar13, GK00, Mar52], just to name a few that are particularly relevant for this work.

Explicit solvers for multiparametric programming. Traditionally, $x^\star(\theta)$ is evaluated using an iterative numerical method that takes a given parameter value and computes an (almost) optimal point $x^\star(\theta)$ [GC24, SBG+20, OCPB16, DCB13]. We focus here on a very special case when $x^\star(\theta)$ can be expressed in closed form, as an explicit function that maps a given value of θ directly to a solution $x^\star(\theta)$. Such explicit solvers are practical for only some problems, and generally only smaller instances, but when they are practical they can offer a number of advantages over generic iterative solvers. Developing such solvers for parametric programs is now known as multiparametric programming.

1.2 Related work

Multiparametric programming. Investigations of the theoretical properties of parametric convex optimization problems date back to the 60s [MR64] and were largely extended in the 80s [Fia83]. After early work on explicitly solving parametric linear programs (LPs) in the context of economics [GN72] in the 70s, researchers started investigating the explicit solution of different convex optimization problems in the early 2000s. Some of the first papers developed explicit solutions to model predictive control problems based on quadratic programs (QPs) [BMDP02] and LPs [BBM+02], and general algorithms were developed for explicitly solving QPs [BMDP02, TJB03a, PS10, GBN11] and LPs [BBM03]. These are often cited as multiparametric linear or quadratic programming, respectively, where the latter is often abbreviated as MPQP. Further, people have worked on verifying the

complexity of such methods [CB17], on approximate or suboptimal multiparametric programming [BF03, JG03, BF06], on multiparametric programming for linear complementary problems [JM06], and on synthesizing specialized hardware for multiparametric programming [JJST06, BOPS11, ROL+23].

Software implementations include the Hybrid Toolbox [Bem04], the Multi-Parametric Toolbox [HKJM13], the Model Predictive Control Toolbox [Bem15], and the POP Toolbox [ODP+16] in Matlab, the MPQP solver in the proprietary FORCES PRO software [DJ14], and the PDAQP solver [AA24] in Julia and Python, which we use in this work.

Potential advantages of an explicit solver over a more general purpose iterative solver can include transparency, interpretability, reliability, and speed. Since the solver is essentially a lookup table, with an explicit affine function associated with each region, there is no question of convergence. Indeed, we can explicitly determine the maximum number of floating-point operations (FLOPS) required to compute $x^\star(\theta)$ given θ [CB17, ABA24]. An explicit solver involves no division, so floating-point overflow or divide-by-zero exceptions cannot occur. For the same reason, it is possible to store the coefficients in a lower precision format such as 16-bit floating-point (FP16) to reduce storage, and possibly to carry out the computations in lower precision as well, to increase speed or use low-cost electronic boards (at the cost of a modest decrease in accuracy).

The disadvantages of using an explicit solver all relate to its worst-case exponential scaling with problem size. This limits its practical use to relatively small problems, which however do arise in many application areas. Even when it is practical to use an explicit solver, the solver data size can be large, since we must store the coefficients of the explicit solution map.

Code generation for convex optimization. While typical convex optimization solvers are designed for general-purpose computers [DB16, GC24], we are mostly interested in embedded applications with hard real-time constraints, and also non-embedded applications where extreme speeds are required.

A *code generator* heavily exploits the structure of the functions in (1) and generates custom C code for solving the problem fast and reliably for changing values of θ, while fulfilling rules for safety-critical code [Hol06]. Examples of code generators for iterative solvers are CVXGEN [MB12] (for general QPs), CVXPYgen [SBD+22], which interfaces with the OSQP code generator [BSM+17] and QOCOGEN [CA25] (for QPs and second-order cone programs, respectively), and acados [VFK+21] and the proprietary FORCES PRO [DJ14], both specifically designed for QP-based and nonlinear control problems. These code generators are used for many applications. CVXGEN, for example, is used to guide and control all of the SpaceX first stage landings [Bla16].

Domain-specific languages for optimization. Code generators typically accept problem specifications given in a domain specific language (DSL) for convex optimization [Löf04, GB14]. A DSL allows the user to describe the problem in a natural high level human readable way, eliminating the effort (and risk of error) in transforming the problem to the standard form required by a solver. For example there can be multiple variables or parameters, with

names that make sense in the application; the code generator takes care of mapping these to our generic $x \in \mathbf{R}^n$ and $\theta \in \Theta$. Well-known DSLs include YALMIP [Löf04] and CVX [GB14] in Matlab, CVXPY [DB16] in Python, Convex.jl [UMZ+14] and JuMP [DHL17] in Julia, and CVXR [FNB20] in R. We focus on CVXPY.

1.3 Contribution

In this paper, we adapt the code generator CVXPYgen with the multiparametric explicit QP solver PDAQP to generate explicit solvers for convex optimization problems, described in CVXPY, that can be reduced to QPs. Along with C and C++ code for the generated solver, we generate a Python interface for rapid prototyping and non-embedded applications.

We give four representative application examples, involving linear regression, power management in residential buildings, model predictive control, and financial portfolio optimization. Our code generator accelerates the solve time for these examples by up to three orders of magnitude compared to directly using CVXPY and its default QP solver, with solve times down to hundreds of nanoseconds (on a standard laptop).

1.4 Outline

In §2 we give a quick derivation of the explicit solution map for a parametric QP. In §3 we explain how we generate source code for an explicit solver that is described in the modeling language CVXPY, and how to interface with the generated code. We illustrate the explicit solver code generation process in §4. In §5 we assess the numerical performance of the explicit solvers for several practical examples. We report the time it takes to generate and compile the generated code, the size of the resulting binary files, and the solve times.

2 Explicit solution of parametric QPs

2.1 Parametric QP

We consider the QP

$$\begin{array}{ll} \text{minimize} & (1/2)x^T P x + q^T x \\ \text{subject to} & Ax \leq b, \end{array} \tag{2}$$

where $x \in \mathbf{R}^n$ is the variable, and the data are $P \in \mathbf{S}^n_{++}$ (the set of symmetric positive definite $n \times n$ matrices), $q \in \mathbf{R}^n$, $A \in \mathbf{R}^{m \times n}$, and $b \in \mathbf{R}^m$. The inequality in the constraints is elementwise.

Our focus is on the case when the data P and A are given, and q and b are affine functions of a parameter $\theta \in \mathbf{R}^p$,

$$q = u + U\theta, \qquad b = v + V\theta, \tag{3}$$

where $u \in \mathbf{R}^n$, $U \in \mathbf{R}^{n \times p}$, $v \in \mathbf{R}^m$, and $V \in \mathbf{R}^{m \times p}$ are given. We refer to the QP (2) parametrized by θ in (3) as a parametrized QP. Since the objective is strictly convex, there

is at most one solution of the QP (2) for each value of θ. We refer to the mapping from θ to the optimal x (when it exists) as the solution map of the parametrized QP (2).

Other QP forms. There are several other standard forms for a parametrized QP, both for analysis and as an interface to solvers [BMDP02, TJB03a, AA24], but it is easy to translate between them by introducing additional variables and constraints [BV04].

Constraints on the parameters. In many applications we are also given a set $\Theta \subseteq \mathbf{R}^p$ of possible parameter values. For simplicity we ignore this, but occasionally mention how this set of known possible values of θ can be handled. When we do address the parameter set, we assume it is a polyhedron.

2.2 Optimality conditions

Active constraints. We denote the ith row of A as a_i^T, so the inequality constraints in (2) can be expressed as $a_i^T x \leq b_i$, $i = 1, \ldots, m$. We say that the ith inequality constraint is tight or active if $a_i^T x = b_i$. We let $\mathcal{A} = \{i \mid a_i^T x = b_i\} \subseteq \{1, \ldots, m\}$ denote the set of active constraints [CB17, GMW19] (which depends on x, A, and b).

Let $\lambda \in \mathbf{R}_+^m$ denote a dual variable associated with the linear inequality constraints in (2). The optimality conditions for problem (2) are

$$\begin{aligned} &Ax \leq b, \\ &\lambda \geq 0, \\ &Px + q + A^T\lambda = 0, \\ &\lambda_i(a_i^T x - b_i) = 0, \quad i = 1, \ldots, m. \end{aligned}$$

The first is primal feasibility; the second is nonnegativity of dual variables; the third is dual feasibility (stationarity of the Lagrangian); and the last one is complementary slackness [BV04, §5.5.3].

Let $\tilde{A}$, $\tilde{b}$, and $\tilde{\lambda}$ denote the row slices of A, b, and λ, respectively, corresponding to the active constraints, *i.e.*, $i \in \mathcal{A}$. Let $\hat{A}$, $\hat{b}$, and $\hat{\lambda}$ denote the row slices of A, b, and λ, respectively, corresponding to the inactive constraints, *i.e.*, $i \notin \mathcal{A}$. By complementary slackness, we must have $\lambda_i = 0$ for $i \notin \mathcal{A}$, so $\hat{\lambda} = 0$. With $\hat{\lambda} = 0$, which we now assume, complementary slackness holds. Since $\hat{\lambda} = 0$, $A^T\lambda$ can be expressed as $\tilde{A}^T\tilde{\lambda}$, and dual feasibility can be expressed as

$$Px + q + \tilde{A}^T\tilde{\lambda} = 0.$$

Since $\mathcal{A}$ is the active set corresponding to x, we have

$$\tilde{A}x = \tilde{b}.$$

These two sets of linear equations can be summarized as the Karush-Kuhn-Tucker (KKT) system

$$\begin{bmatrix} P & \tilde{A}^T \\ \tilde{A} & 0 \end{bmatrix} \begin{bmatrix} x \\ \tilde{\lambda} \end{bmatrix} = \begin{bmatrix} -q \\ \tilde{b} \end{bmatrix}. \tag{4}$$

We assume that linear independence constraint qualification (LICQ) [Ber99, NW06, BV04] holds, *i.e.*, that the rows of $\tilde{A}$ are linearly independent. Then, it is well known that (4) can be uniquely solved for $(x, \tilde{\lambda})$ [BV04, §10.1.1], [BV18, §12.3] and we re-write (4) as

$$\begin{bmatrix} x \\ \tilde{\lambda} \end{bmatrix} = \begin{bmatrix} P & \tilde{A}^T \\ \tilde{A} & 0 \end{bmatrix}^{-1} \begin{bmatrix} -q \\ \tilde{b} \end{bmatrix}. \tag{5}$$

This shows that knowledge of the active set $\mathcal{A}$ determines the primal and dual solutions of the QP (2). We note that $(x, \tilde{\lambda})$ are the solution of the problem of minimizing the objective of the QP subject to the linear equality constraints $\tilde{A}x = \tilde{b}$.

From (5) we see that the solution is a linear function of the data q and b, provided the active set does not change. This implies that the solution is an affine function of the parameter θ, provided the active set does not change.

When $(x, \tilde{\lambda})$ have the values (5) (with $\hat{\lambda} = 0$), they satisfy complementary slackness and dual feasibility. The remaining two optimality conditions, primal feasibility and dual nonnegativity, can be expressed as

$$\begin{bmatrix} \hat{A} & 0 \\ 0 & -I \end{bmatrix} \begin{bmatrix} P & \tilde{A}^T \\ \tilde{A} & 0 \end{bmatrix}^{-1} \begin{bmatrix} -q \\ \tilde{b} \end{bmatrix} \leq \begin{bmatrix} \hat{b} \\ 0 \end{bmatrix}, \tag{6}$$

since $Ax \leq b$ holds (as equality) for rows with $i \in \mathcal{A}$ and $\hat{\lambda} = 0$. The inequality (6) is a set of linear inequalities in the data b and q, and therefore defines a polyhedron. When (b, q) is in this polyhedron, called the critical region associated with the active set $\mathcal{A}$, (x, λ) given by the linear function (5) are primal and dual optimal for the QP (2).

Since compositions of affine functions are affine, it follows that the primal and dual solutions of the QP (2) are (locally) affine functions of θ. Since the inverse image of a polyhedron under an affine mapping is a polyhedron, the values of θ over which this affine function gives the solution is also a polyhedron. Thus the solution map is a piecewise affine function of θ, with the polyhedral regions determined by the active set. By the uniqueness of the solution, it is not difficult to prove that such a map is also continuous across region boundaries [BMDP02]. This continuity property guarantees a certain degree of robustness to numerical errors when evaluating the map.

2.3 Explicit parametric QP solver

The optimality conditions discussed above suggest a naive explicit solver, which can be practical when m is small. We search over all 2^m potential active sets. For each one we compute x and λ via (5), and then check whether (6) holds. If this happens, we have found the solution; if not, the problem is infeasible.

Now we consider the parameter dependence. For each of the 2^m potential active sets, we can express (5) as

$$(x, \tilde{\lambda}) = F\theta + g, \qquad \hat{\lambda} = 0,$$

where

$$F = \begin{bmatrix} P & \tilde{A}^T \\ \tilde{A} & 0 \end{bmatrix}^{-1} \begin{bmatrix} -U \\ \tilde{V} \end{bmatrix}, \qquad g = \begin{bmatrix} P & \tilde{A}^T \\ \tilde{A} & 0 \end{bmatrix}^{-1} \begin{bmatrix} -u \\ \tilde{v} \end{bmatrix},$$

where $\tilde{V}$ and $\tilde{v}$ are the (row) slices of V and v, respectively, corresponding to $\mathcal{A}$. We then express (6) in terms of θ as

$$H\theta \leq j,$$

where

$$H = \begin{bmatrix} \hat{A} & 0 \\ 0 & -I \end{bmatrix} F - \begin{bmatrix} \hat{V} \\ 0 \end{bmatrix}, \qquad j = -\begin{bmatrix} \hat{A} & 0 \\ 0 & -I \end{bmatrix} g + \begin{bmatrix} \hat{v} \\ 0 \end{bmatrix}.$$

For some choices of potential active sets, the inequalities $H\theta \leq j$ (together with $\theta \in \Theta$) are infeasible. We drop these, and consider only the remaining K sets of potential active sets, and label them as $k = 1, \ldots, K$. We can compute the coefficients of the piecewise affine solution map, denoted F_k and g_k, and their associated region, defined by H_k and j_k, before knowing the specific value of θ. Thus we have the explicit solution map

$$(x, \lambda) = F_k\theta + g_k \quad \text{when} \quad H_k\theta \leq j_k, \quad k = 1, \ldots, K. \tag{7}$$

Different existing MPQP solvers differ in how they avoid enumerating all 2^m active sets [AA24, §II-c]. When $\theta \subseteq \Theta$ satisfies none of the inequalities above, the QP is infeasible. The collection of coefficient matrices and vectors F_k, g_k, H_k, j_k, $k = 1, \ldots, K$ gives an explicit representation of the solution map of the parametrized QP (2). Since we can compute these matrices and vectors (and determine the value of K) before we have specified θ, we refer to computing these coefficients as the offline solve. Evaluating (7) for a given value of θ is called the online solve. Note that it involves no division.

The number of coefficients in the explicit solver is around $K(n + m)p$, up to a factor of K larger than the number of coefficients in the original problem, $n^2 + nm + (n + m)p$ (neglecting sparsity). Even though K can grow exponentially with m, it is often practical for small problems [BMDP02, TJB03a, CB17].

Implementation. When we explicitly solve a parametrized QP in practice, some of our initial assumptions can be relaxed. We can handle positive semidefinite P (instead of just positive definite); we directly handle equality constraints; and we do not require LICQ. (When P is not positive definite, the solver provides *a* solution, rather than *the* solution, since the solution need not be unique in this case.)

In the offline phase, implementations do not search all 2^m possible active sets, but instead find nonempty regions one by one. This allows us to handle problems where 2^m is very large, but K, the number of (nonempty) regions, is still moderate.

In the online solve, explicit solvers store the regions in a way that facilitates faster search than a simple linear search over $k = 1, \ldots, K$, typically involving a pre-computed tree. Other methods are used to either reduce the storage or increase the speed of the online evaluations.

We use the specific solver PDAQP [AA24], which has several such accelerations implemented. The offline solve is made more efficient by systematic searching over neighboring regions. The online solve benefits from a binary search tree for the search over regions [TJB03b]. Complete details can be found in

https://github.com/darnstrom/pdaqp.

3 Code generation

3.1 Domain-specific languages for optimization

When solving a problem instance, DSLs perform a sequence of three steps. First, the DSL transforms the user-defined problem into a form accepted by a standard or canonical solver. For example, a constraint like

$$0 \leq x \leq 1$$

for $x \in \mathbf{R}$ is translated to

$$Ax \leq b, \quad A = \begin{bmatrix} 1 \\ -1 \end{bmatrix}, \quad b = \begin{bmatrix} 1 \\ 0 \end{bmatrix},$$

as it appears in a canonical form like (2). In a second step, a canonical solver (like PDAQP) is called to solve the canonical problem. Ultimately, a solution for the user-defined problem is retrieved from the solution returned by the canonical solver.

Typically, these three steps are performed every time a problem instance is solved. We call such systems *parser-solvers*. When dealing with a parametric QP, whose structure does not change between solves, repeated parsing, *i.e.*, discovering how to reduce the problem to canonical form, is unnecessary and usually inefficient.

3.2 Code generation for explicitly solving QPs

Consider an application where we solve many instances of a specific parametric QP, possibly in an embedded system with hard real-time constraints. For such applications, a *code generator* makes more sense.

As illustrated in figure 1, a code generator for explicitly solving QPs takes as input the parametric QP, and generates source code that is tailored for (explicitly) solving instances of that parametric QP. The source code is then compiled into an efficient custom solver, which has a number of benefits compared to parser-solvers. First, by exploiting the parametric structure and caching canonicalization, the compiled solver becomes faster. Second, the compiled solver can be deployed in embedded systems, satisfying rules for safety-critical code [Hol06].

We extend the open-source code generator CVXPYgen [SBD+22] to generate code for explicitly solving QPs. The QP is modeled with CVXPY before CVXPYgen generates library-, allocation-, and division-free code for translating between the user-defined problem

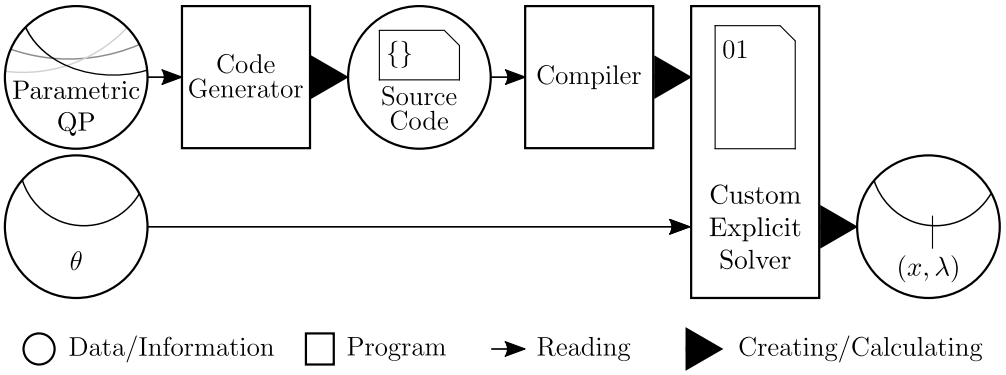

Figure 1: Code generation for explicitly solving a parametric QP.

and a canonical form (that of PDAQP in this case) for explicitly solving QPs. Open source code and full documentation for CVXPYgen and its explicit solve feature is available at

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

Disciplined parametrized programming. We require the QP to be modeled in CVXPY using disciplined parametrized programming (DPP) [AAB+19]. The DPP rules mildly restrict how parameters may enter the objective and constraints. Generally speaking, if parameters enter all expressions in an affine way, then the problem is DPP-compliant. Details on the DPP rules can be found at https://www.cvxpy.org.

Canonicalization. When a problem is modeled according to DPP, parameter canonicalization and solution retrieval are affine mappings,

$$\theta = C\theta^{\text{user}} + c, \qquad x^{\text{user}} = Rx + r,$$

where θ^{user} and x^{user} are the user-defined parameter and variable, respectively. The matrices C and R are typically very sparse. The retrieval matrix R is usually wide, *i.e.*, there are more canonical variables than user-defined variables, since the canonicalization step introduces auxiliary variables [AVDB18]. CVXPYgen generates code for the respective sparse matrix-vector multiplications. In fact, the solution retrieval $Rx + r$ often reduces to simple pointing to memory in C.

Parameter constraints. The user may specify constraints on parameters. Suppose that the values of the user-defined parameter θ^{user} lie in the set Θ^{user}. Then, similarly to how the parameter θ^{user} itself is canonicalized, the parameter constraints are translated to the canonical set of possible parameters Θ.

Limitations. In PDAQP the number of inequality constraints m is limited to 1024 so the regions can be efficiently represented as a bit string. If K, or the size of the data in the explicit solver, exceed a given limit, the offline phase is terminated with a warning.

```
1  import cvxpy as cp
2  from cvxpygen import cpg
3
4  d, p, X, l, u = ...
5  beta = cp.Variable(d, name='beta')
6  v = cp.Variable(name='v')
7  y = cp.Parameter(p, name='y')
8
9  obj = cp.Minimize(cp.sum_squares(X @ beta + v - y))
10 constr = [beta >= 0, l <= y, y <= u]
11 prob = cp.Problem(obj, constr)
12
13 cpg.generate_code(prob, solver='explicit')
```

Figure 2: Generating an explicit solver with CVXPYgen.

4 Hello world

Here we present a simple example to illustrate how explicit solver code generation works. Consider the parametric QP

$$\begin{array}{ll} \text{minimize} & \|X\beta + v\mathbf{1} - y\|_2^2 \\ \text{subject to} & \beta \geq 0, \end{array} \tag{8}$$

where $\beta \in \mathbf{R}^d$ and $v \in \mathbf{R}$ are the variables, $\theta^{\text{user}} = y \in \mathbf{R}^p$ is the parameter with $\Theta^{\text{user}} = \{y \mid l \leq y \leq u\}$, and $\mathbf{1}$ denotes the vector with all entries one. The bounds $l \in \mathbf{R}^p$ and $u \in \mathbf{R}^p$ and the matrix $X \in \mathbf{R}^{p \times d}$ are given data.

4.1 Modeling and code generation

The problem can be formulated in CVXPY as shown in figure 2, up to line 11. Note that in line 10, we specify Θ^{user} with standard CVXPY constraints. We generate the explicit solver in line 13.

4.2 C interface

Figure 3 shows how the generated explicit solver can be used in C. In line 7, the first entry of the parameter y is updated to the value 1.2. The function `cpg_update_y` ensures that the parameter values are within their pre-specified limits (otherwise, it maps the given value back onto Θ). In line 8, the problem is solved explicitly with the `cpg_solve` function. In lines 9 and 10, respectively, the first entry of the optimal coefficients β and the optimal v is read and printed. In line 11, the resulting objective value is calculated and printed. Note that the calculation of the objective value is kept in a separate function from the solve function, for maximal efficiency.

```c
#include <stdio.h>
#include "cpg_workspace.h"
#include "cpg_solve.h"

int main(int argc, char *argv[]){

  cpg_update_y(0, 1.2);
  cpg_solve();
  printf("%f\n", CPG_Result.prim->beta[0]);
  printf("%f\n", CPG_Result.prim->v);
  printf("%f\n", cpg_obj());

  return 0;

}
```

Figure 3: Using the explicit solver.

Figure 4 shows the C structs that store the result. In the example above, we access the primal solution `beta` and `v` via the `prim` field of the result struct `CPG_Result`. The dual variables in `CPG_Dual_t` are named according to the index in the list of CVXPY constraints.

4.3 CVXPY interface

We consider a small instance of the parametric QP (8) with $d = 2$, $p = 3$, the entries of X generated IID from $\mathcal{N}(0, 1)$, $l = 0$, and $u = \mathbf{1}$. We assign the entries of y randomly between 0 and 1 and solve the problem three times: with CVXPY using the iterative OSQP solver [SBG+20], with CVXPYgen using OSQP, and with CVXPYgen using the explicit PDAQP solver.

Figure 5 shows the comparison, demonstrating how the explicit solver can be used via its auto-generated CVXPY interface. Starting in line 15, we show that the primal and dual solutions and the objective values are all close, respectively.

```
typedef struct {
  cpg_float  *beta;     // primal variable beta
  cpg_float   v;        // primal variable v
} CPG_Prim_t;

typedef struct {
  cpg_float  *d0;       // dual variable d0
} CPG_Dual_t;

typedef struct {
  CPG_Prim_t *prim;     // primal solution
  CPG_Dual_t *dual;     // dual solution
} CPG_Result_t;
```

Figure 4: Data structure of explicit solver result.

```python
from code_osqp.cpg_solver import cpg_solve
prob.register_solve('gen_OSQP', cpg_solve)

from code_explicit.cpg_solver import cpg_solve
prob.register_solve('gen_explicit', cpg_solve)

def print_result():
  print(f'v:    {v.value}')
  print(f'beta: {beta.value}')
  print(f'dual: {constr[0].dual_value}')
  print(f'obj:  {obj.value}')

y.value = [0.6, 0.8, 0.2]

prob.solve(solver='OSQP')
print_result()

# v:    0.916741
# beta: [0.000000 0.288547]
# dual: [0.750370 0.000000]
# obj:  0.059628

prob.solve(method='gen_OSQP')
print_result()

# v:    0.916741
# beta: [0.000000 0.288547]
# dual: [0.750370 0.000000]
# obj:  0.059628

prob.solve(method='gen_explicit')
print_result()

# v:    0.916740
# beta: [0.000000 0.288546]
# dual: [0.750370 0.000000]
# obj:  0.059628
```

Figure 5: Using the explicit solver in CVXPY.

	Solve (Python)	Solve (C)	Gen. + compile	Gen.	Binary size
CVXPY	0.6089 ms	–	–	–	–
CVXPYgen OSQP	0.1764 ms	0.1257 ms	5.7 s	0.1 s	80 KB
CVXPYgen explicit	0.0127 ms	0.0004 ms	13.7 s	9.5 s	15 KB

Table 1: Timing and binary sizes for monotone regression problem.

5 Applications

In this section we report timing and code size details for some typical application examples. In each case we compare CVXPYgen using the explicit PDAQP solver to CVXPYgen using the iterative OSQP solver and standard CVXPY using OSQP. When using CVXPYgen with the OSQP solver, we use OSQP's code generation feature, which caches the factorization of the KKT system, for accelerated solving [BSM+17]. When using OSQP in CVXPY or CVXPYgen, we set both the relative and absolute tolerances to 10^{-4} (the default in CVXPY). We run the experiments on an Apple M1 Pro, compiling with Clang at optimization level 3. For iterative and explicit code generation, we report solve times in C, and the overall time when solving from Python via the auto-generated CVXPY interface. We also give the time it takes to generate and compile the code.

5.1 Monotone regression

We consider the monotone regression problem

$$\begin{array}{ll} \text{minimize} & \|Ax - b\|_2^2 \\ \text{subject to} & x_1 \leq x_2 \leq \cdots \leq x_d, \end{array}$$

where $x \in \mathbf{R}^d$ is the variable and $\theta^{\text{user}} = b \in \mathbf{R}^q$ is the parameter, with $\Theta^{\text{user}} = [-1, 1]^q$. The matrix $A \in \mathbf{R}^{q \times d}$ is given. (This is called monotone regression since the components x_i are constrained to be monotonically nondecreasing.)

Problem instances. We consider $d = 5$ and $q = 10$. The entries of A are generated IID from $\mathcal{N}(0, 1)$, and we generate 100 problem instances where b is sampled uniformly from Θ^{user}.

Results. We obtain $n = 15$ variables, $m = d-1 = 4$ inequality constraints, and $p = q = 10$ parameters. (In this case, the canonicalization step introduced $n-d = 10$ auxiliary variables.) We find $K = 16$ regions (which equals 2^m, the maximum possible number of active sets). Table 1 shows the average solve times and binary sizes.

5.2 Power management

Problem. A nonnegative electric power load L is served by a PV (photovoltaic solar panel) system, a storage battery, and a grid connection [BNC+17, NOBL25]. We denote the solar

	Solve (Python)	Solve (C)	Gen. + compile	Gen.	Binary size
CVXPY	0.6564 ms	–	–	–	–
CVXPYgen OSQP	0.1261 ms	0.0541 ms	5.1 s	0.1 s	75 KB
CVXPYgen explicit	0.0207 ms	0.0001 ms	12.6 s	8.7 s	10 KB

Table 2: Timing and binary sizes for the power management problem.

power as s, the battery power as b, and the grid power as g. These three power sources supply the load, so we have

$$L = s + b + g.$$

The PV power satisfies $0 \leq s \leq S$, where $S \geq 0$ is the available PV power. The battery power satisfies $-C \leq b \leq D$, where $D > 0$ is the maximum possible discharge power and $C > 0$ is the maximum possible charge power. The grid power satisfies $g \geq 0$, *i.e.*, we cannot sell power back to the grid. The (positive) price of the grid power is P, so the grid cost is Pgh, where h is the duration of one time period, over which we hold the power values constant.

The battery state of charge at the beginning of the time period is denoted q, and satisfies $0 \leq q \leq Q$, where Q is the battery capacity. At the beginning of the next time period the battery charge is $q^+ = q - hb$. We must have $0 \leq q^+ \leq Q$.

We take the cost function

$$Pgh + \alpha(q^+ - q^{\text{tar}})^2 + \beta b^2,$$

where α and β are given and positive, and q^{tar} is a given target battery charge value.

To choose the powers we solve the QP

$$\begin{array}{ll} \text{minimize} & Pgh + \alpha(q^+ - q^{\text{tar}})^2 + \beta b^2 \\ \text{subject to} & L = s + b + g, \\ & 0 \leq s \leq S, \quad -C \leq b \leq D, \quad g \geq 0, \\ & q^+ = q - hb, \quad 0 \leq q^+ \leq Q, \end{array}$$

where s, b, g, and q^+ are the variables, and $\theta^{\text{user}} = (L, S, P, q)$ are parameters. The remaining constants, C, D, h, Q, q^{tar}, α, and β, are known. We take

$$\Theta^{\text{user}} = [0, 1] \times [0, 0.5] \times [1, 2] \times [0, Q].$$

Problem instances. We set $C = D = 1$, $h = 0.05$, $Q = 1$, $q^{\text{tar}} = 0.5$, and $\alpha = \beta = 0.1$. We generate 100 problem instances where (L, S, P, q) is sampled uniformly from Θ^{user}.

Results. After canonicalization, there are $n = 5$ variables (including one auxiliary variable), $m = 7$ inequality constraints, and $p = 4$ parameters. We find $K = 5$ regions. Table 2 shows the average solve times and binary sizes.

	Solve (Python)	Solve (C)	Gen. + compile	Gen.	Binary size
CVXPY	1.102 ms	–	–	–	–
CVXPYgen OSQP	0.875 ms	0.790 ms	5.2 s	0.1 s	110 KB
CVXPYgen explicit	0.025 ms	0.001 ms	13.7 s	9.1 s	93 KB

Table 3: Timing and binary sizes for the model predictive control problem.

5.3 Model predictive control

We consider the linear dynamical system [BB91]

$$z_{t+1} = Az_t + Bu_t, \quad t = 0, 1, \ldots, H-1,$$

where $z_t \in \mathbf{R}^{n_z}$ is the state and $u_t \in \mathbf{R}^{n_u}$ is the input, which must satisfy $\|u_t\|_\infty \leq 1$. The matrices $A \in \mathbf{R}^{n_z \times n_z}$ and $B \in \mathbf{R}^{n_z \times n_u}$ are given. We solve the model predictive control problem [GPM89, KC16]

$$\begin{array}{ll} \text{minimize} & z_H^T P z_H + \sum_{t=0}^{H-1} \left(z_t^T Q z_t + u_t^T R u_t\right) \\ \text{subject to} & z_{t+1} = Az_t + Bu_t, \quad t = 0, \ldots, H-1, \\ & \|u_t\|_\infty \leq 1, \quad t = 0, \ldots, H-1, \\ & z_0 = z^{\text{init}}, \end{array}$$

where $z_0, \ldots, z_H$ and $u_0, \ldots, u_{H-1}$ are the variables and $\theta^{\text{user}} = z^{\text{init}}$ is the parameter. We take $\Theta^{\text{user}} = [-1, 1]^{n_z}$. The objective matrices $P \in \mathbf{S}^{n_z}_{++}$, $Q \in \mathbf{S}^{n_z}_{++}$, and $R \in \mathbf{S}^{n_u}_{++}$ (along with A and B) are given.

Problem instances. We consider $n_z = 6$ states, $n_u = 1$ input, and a horizon length of $H = 5$. We construct A by sampling its diagonal entries from $\mathcal{N}(0, 1)$ and its off-diagonal entries IID from $\mathcal{N}(0, 0.01)$, before scaling the whole matrix such that A has spectral radius 1. The entries of the input matrix B are sampled IID from $\mathcal{N}(0, 0.001)$. We set the controller weights to $Q = I$ and $R = 0.1I$, and compute P as the solution to the algebraic Riccati equation associated with the infinite-horizon problem [KS72]. We generate 100 problem instances where the entries of z^{init} are generated uniformly from Θ^{user}.

Results. After canonicalization, we have $n = 77$ variables (of which 36 are auxiliary variables), $m = 10$ inequality constraints, and $p = n_z = 6$ parameters. We find $K = 63$ regions. Table 3 shows the average solve times and binary sizes.

5.4 Portfolio optimization

We construct a financial portfolio consisting of holdings in N assets [GK00, Nar13, Pal25]. We represent the holdings relative to the total (positive) portfolio value, in terms of non-negative weights $w \in \mathbf{R}^N_+$, where $\mathbf{1}^T w = 1$, with w_i being the fraction of the (positive) total portfolio value invested in asset i. With estimated mean annualized asset returns $\mu \in \mathbf{R}^N$

Ticker symbol	Company
AAPL	Apple
AMZN	Amazon
BRK.A	Berkshire Hathaway
FB (now META)	Facebook (now Meta)
GOOGL	Alphabet
MSFT	Microsoft
XOM	ExxonMobil

Table 4: Stocks used in the portfolio optimization problem.

	Solve (Python)	Solve (C)	Gen. + compile	Gen.	Binary size
CVXPY	0.5441 ms	–	–	–	–
CVXPYgen OSQP	0.0502 ms	0.0070 ms	5.1 s	0.1 s	76 KB
CVXPYgen explicit	0.0113 ms	0.0005 ms	20.8 s	16.5 s	234 KB

Table 5: Timing and binary sizes for the portfolio optimization problem.

[GK00], the estimated mean annualized portfolio return is $\mu^T w$. The variance or risk of the portfolio return is $w^T \Sigma w$, where $\Sigma \in \mathbf{S}^N_{++}$ is an estimate for the covariance matrix of the annualized asset returns. Our objective is to maximize the risk-adjusted expected annualized return

$$\mu^T w - \gamma w^T \Sigma w,$$

where $\gamma > 0$ is the risk-aversion factor. To find the portfolio we solve the Markowitz problem [Mar52, BJK+24]

$$\begin{array}{ll} \text{maximize} & \mu^T w - \gamma w^T \Sigma w \\ \text{subject to} & \mathbf{1}^T w = 1, \quad w \geq 0, \end{array}$$

where the portfolio weights $w \in \mathbf{R}^N$ are the variable and the parameter is $\theta^{\text{user}} = \mu$ with $\Theta^{\text{user}} = [-1, 1]^N$. This means that we expect no annualized returns beyond $\pm 100\%$. The covariance matrix $\Sigma \in \mathbf{S}^N_{++}$ and the risk-aversion factor $\gamma > 0$ are given.

Problem instances. We take $N = 7$ assets. To obtain data we choose the 7 stocks with the largest market capitalization as of January 1, 2017, listed in table 4. We compute Σ by first taking the sample covariance of the 7 assets' daily returns in the years 2017 and 2018, and then annualizing the result. We choose $\gamma = 2$. We generate 250 problem instances where μ is taken as the one-year trailing average of returns for 250 trading days in the year 2019.

Results. We have $n = N = 7$ variables, $m = N = 7$ inequality constraints, and $p = N = 7$ parameters. We find $K = 127$ regions, only one less than the maximum 2^m potential combinations of investing in an asset or not, since the constraint $\mathbf{1}^T w = 1$ prevents $w = 0$ (not investing in any asset). Table 5 shows the average solve times and binary sizes.

6 Conclusions

We have added new functionality to the code generator CVXPYgen that generates an explicit solver (in C) for a parametrized convex optimization problem, when that is tractable. The user can prototype a problem in CVXPY, with code close to the math and convenient names for multiple variables and parameters, using a generic iterative solver; a change of one option in code generation will generate an explicit solver for the parametrized problem. For typical (small) problems from various application domains, our numerical experiments show the generated explicit solvers exhibit solve times at (or below) one microsecond, giving up to three orders of magnitude speedup over an iterative solver.

References

[AA24] D. Arnström and D. Axehill. A high-performant multi-parametric quadratic programming solver. In *2024 IEEE 63rd Conference on Decision and Control (CDC)*, pages 303–308, 2024.

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

[ABA24] D. Arnström, D. Broman, and D. Axehill. Exact worst-case execution-time analysis for implicit model predictive control. *IEEE Transactions on Automatic Control*, 69(10):7190–7196, 2024.

[AVDB18] A. Agrawal, R. Verschueren, S. Diamond, and S. Boyd. A rewriting system for convex optimization problems. *Journal of Control and Decision*, 5(1):42–60, 2018.

[BB91] S. Boyd and C. Barratt. *Linear controller design: Limits of performance*, volume 78. Citeseer, 1991.

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

[BBM+02] A. Bemporad, F. Borrelli, M. Morari, et al. Model predictive control based on linear programming – the explicit solution. *IEEE transactions on automatic control*, 47(12):1974–1985, 2002.

[BBM03] F. Borrelli, A. Bemporad, and M. Morari. Geometric algorithm for multiparametric linear programming. *Journal of optimization theory and applications*, 118:515–540, 2003.

[BBM17] F. Borrelli, A. Bemporad, and M. Morari. *Predictive control for linear and hybrid systems*. Cambridge University Press, 2017.

[Bem04] A. Bemporad. Hybrid Toolbox - User's Guide, 2004. http://cse.lab.imtlucca.it/~bemporad/hybrid/toolbox.

[Bem15] A. Bemporad. A multiparametric quadratic programming algorithm with polyhedral computations based on nonnegative least squares. *IEEE Transactions on Automatic Control*, 60(11):2892–2903, 2015.

[Ber99] D. Bertsekas. *Nonlinear Programming*. Athena Scientific, 2nd edition, 1999.

[BF03] A. Bemporad and C. Filippi. Suboptimal explicit receding horizon control via approximate multiparametric quadratic programming. *Journal of optimization theory and applications*, 117:9–38, 2003.

[BF06] A. Bemporad and C. Filippi. An algorithm for approximate multiparametric convex programming. *Computational optimization and applications*, 35:87–108, 2006.

[BJK+24] S. Boyd, K. Johansson, R. Kahn, P. Schiele, and T. Schmelzer. Markowitz portfolio construction at seventy. *Journal of Portfolio Management*, 50(8):117–160, 2024. Also available at https://arxiv.org/pdf/2401.05080.

[Bla16] L. Blackmore. Autonomous precision landing of space rockets. In *Frontiers of Engineering: Reports on Leading-Edge Engineering from the 2016 Symposium*, volume 46, pages 15–20. The Bridge Washington, DC, 2016.

[BM07] A. Bemporad and M. Morari. Robust model predictive control: A survey. In *Robustness in identification and control*, pages 207–226. Springer, 2007.

[BMDP02] A. Bemporad, M. Morari, V. Dua, and E. Pistikopoulos. The explicit linear quadratic regulator for constrained systems. *Automatica*, 38(1):3–20, 2002.

[BNC+17] R. Byrne, T. Nguyen, D. Copp, B. Chalamala, and I. Gyuk. Energy management and optimization methods for grid energy storage systems. *IEEE Access*, 6:13231–13260, 2017.

[BOPS11] A. Bemporad, A. Oliveri, T. Poggi, and M. Storace. Ultra-fast stabilizing model predictive control via canonical piecewise affine approximations. *IEEE Transactions on Automatic Control*, 56(12):2883–2897, 2011.

[BSM+17] G. Banjac, B. Stellato, N. Moehle, P. Goulart, A. Bemporad, and S. Boyd. Embedded code generation using the OSQP solver. In *IEEE Conference on Decision and Control*, pages 1906–1911. IEEE, 2017.

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

[BV18] S. Boyd and L. Vandenberghe. *Introduction to Applied Linear Algebra: Vectors, Matrices, and Least Squares*. Cambridge University Press, 2018.

[CA25] G. Chari and B. Açıkmeşe. QOCO: A quadratic objective conic optimizer with custom solver generation. *arXiv preprint arXiv:2503.12658*, 2025.

[CB17] G. Cimini and A. Bemporad. Exact complexity certification of active-set methods for quadratic programming. *IEEE Transactions on Automatic Control*, 62(12):6094–6109, 2017.

[CP11] P. Combettes and J. Pesquet. Proximal splitting methods in signal processing. *Fixed-point algorithms for inverse problems in science and engineering*, pages 185–212, 2011.

[CP16] A. Chambolle and T. Pock. An introduction to continuous optimization for imaging. *Acta Numerica*, 25:161–319, 2016.

[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 (ECC)*, pages 3071–3076. IEEE, 2013.

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

[DJ14] A. Domahidi and J. Jerez. FORCES Professional. Embotech GmbH, 2014.

[Fia83] A. Fiacco. *Introduction to Sensitivity and Stability Analysis in Nonlinear Programming*. Academic Press, London, U.K., 1983.

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

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

[GBN11] A. Gupta, S. Bhartiya, and P. Nataraj. A novel approach to multiparametric quadratic programming. *Automatica*, 47(9):2112–2117, 2011.

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

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

[GMW19] P. Gill, W. Murray, and M. Wright. *Practical optimization*. SIAM, 2019.

[GN72] T. Gal and J. Nedoma. Multiparametric linear programming. *Management Science*, 18(7):406–422, 1972.

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

[HKJM13] M. Herceg, M. Kvasnica, C. N. Jones, and M. Morari. Multi-parametric toolbox 3.0. *European Control Conference (ECC)*, pages 502–510, 2013.

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

[JG03] T. Johansen and A. Grancharova. Approximate explicit constrained linear model predictive control via orthogonal search tree. *IEEE Transactions on Automatic Control*, 58(5):810–815, 2003.

[JJST06] T. Johansen, W. Jackson, R. Schreiber, and P. Tondel. Hardware synthesis of explicit model predictive controllers. *IEEE Transactions on control systems technology*, 15(1):191–197, 2006.

[JM06] C. Jones and M. Morrari. Multiparametric linear complementarity problems. In *Proceedings of the 45th IEEE Conference on Decision and Control*, pages 5687–5692. IEEE, 2006.

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

[KC16] B. Kouvaritakis and M. Cannon. *Model Predictive Control*. Springer, 2016.

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

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

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

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

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

[MR64] O. Mangasarian and J. Rosen. Inequalities for stochastic nonlinear programming problems. *Operations Research*, 12:143–154, 1964.

[Nar13] R. Narang. *Inside the Black Box: A Simple Guide to Quantitative and High-frequency Trading*. John Wiley & Sons, 2013.

[NOBL25] O. Nnorom, G. Ogut, S. Boyd, and P. Levis. Aging-aware battery control via convex optimization. *arXiv preprint arXiv:2505.09030*, 2025.

[NW06] J. Nocedal and S. Wright. *Numerical Optimization*. Springer, 2nd edition, 2006.

[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.

[ODP+16] R. Oberdieck, N. Diangelakis, M. Papathanasiou, I. Nascu, and E. Pistikopoulos. POP – Parametric optimization toolbox. *Industrial & Engineering Chemistry Research*, 55(33):8979–8991, 2016.

[Pal25] D. Palomar. *Portfolio Optimization: Theory and Application*. Cambridge University Press, 2025.

[PS10] P. Patrinos and H. Sarimveis. A new algorithm for solving convex parametric quadratic programs based on graphical derivatives of solution mappings. *Automatica*, 46(9):1405–1418, 2010.

[RMD+17] J. Rawlings, D. Mayne, M. Diehl, et al. *Model Predictive Control: Theory, Computation, and Design*. Nob Hill Publishing Madison, WI, 2017.

[ROL+23] A. Ravera, A. Oliveri, M. Lodi, A. Bemporad, W. Heemels, E. Kerrigan, and M. Storace. Co-design of a controller and its digital implementation: The MOBY-DIC2 toolbox for embedded model predictive control. *IEEE Transactions on Control Systems Technology*, 31(6):2871–2878, 2023.

[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.

[TJB03a] P. Tøndel, T. Johansen, and A. Bemporad. An algorithm for multi-parametric quadratic programming and explicit MPC solutions. *Automatica*, 39(3):489–497, 2003.

[TJB03b] P. Tøndel, T. Johansen, and A. Bemporad. Evaluation of piecewise affine control via binary search tree. *Automatica*, 39(5):945–950, 2003.

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

[VFK+21] 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*, pages 1–37, 2021.

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

[ZE10] M. Zibulevsky and M. Elad. L_1-L_2 optimization in signal and image processing. *IEEE Signal Processing Magazine*, 27(3):76–88, 2010.