CVXGEN: A Code Generator for Embedded Convex Optimization

Jacob Mattingley and Stephen Boyd

November 3, 2010

Abstract

CVXGEN is a software tool that takes a high level description of a convex optimization problem family, and automatically generates custom C code that compiles into a reliable, high speed solver for the problem family. The current implementation targets problem families that can be transformed, using disciplined convex programming techniques, to convex quadratic programs of modest size. CVXGEN generates simple, flat, library-free code suitable for embedding in real-time applications. The generated code is almost branch free, and so has highly predictable run-time behavior. The combination of regularization (both static and dynamic) and iterative refinement in the search direction computation yields reliable performance, even with poor quality data. In this paper we describe how CVXGEN is implemented, and give some results on the speed and reliability of the automatically generated solvers.

Contents

1 Introduction **3**
1.1 Embedded convex optimization . . . 3
1.2 Prior work . . . 4
1.3 Overview . . . 5

2 CVXGEN example **5**

3 Problem specification **8**
3.1 Symbols . . . 8
3.2 Functions and expressions . . . 9
3.3 Convexity . . . 9
3.4 Objective and constraints . . . 10

4 Using CVXGEN **11**
4.1 Generated files . . . 11
4.2 Using the generated code . . . 11
4.3 Solver settings . . . 12
4.4 Handling infeasibility and unboundedness . . . 12
4.5 Increasing solver speed . . . 13

5 Implementation **13**
5.1 Parsing and canonicalization . . . 14
5.2 Solving the standard-form QP . . . 14
5.3 Solving the KKT system . . . 17
5.3.1 Regularization . . . 17
5.3.2 Iterative refinement . . . 18
5.3.3 Dynamic regularization . . . 19
5.3.4 Choosing a permutation . . . 19
5.4 Code generation . . . 20
5.4.1 Templating language . . . 21
5.4.2 Explicit coding style . . . 22

6 Numerical examples **22**
6.1 Simple quadratic program . . . 22
6.2 Support vector machine . . . 22
6.3 Lasso . . . 24
6.4 Model predictive control . . . 24
6.5 Settings and reliability . . . 26

7 Conclusion **29**

1 Introduction

Convex optimization is widely used, since convex optimization problems can be solved reliably and efficiently, with both useful theoretical performance guarantees, and well-developed, practical methods and tools [BV04, NN94, Ye97, NW99]. Current application areas include control [BB91, BEFB94, DDB95], circuit design [HBL01, HMBL99, BKPH05], economics and finance [Mar52, CT07], networking [KMT98, WJLH06], statistics and machine learning [Vap00, CST00], quantum information theory [EMV03], combinatorial optimization [GGL96] and signal processing [Spe07, CRR69].

Parser-solvers like CVX [GB08a] and YALMIP [L04] make the process of specifying and solving a convex problem simple, and so are ideal for prototyping an algorithm or method that relies on convex optimization. But the resulting solve times are measured in seconds or minutes, which precludes their use in faster real-time systems. In addition, these tools require extensive libraries and commercial software to run, and so are not suitable for embedding in many applications. Conventionally, however, moving from a general purpose parser-solver to a high-speed, embeddable solver requires extensive modeling and conversion by hand. This is a time consuming process that requires significant expertise, so embedded convex optimization applications have so far been limited.

This paper describes the capabilities and implementation of CVXGEN, a code generator for convex optimization problem families that can be reduced to quadratic programs (QPs). CVXGEN takes a high level description of a convex optimization problem family, and automatically generates flat, library-free C code that can be compiled into a high speed custom solver for the problem family. For small and medium sized problems (with up to hundreds of optimization variables), CVXGEN generates solvers with solve times measured in microseconds or milliseconds. These solvers can therefore be embedded in applications requiring hundreds (or thousands) of solves per second. The generated solvers are very reliable and robust, gracefully handling even relatively poor quality data (such as redundant constraints).

1.1 Embedded convex optimization

The setting we will consider here is embedded convex optimization, where finding the solution to convex optimization problems is part of a wider algorithm. This imposes special requirements on the solver. First, the solver must be robust. While it may be acceptable for a general purpose solver to occasionally fail, since a human can readily intervene, this is not acceptable for a solver running automatically. In particular, the solver should never cause a 'fatal error' such as a division-by-zero or segmentation fault that could crash the entire on-line algorithm. This should apply even in the presence of poor quality data.

The solver must also be fast. Depending on the sample rate of the system, the time available to solve each optimization problem may be very small, with operation speeds measured in Hz or kHz. This requires solve times measured in milliseconds or microseconds. In particular, the maximum solve time should be known ahead of time.

Finally, the solver should have a simple footprint. General purpose parser-solvers usually

depend on either an integrated environment like Matlab, or, at least, extensive pre-built libraries. This makes it difficult to adapt and validate the solver for use in embedded applications. Instead, the solver should ideally use simple, flat code with minimal, or no, library dependencies.

These requirements are particularly important for embedded applications, but can also benefit off-line applications. For example, with a high-speed solver, Monte Carlo simulation can be carried out many times faster than real-time. This is particularly useful for verifying algorithm performance on historic or simulated data.

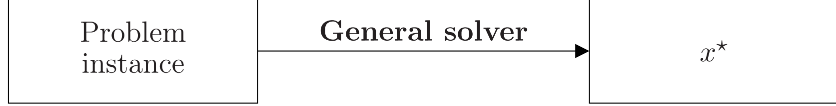
While embedded solvers have certain requirements, they also have certain features that can be exploited to make their design less challenging. Embedded solvers often require only limited accuracy. This allows early termination and makes numerical scaling problems less likely. As an example, with model predictive control (MPC), even very low accuracy can result in acceptable control performance [WB08]. Another difference is that, for an embedded solver, the *problem family* (*i.e.*, the problem statement, dimensions and sparsity) remains constant with each solution. Each solver will perform many solves with different *problem instances* (*i.e.*, the fully specified optimization problem, including data).

This means we have the opportunity to spend considerable time preparing the solver at development time, making use of the (known) exact size and structure of the problem to reduce the (later) solve time. To make this important point clear, consider first figure 1(a), which shows how a general purpose parser-solver works. The parser-solver is always called with both the problem structure and data as part of the problem instance, so if used repeatedly, it must also repeat all preparation and transformations before producing the optimal point $x^\star$. By contrast, consider figure 1(b), which shows how a code generator works. The code generator creates source code from a problem family description, which is then compiled into a custom embedded solver. Rather than needing to process the problem structure separately for every instance, the data are put into the embedded solver directly, which then produces an optimal point $x^\star$.

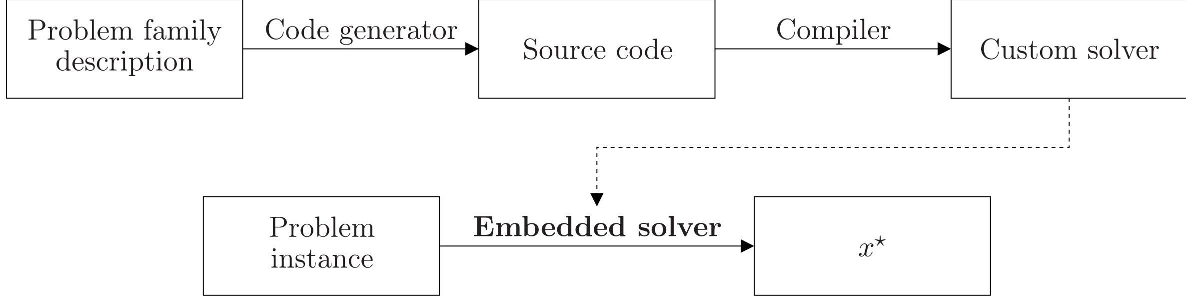
1.2 Prior work

The idea of automatic code generation is rather old, and has been used since (at least) the 1970s for parser-generators like Yacc [Joh75]. Domain-specific code generators are prevalent also; see, for example, [Kan93, Bac96, Bac97, SB04]. The authors' previous papers discuss various current applications of real-time optimization [MB09a], especially in control systems [MWB10] and signal processing [MB09b]; however, we do not know of any previous general purpose convex optimization code generators.

CVXGEN was originally part of CVXMOD [MB08], a general purpose convex optimization parser-solver for Python. Rudimentary code generation capability was added to CVXMOD, but this functionality was soon moved into the separate (Ruby) project CVXGEN. We mention this since previous papers refer to CVXMOD as a code generator [MB09a, MB09b]; however, we consider CVXMOD to be an early prototype of CVXGEN.



(a) A general purpose parser-solver turns a single problem instance into a single optimal point.



(b) An automatic code generator produces code for an embedded solver, which is then used to provide the optimal points for many different problem instances.

Figure 1: Operation of a parser-solver, and a code generator for embedded solvers.

1.3 Overview

The remainder of this paper discusses the use and implementation of CVXGEN. In §2, we give an example showing how CVXGEN looks to the user. In §3, we describe the CVXGEN specification language, which is used to describe each problem family. In §4, we show how the generated solver code is embedded in an application. The CVXGEN implementation requires parsing and conversion to a standard form QP, discussed in §5.1. Solving the standard form QP is covered in §5.2 and §5.3, and the code generation process itself is described in §5.4. Finally, we report on speed and reliability by showing several examples in §6.

2 CVXGEN example

In this section, we will look at a simple example that uses CVXGEN for an embedded convex optimization problem arising in multi-period trading. (For similar applications, see [BV04, §4.4.1 and §4.6.3].)

Multi-period trading example. First we describe the overall application setting (but not in detail). We let $x_t \in \mathbf{R}^n$ denote the vector of asset holdings in time period t, with $(x_t)_i$ denoting the dollar value of asset i. (Negative values of $(x_t)_i$ represent short positions.) We let $u_t \in \mathbf{R}^n$ denote the vector of trades executed at the beginning of investment period t; this results in the post-trade portfolio vector $z_t = x_t + u_t$. The post-trade portfolio is invested for the period, and results in a portfolio at the next period given by

$$x_{t+1} = r_t \circ z_t,$$

where $r_t \in \mathbf{R}_+^n$ is the vector of (total) returns for the assets, and $\circ$ is the Hadamard (elementwise) product. (The return vector r_t is unknown when the trade vector u_t is chosen.)

The trades must be self-financing, including transaction costs. Using linear transaction costs, this constraint can be expressed as

$$\mathbf{1}^T u_t + b_t^T (u_t)_+ + s_t^T (u_t)_- \leq 0,$$

where $b_t \in \mathbf{R}_+^n$ ($s_t \in \mathbf{R}_+^n$) is the vector of buying (selling) transaction cost rates in period t, $(\cdot)_+$ ($(\cdot)_-$) denotes the nonnegative (nonpositive) part of a number, and $\mathbf{1}$ denotes the vector with all entries one. The first term, $\mathbf{1}^T u_t$, denotes the total cash required to carry out the trades given by u_t, not including transaction costs. The second and third terms are the (nonnegative) total buying and selling transaction costs, respectively.

We also have a limit on the leverage of the post-trade portfolio, expressed as

$$\mathbf{1}^T (z_t)_- \leq \eta \mathbf{1}^T z_t.$$

This limits the total post-trade short position (the left-hand side) to be no more than a fraction $\eta \geq 0$ of the total post-trade portfolio value.

The *trading policy*, *i.e.*, how u_t is chosen as a function of data known at time period t, will be based on solving the optimization problem

$$\begin{array}{ll} \text{maximize} & q_t^T z_t - z_t^T Q_t z_t \\ \text{subject to} & z_t = x_t + u_t \\ & \mathbf{1}^T u_t + b_t^T (u_t)_+ + s_t^T (u_t)_- \leq 0 \\ & \mathbf{1}^T (z_t)_- \leq \eta \mathbf{1}^T z_t, \end{array} \tag{1}$$

with variables $u_t \in \mathbf{R}^n$, $z_t \in \mathbf{R}^n$, and parameters (problem data)

$$q_t, \quad Q_t, \quad x_t, \quad b_t, \quad s_t, \quad \eta. \tag{2}$$

Here $q_t \in \mathbf{R}^n$ is an estimate or prediction of the return r_t, available at time t, and $Q_t \in \mathbf{S}_+^n$ (with $\mathbf{S}_+^n$ denoting the cone of $n \times n$ positive semidefinite matrices) encodes uncertainty in return, *i.e.*, risk, along with an appropriate scaling factor. (Traditionally these parameters are the mean and a scaled variance of r_t, respectively. But here we consider them simply parameters used to shape the trading policy.)

Our trading policy works as follows. In period t, we obtain the problem parameters (2). Some of these are known (such as the current portfolio x_t); others are specified or chosen (such as η); and others are estimated by an auxiliary algorithm (q_t, Q_t, b_t, s_t). We then solve the problem (1), which is feasible provided $\mathbf{1}^T x_t \geq 0$. (If this is not the case, we declare ruin and quit.) We then choose the trade u_t as a solution of (1). By construction, this trade will be self-financing, and will respect the post-trade leverage constraint. (The solution can be $u_t = 0$, meaning that no trades should be executed in period t.)

CVXGEN specification. The CVXGEN specification of this problem family is shown in figure 2, for the case with $n = 20$ assets. The specification is explored in detail in §3, but for now we point out the obvious correspondence between the optimization problem (1) and the CVXGEN description in figure 2.

```
dimensions
  n = 20
end

parameters
  q_t (n);  Q_t (n,n) symmetric psd
  b_t (n) nonnegative;  s_t (n) nonnegative
  eta nonnegative
  x_t (n)
end

variables
  u_t (n);  z_t (n)
end

maximize
  q_t'*z_t - quad(z_t, Q_t)
subject to
  z_t == x_t + u_t
  sum(u_t) + b_t'*pos(u_t) + s_t'*neg(u_t) <= 0
  sum(neg(z_t)) <= eta*sum(z_t)
end
```

Figure 2: CVXGEN problem specification for the multi-period trading problem example.

For this problem family, code generation takes 24 s, and the generated code requires 7.9 s to compile. The generated solver solves instances of the problem (1) in 200 μs. For comparison, instances of the problem (1) require around 600 ms to solve using CVX, so code generation yields a speed-up of around 3000×.

Even if the actual trading application does not require a 200 μs solve time, a fast solver is still very useful. For example, to test the performance of our trading policy (together with the auxiliary algorithm that provides the parameter estimates in each period), we would need to solve, sequentially, many instances of the problem (1). Simulating or testing the trading policy for one year, with trading every 10 minutes (say) and around 2000 hours of trading per year, requires solving 12000 instances of the problem (1) (sequentially, so it cannot be done in parallel). Using CVX, the trading policy simulation time would be around two hours (not counting the time required to produce the parameter estimates). Using the CVXGEN-generated solver for this same problem, on the same computer, the year-long simulation can be carried out in a few seconds. The speed-up is important, since we may need to simulate the trading policy many times as we adjust the parameters or develop the auxiliary prediction algorithm.

3 Problem specification

Here we describe CVXGEN's problem specification language. The language is built on the principles of disciplined convex programming (DCP) [Gra04, GBY06, GB08b]. By imposing several simple rules on the problem specification, we ensure that valid problem statements represent convex problems, which can be transformed to canonical form in a straightforward and automatic way. Figure 2 shows the CVXGEN problem specification for problem (1).

3.1 Symbols

Dimensions. The first part of the problem specification shows the numeric dimensions of each of the problem's parameters and variables. This highlights an important point: The numeric size of each parameter and variable must be specified at code generation time, and cannot be left symbolic.

Parameters. Parameters are placeholders for problem data, which are not specified numerically until solve time. Parameters are used to describe problem families; the actual parameter values, specified when the solver is called, define each problem instance. Parameter values are specified with a name and dimensions, and include optional attributes, which are used for DCP convexity verification. Available attributes are `nonnegative`, `nonpositive`, `psd`, `nsd` and `symmetric` and `diagonal`. All except `diagonal` are used for convexity verification; `diagonal` is used to specify the sparsity structure.

Variables. The third block shows optimization variables, which are to be found during the solve phase, *i.e.*, when the solver is called. Variables are also specified with a name and

dimension, and optional attributes.

3.2 Functions and expressions

Expressions are created from parameters and variables using addition, subtraction, multiplication, division and several additional functions. These expressions can then be used in the objective and constraints. The example in figure 2 shows basic matrix and vector multiplication and addition, transposition, and the use of several different functions. Expressions may also be created with scalar division (although with no optimization variables in the denominator) and vector indexing.

CVXGEN comes with a small set of functions that can be composed to create problem descriptions, when supported by the relevant convex calculus (see §3.3). There are two sets of functions provided by CVXGEN. The first set may be used in the objective and constraints, and consists of elementwise absolute value (`abs`), vector and elementwise maximum and minimum (`max` and `min`), ℓ_1 and ℓ_∞ norms (`norm_1` and `norm_inf`), vector summation (`sum`) and elementwise positive part and negative part (`pos` and `neg`). The second set of functions consists of the quadratic and square functions. These can only be used in the objective. This is necessary so that the problem can be transformed to a QP.

3.3 Convexity

CVXGEN library functions. Functions (or operators) in the CVXGEN library are marked for curvature (affine, convex, concave), sign (nonnegative, nonpositive or unknown), and monotonicity (nondecreasing, nonincreasing, or unknown). Affine means both convex and concave. Function monotonicity sometimes depends on the sign of the function arguments; for example, `square` is marked as nonincreasing only if its argument is nonnegative. Here are some examples of CVXGEN functions:

- The sum function is affine and nondecreasing. It is nonnegative when its arguments are nonnegative, and nonpositive when its arguments are nonpositive.
- The square function is convex and nonnegative. It is nondecreasing for nonnegative arguments, and nonincreasing for nonpositive arguments.
- Negation is affine and nonincreasing. It is nonpositive when its argument is nonnegative, and nonnegative when its argument is nonpositive.

CVXGEN expressions. CVXGEN expressions are created from literal constants, parameters, variables, and functions from the CVXGEN library. Expressions are allowed only when CVXGEN can guarantee that the expression is convex, concave, or affine from these attributes. The composition rules used by CVXGEN, which are are similar to those used in CVX [GB08a], are given below, where we use terms like 'affine', 'convex', and 'nonnegative', to mean 'verified by CVXGEN to be affine' (or convex or nonnegative).

- A constant expression is one of:
  - A literal constant.
  - A parameter.
  - A function of constant expressions.
- An affine expression is one of:
  - A constant expression.
  - An optimization variable.
  - An affine function of affine expressions.
- A convex expression is one of:
  - An affine expression.
  - A convex function of an affine expression.
  - A convex nondecreasing function of a convex expression.
  - A convex function, nondecreasing when its argument is nonnegative, of a convex nonnegative expression.
  - A convex nonincreasing function of a concave expression.
  - A concave function, nondecreasing when its argument is nonnegative, of a convex nonnegative expression.
- (An analogous set of rules for concave expressions.)

The calculus of signs is obvious, so we omit it. This set of rules is not minimal: Note, for example, that the rules for affine expressions may be derived by recognizing that affine means both convex and concave.

As an example, consider the expression `p*abs(2*x + 1) - q*square(x + r)`, where `x` is a variable, and `p`, `q`, and `r` are parameters. The above rules verify that this expression is convex, provided `p` is nonnegative and `q` is nonpositive. The expression is verified to be concave, provided `p` is nonpositive and `q` is nonnegative. The expression is invalid in all other cases.

3.4 Objective and constraints

The objective is a direction (`minimize` or `maximize`) and a (respectively, convex or concave) scalar expression. Feasibility problems are specified by omitting the objective. The problem specification in figure 2 is a concave maximization problem.

Constraints have an expression, a relation sign (`<=`, `==` or `>=`) and another expression. Valid constraints must take one of the forms:

- `convex <= concave`,
- `concave <= convex` or
- `affine == affine`.

The CVXGEN specification in figure 2 contains two constraints: an affine equality constraint and a convex-less-than-affine inequality constraint (which is a special case of convex-less-than-concave, since affine functions are also concave). In CVXGEN, the square function cannot appear in constraints, since the problem is converted to a convex QP.

4 Using CVXGEN

CVXGEN performs syntax, dimension and convexity checks on each problem description. Once the problem description has been finalized, CVXGEN converts the description into a custom C solver. The user interface to the generated solver has just a few parts. No configuration, beyond the problem description, is required prior to code generation.

4.1 Generated files

Code generation produces five primary C source files. The bulk of the algorithm is contained in `solver.c`, which has the main `solve` function and core routines. KKT matrix factorization and solution is carried out by functions in `ldl.c`, while `matrix_support.c` contains code for filling vectors and matrices, and performing certain matrix-vector products. All data structures and function prototypes are defined in `solver.h`, and `testsolver.c` contains simple driver code for exercising the solver.

Additional functions for testing are provided by `util.c`, and a `Makefile` is supplied for automated building. CVXGEN also generates code for a Matlab interface, including a driver for simple comparison to CVX.

4.2 Using the generated code

For suitability when embedding, CVXGEN solvers require no dynamic memory allocation. Each solver uses four data structures, which can be statically allocated and initialized just once. These contain problem data (in the `params` data structure), algorithm settings (in `settings`), additional working space (in `work`), and, after solution, optimized variable values (in `vars`).

Once the structures have been defined, the solver can be used in a simple control or optimization loop like this:

```
for (;;) { // Main control loop.
  load_data(params);
  // Solve individual problem instance defined in params.
```

```
  num_iters = solve(params, vars, work, settings);
  // Solution available in vars; status details in work.
}
```

All data in CVXGEN are stored in flat arrays, in column-major form with zero-based indices. For consistency, the same applies for vectors, and even scalars. Symmetric matrices are stored in exactly the same way, but only the diagonal entries are stored for `diagonal` matrices. For performance reasons, no size, shape or attribute checks are performed on parameters. In all cases, we assume that valid data are provided to CVXGEN.

4.3 Solver settings

While CVXGEN is designed for excellent performance with no configuration, several customizations are available. These are made by modifying values inside the `settings` structure. The most important settings are

`settings.eps`, with default 10^{-6}. CVXGEN will not declare a problem converged until the duality gap is known to be bounded by `eps`.

`settings.resid_tol`, with default 10^{-4}. CVXGEN will not declare a problem converged until the norm of the equality and inequality residuals are both less than `resid_tol`.

`settings.max_iters`, with default 25. CVXGEN will exit early if `eps` and `resid_tol` are satisfied. It will also exit when it has performed `max_iters` iterations, regardless of the quality of the point it finds. Most problems require far fewer than 25 iterations.

`settings.kkt_reg`, with default 10^{-7}. This controls the regularization ϵ added to the KKT matrix. See §5.3.1.

`settings.refine_steps`, with default 1. This controls the number of steps of iterative refinement. See §5.3.2.

4.4 Handling infeasibility and unboundedness

The solver generated by CVXGEN does not explicitly handle infeasible or unbounded problems. In both cases, the solver will terminate once it reaches the iteration limit, without convergence. This is by design, and can be overcome by using a model which is always feasible.

One way to ensure feasibility is to replace constraints with penalty terms for constraint violation. For example, instead of the equality constraint $Ax = b$, add the penalty term $\lambda\|Ax - b\|_1$, with $\lambda > 0$, to the objective. This term is the sum of the absolute values of the constraint violations. With sufficiently large λ, the constraint will be satisfied (provided the problem is feasible); see, e.g., [Ber75]. Inequality constraints $Gx \leq h$ can be treated in a similar way, using a penalty term $\lambda\mathbf{1}^T(Gx - h)_+$.

A (classical) option for handling possible infeasibility is to create an additional 'phase I solver', which finds a feasible point if one exists, and otherwise finds a point that minimizes some measure of infeasibility. This solver can be called after the original solver has failed to converge [BV04, §11.4].

To avoid unbounded problems, problems should include additional constraints, such as lower and upper bounds on some or all variables. These should be set sufficiently large so that bounded problems are unaffected, and may be checked for tightness, after solution. (If any bound constraints are tight, we mark the problem instance as likely unbounded.)

4.5 Increasing solver speed

CVXGEN is designed to solve convex optimization problems extremely quickly with default settings. Several improvements are available, however, for the user wanting best performance. The most important technique is to make the optimization problem as small as possible, by reducing the number of variables, constraints or objective terms. With model predictive control problems, for example, see [WB08].

An important part of optimization is compiler choice. We recommend using the most recent compiler for your platform, along with appropriate compiler optimizations. The results here were generated with `gcc-4.4`, with the `-Os` option. Good optimization settings are important: A typical improvement with the right settings is a factor of three. Using `-Os` is appropriate, since it aims to reduce code size, and CVXGEN problems often have relatively large code size.

Changing the solver settings can also improve performance. For applications where average solve times are more important than maximum times, we recommend using relaxed constraint satisfaction and duality gap specifications (see §4.3), which allow early termination once a good (but not provably optimal) solution is found. Often, a near-optimal point is found early, with subsequent iterations merely confirming the point's quality.

If the maximum solve time is more important than the average time, lower the fixed iteration limit. This may lead to a reduced-quality (or even infeasible) solution, and should be used with care, but will give excellent performance for some applications. Again, see [WB08].

5 Implementation

In this portion of the paper, we describe the techniques used to create CVXGEN solvers and make them fast and robust. While CVXGEN handles only problems that transform to QPs, nearly all of the techniques described would apply, with minimal variation, to more general convex problem families.

5.1 Parsing and canonicalization

Before code generation, CVXGEN problem specifications, in the form discussed in §3, are parsed and converted to an internal CVXGEN representation. All convexity and dimension checking is performed in this internal layer. Once parsed, the problem family is analyzed to determine the problem transformations required to target a single canonical form. With vector variable $x \in \mathbf{R}^n$, the canonical form is

$$\begin{array}{ll} \text{minimize} & (1/2)x^T Q x + q^T x \\ \text{subject to} & Gx \leq h, \quad Ax = b, \end{array} \tag{3}$$

with problem data $Q \in \mathbf{S}^n_+$, $q \in \mathbf{R}^n$, $G \in \mathbf{R}^{p \times n}$, $h \in \mathbf{R}^p$, $A \in \mathbf{R}^{m \times n}$ and $b \in \mathbf{R}^m$.

Importantly, the output of the parsing stage is not a single transformed problem, but instead a method for performing the mapping between problem instance data and the generated custom CVXGEN solver. In particular, the output is C code that takes a problem instance and transforms it for use as the Q, q, G, h, A and b in the canonical form. This step also produces code for taking the optimal point x from the canonical form, and transforming it back to the variables in the original CVXGEN problem specification.

Transformations are performed by recursive epigraphical (hypographical) expansions. Each expansion replaces a non-affine convex (concave) function with a newly introduced variable, and adds additional constraints to create an equivalent problem. For a simple example, consider the constraint, with variable $y \in \mathbf{R}^n$,

$$\|Ay - b\|_\infty \leq 3.$$

By introducing the variable $t \in \mathbf{R}^m$ (assuming $A \in \mathbf{R}^{m \times n}$), the original constraint can be replaced with the constraints

$$\mathbf{1}^T t \leq 3, \quad -t \leq Ax - b \leq t,$$

which, crucially, are all affine.

This process is performed recursively, for the objective and all constraints, until all constraints are affine and the objective is affine-plus-quadratic. After that, all variables are vertically stacked into one, larger, variable x, and the constraints and objective are written in terms of the new variable. Finally, code is generated for the forward and backward transformations.

5.2 Solving the standard-form QP

Once the problem is in canonical form, we use a standard primal-dual interior point method to find the solution. While there are alternatives, such as active set or first order methods, an interior point method is particularly appropriate for embedded optimization, since, with proper implementation and tuning, it can reliably solve to high accuracy in 5–25 iterations, without warm start. While we initially used a primal barrier method, we found that

primal-dual methods, particularly with Mehrotra predictor-corrector, give more consistent performance on a wide range of problems.

For completeness, we now describe the algorithm. This standard algorithm is taken from [Van10], but similar treatments may be found in [Wri97, NW99, Stu02], with the Mehrotra predictor-corrector, in particular, described in [Meh92, Wri97].

Introduce slack variables. Given a QP in the form (3), introduce a slack variable $s \in \mathbf{R}^p$, and solve the equivalent problem

$$\begin{array}{ll} \text{minimize} & (1/2)x^TQx + q^Tx \\ \text{subject to} & Gx + s = h, \quad Ax = b, \quad s \geq 0, \end{array}$$

with variables $x \in \mathbf{R}^n$ and $s \in \mathbf{R}^p$. With dual variables $y \in \mathbf{R}^m$ associated with the equality constraints, and $z \in \mathbf{R}^p$ associated with the inequality constraints, the KKT conditions for this problem are

$$\begin{array}{c} Gx + s = h, \quad Ax = b, \quad s \geq 0 \\ z \geq 0 \\ Qx + q + G^Tz + A^Ty = 0 \\ z_i s_i = 0, \quad i = 1, \ldots, p. \end{array}$$

Initialization. The initialization we use exactly follows that given in [Van10, §5.3]. We first find the (analytic) solution of the pair of primal and dual problems

$$\begin{array}{ll} \text{minimize} & (1/2)x^TQx + q^Tx + (1/2)\|s\|_2^2 \\ \text{subject to} & Gx + s = h, \quad Ax = b, \end{array}$$

with variables x and s, and

$$\begin{array}{ll} \text{minimize} & -(1/2)w^TQw - h^Tz - b^Ty - (1/2)\|z\|_2^2 \\ \text{subject to} & Qw + q + G^Tz + A^Ty = 0, \end{array}$$

with variables w, y and z. We can solve for the optimality conditions of both problems simultaneously by solving the linear system

$$\begin{bmatrix} Q & G^T & A^T \\ G & -I & 0 \\ A & 0 & 0 \end{bmatrix} \begin{bmatrix} x \\ z \\ y \end{bmatrix} = \begin{bmatrix} -q \\ b \\ h \end{bmatrix}.$$

We then use the solution to set the initial primal and dual variables to $x^{(0)} = x$ and $y^{(0)} = y$. Then, we set $z = Gx - h$ and $\alpha_p = \inf\{\alpha \mid -z + \alpha\mathbf{1} \geq 0\}$, and use

$$s^{(0)} = \begin{cases} -z & \alpha_p < 0 \\ -z + (1 + \alpha_p)\mathbf{1} & \text{otherwise} \end{cases}$$

as the initial value of s. Finally, we set $\alpha_d = \inf\{\alpha \mid z + \alpha\mathbf{1} \geq 0\}$, and use

$$z^{(0)} = \begin{cases} z & \alpha_d < 0 \\ z + (1 + \alpha_d)\mathbf{1} & \text{otherwise} \end{cases}$$

as the initial value of z. We now have the starting point $(x^{(0)}, s^{(0)}, z^{(0)}, y^{(0)})$.

Main iterations.

1. Evaluate stopping criteria (residual sizes and duality gap). Halt if the stopping criteria are satisfied.

2. Compute affine scaling directions by solving

$$\begin{bmatrix} Q & 0 & G^T & A^T \\ 0 & Z & S & 0 \\ G & I & 0 & 0 \\ A & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} \Delta x^{\text{aff}} \\ \Delta s^{\text{aff}} \\ \Delta z^{\text{aff}} \\ \Delta y^{\text{aff}} \end{bmatrix} = \begin{bmatrix} -(A^T y + G^T z + Px + q) \\ -Sz \\ -(Gx + s - h) \\ -(Ax - b) \end{bmatrix},$$

where $S = \mathbf{diag}(s)$ and $Z = \mathbf{diag}(z)$. We will shortly see that we do not solve this system directly.

3. Compute centering-plus-corrector directions by solving

$$\begin{bmatrix} Q & 0 & G^T & A^T \\ 0 & Z & S & 0 \\ G & I & 0 & 0 \\ A & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} \Delta x^{\text{cc}} \\ \Delta s^{\text{cc}} \\ \Delta z^{\text{cc}} \\ \Delta y^{\text{cc}} \end{bmatrix} = \begin{bmatrix} 0 \\ \sigma\mu\mathbf{1} - \mathbf{diag}(\Delta s^{\text{aff}})\Delta z^{\text{aff}} \\ 0 \\ 0 \end{bmatrix},$$

where $\mu = s^T z/p$,

$$\sigma = \left(\frac{(s + \alpha\Delta s^{\text{aff}})^T (z + \alpha\Delta z^{\text{aff}})}{s^T z} \right)^3$$

and

$$\alpha = \sup\{\alpha \in [0,1] \mid s + \alpha\Delta s^{\text{aff}} \geq 0, z + \alpha\Delta z^{\text{aff}} \geq 0\}.$$

4. Update the primal and dual variables. Combine the two updates using

$$\begin{aligned} \Delta x &= \Delta x^{\text{aff}} + \Delta x^{\text{cc}} \\ \Delta s &= \Delta s^{\text{aff}} + \Delta s^{\text{cc}} \\ \Delta y &= \Delta y^{\text{aff}} + \Delta y^{\text{cc}} \\ \Delta z &= \Delta z^{\text{aff}} + \Delta z^{\text{cc}} \end{aligned},$$

then find an appropriate step size that maintains nonnegativity of s and z,

$$\alpha = \min\{1,\ 0.99 \sup\{\alpha \geq 0 \mid s + \alpha\Delta s \geq 0, z + \alpha\Delta z \geq 0\}\}.$$

5. Update primal and dual variables:

$$\begin{aligned} x &:= x + \alpha\Delta x \\ s &:= s + \alpha\Delta s \\ y &:= y + \alpha\Delta y \\ z &:= z + \alpha\Delta z \end{aligned}.$$

6. Repeat from step 1.

Nearly all of the computational effort is in the solution of the linear systems in steps 2 and 3. As well as requiring most of the computational effort, the linear system solution is the only operation which requires (hazardous) floating-point division and a risk of algorithm failure. Thus, it is important to have a robust method for solving the linear systems.

5.3 Solving the KKT system

Each iteration of the primal-dual algorithm requires two solves with the so-called KKT matrix. We will symmetrize this matrix, and instead find solutions ℓ to the system $K\ell = r$, with two different right-hand sides r, and the block 2×2 system

$$K = \left[\begin{array}{cc|cc} Q & 0 & G^T & A^T \\ 0 & S^{-1}Z & I & 0 \\ \hline G & I & 0 & 0 \\ A & 0 & 0 & 0 \end{array}\right].$$

The matrix K is quasisemidefinite, *i.e.*, symmetric with $(1, 1)$ block diagonal positive semidefinite, and $(2, 2)$ block diagonal negative semidefinite. This special structure occurs in most interior-point methods, and allows us to use special solve methods [Tum02, Van95, VC93]. In our case, we will solve this system using a permuted LDL^T factorization with diagonal matrix D, and unit lower-triangular matrix L. With a suitable permutation matrix P, we will find a factorization

$$PKP^T = LDL^T,$$

where, if the factorization exists, L and D are unique. Additionally, the sign pattern of the diagonal entries of D is known in advance [Tum02].

In a traditional optimization setting, we would choose the permutation P *on-line*, with full knowledge of the numerical values of K. This allows us to pivot to maintain stability and ensure existence of the factorization [GSS96], but has the side effect of requiring complex data structures and nondeterministic code that involves extensive branching. This contributes significant overhead to the factorization. If, by contrast, we choose the permutation off-line, we can generate explicit, branch- and loop-free code that can be executed far more quickly. Unfortunately, for quasisemidefinite K we cannot necessarily choose, in advance, a permutation for which the factorization exists and is stable. In fact, the matrix K may even be singular, or nearly so, if the supplied parameters are poor quality.

5.3.1 Regularization

To ensure the factorization always exists and is numerically stable, we will modify the linear system. Instead of the original system K, we will regularize the KKT matrix by choosing

$\epsilon > 0$ and work with

$$\widetilde{K} = \left[\begin{array}{cc|cc} Q & 0 & G^T & A^T \\ 0 & S^{-1}Z & I & 0 \\ \hline G & I & 0 & 0 \\ A & 0 & 0 & 0 \end{array}\right] + \left[\begin{array}{c|c} \epsilon I & 0 \\ \hline 0 & -\epsilon I \end{array}\right].$$

This new matrix $\widetilde{K}$ is *quasidefinite*, *i.e.*, symmetric with $(1,1)$ block diagonal positive definite, and $(2,2)$ block diagonal negative definite. This means that, for any permutation P the LDL^T factorization must exist [GSS96]. Thus, to find a solution to the system $\widetilde{K}\ell = r$, we can permute and factor $\widetilde{K}$ so that

$$P\widetilde{K}P^T = LDL^T,$$

then find solutions ℓ via

$$\ell = \widetilde{K}^{-1}r = P^T L^{-T} D^{-1} L^{-1} P r,$$

where $(\cdot)^{-1}$ denotes not matrix inversion and multiplication, but the application of backward substitution, scaling, and forward substitution, respectively. This provides solutions to the perturbed system of equations, with coefficient matrix $\widetilde{K}$ instead of K. This is not necessarily a problem, since the search direction found via this method is merely a heuristic, and good performance can still be obtained with $\widetilde{K} \approx K$. However, we will now discuss a method that allows us to recover solutions to the original system with coefficient matrix K.

5.3.2 Iterative refinement

While we can easily find solutions to the system $\widetilde{K}\ell = r$, we actually want solutions to the system $K\ell = r$. We will use iterative refinement to find successive estimates $\ell^{(k)}$ that get progressively closer to solving $K\ell = r$, while using only the operator $\widetilde{K}^{-1}$. (See [DER89, §4.11] for more details). We now describe the algorithm for iterative refinement.

1. Solve $\widetilde{K}\ell^{(0)} = r$ and set $k = 0$. This gives an initial estimate.
2. We now desire a correction term $\delta\ell$ so that $K(\ell^{(k)} + \delta\ell) = r$. However, this would require solving $K\delta\ell = r - K\ell^{(k)}$ to find $\delta\ell$, which would require an operator K^{-1}. Instead, find an approximate correction $\delta\ell^{(k)} = \widetilde{K}^{-1}(r - K\ell^{(k)})$.
3. Update the iterate $\ell^{(k+1)} = \ell^{(k)} + \delta\ell^{(k)}$, and increment k.
4. Repeat steps (2) and (3) until the residual $\|K\ell^{(k)} - r\|$ is sufficiently small. Use $\ell^{(k)}$ as an estimated solution to the system $K\ell = r$.

With this particular choice of $\widetilde{K}$, it can be shown that iterative refinement will converge to a solution of the system with K.

5.3.3 Dynamic regularization

We wish to ensure that the factorization and solution methods can never fail, and in particular, that they never cause floating-point exceptions or excessively large numerical errors. Apart from floating-point overflow caused by data with gross errors, the only possible floating-point problems would come from divide-by-zero operations involving the diagonal entries D_{ii}. If we can ensure that each D_{ii} is bounded away from zero, we avoid these problems.

As mentioned above, we know the sign of each D_{ii} at development time. Specifically, $D_{ii} \geq \epsilon$ corresponds to an entry from the $(1,1)$ block before permutation, and $D_{ii} \leq -\epsilon$ to an entry from the $(2,2)$ block. In the absence of numerical errors or poor data, we already have the necessary guarantee to ensure safe division. However, for safe performance in the presence of such defects, where the computed $\widehat{D}_{ii} \neq D_{ii}$, we will instead use

$$D_{ii} \;=\; s_i((s_i\widehat{D}_{ii})_+ + \epsilon),$$

which is clearly bounded away from zero, and will thus prevent floating-point exceptions. It has a clear interpretation, too: $(s_i\widehat{D}_{ii})_-$ is additional, *dynamic* regularization. Conveniently, iterative refinement with this modified system will still converge, allowing us to obtain a solution to the original KKT system.

5.3.4 Choosing a permutation

In a previous section, we described how, after regularization, for any choice of permutation matrix P, the factorization

$$P\widetilde{K}P^T = LDL^T$$

will exist and will be unique. However, the choice of P is important in another way: It determines the number, and pattern, of nonzero entries in L. All nonzero entries in the lower triangular portion of $P\widetilde{K}P^T$ will cause corresponding nonzero entries in L; additional nonzero entries in L are called *fill-in*. We wish to (approximately) minimize the number of nonzero entries in L, as it approximately corresponds to the amount of work required to factorize and solve the linear system. Thus, we will use a heuristic to choose P to minimize the nonzero count in L. We will use a simple, greedy method, called *local minimum fill-in* [DER89, §7.5]. This technique requires comparatively large amounts of time to determine, but with CVXGEN occurs at code generation time, and thus has no solve time penalty. We now describe the permutation selection algorithm.

1. Create an undirected graph L from $\widetilde{K}$. Initialize the empty list of eliminated nodes, E.

2. For each node $i \notin E$, calculate the fill-in if it were eliminated next. This is simply the number of missing links between uneliminated nodes $j,k \notin E$ for which $L_{jk} = 0$ and $L_{ij} = L_{ik} = 1$.

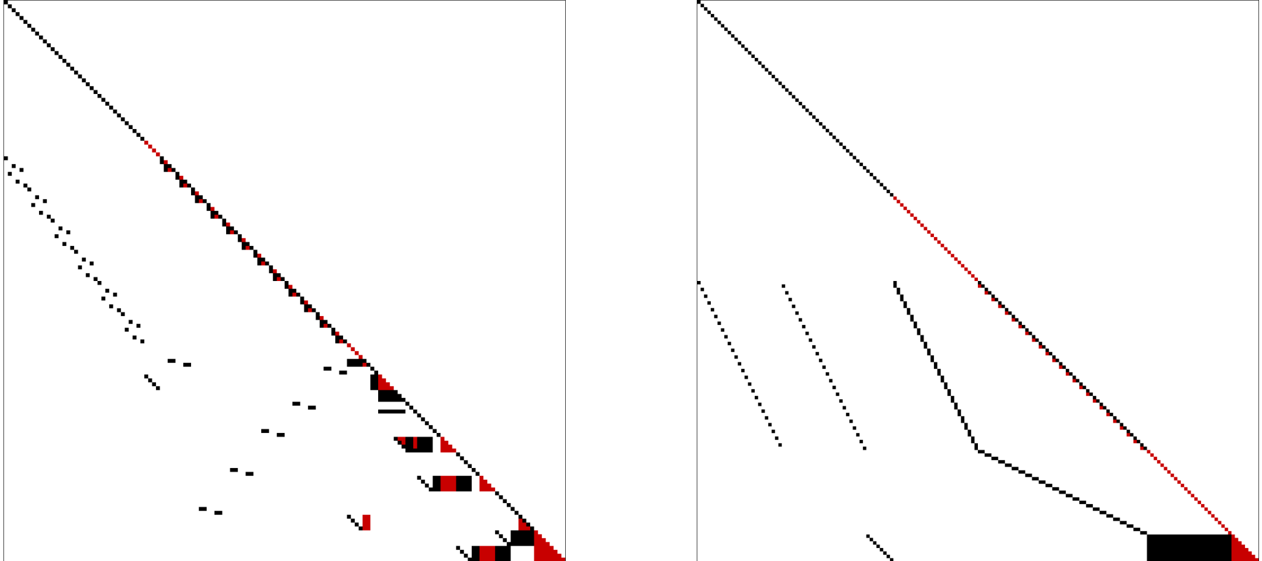


Figure 3: Sparsity patterns of the factor L, with red indicating fill-in.

3. Select the node i for which the fill-in would be lowest, add it to E, and make the appropriate changes to L.
4. Repeat steps (2) and (3) until all nodes have been eliminated. This gives us the elimination ordering, and the structure of non-zero elements in the factor L.

Two example sparsity patterns, after permutation and fill-in, are shown in figure 3. Elements that constitute fill-in are shown in red. The pattern on the left-hand side is for an MPC problem like those described in §6.4. There are 398 non-zero entries in the (non-strict) lower triangle of the regularized KKT matrix; after permutation and fill-in, there are 509 non-zero entries in L. This gives a fill-in factor of 1.28. The pattern on the right-hand side is for a lasso problem like those of §6.3. There are 358 non-zero entries in the KKT matrix; afterward, there are 411, for a fill-in factor of 1.15.

5.4 Code generation

The goal of code generation is to describe the structure and implementation of a solver once, then programmatically transform that implementation, any number of times, into a code tailored for a specific problem. This is much like a compiler, which allows programmers to write code in a more powerful, higher level language, while still getting the performance from (say) assembly code after compilation. CVXGEN uses a templating language to describe the general solver structure, and a modeling and generation layer that fills the holes in each template with detailed code specific to each solver.

5.4.1 Templating language

Much of the code is nearly identical in every generated solver, with only the details changing. This is captured by a templating language, which allows a combination of generic boilerplate code, and problem-specific substitutions to be written in one unified form. CVXGEN uses a templating language where in-place substitutions are marked by '`#{·}`', whole-line substitutions are marked with '`=`', and control logic is marked with '`-`'. Consider this simple example, which generates code for evaluating the surrogate duality gap:

```
gap = 0;
for (i = 0; i < #{p}; i++)
  gap += work.z[i]*work.s[i];
```

Here, `#{p}` is an in-place substitution. Thus, for a problem with 100 inequality constraints, *i.e.*, with $p = 100$, this code segment will be replaced with

```
gap = 0;
for (i = 0; i < 100; i++)
  gap += work.z[i]*work.s[i];
```

This extremely basic example demonstrates how the template has the flavor of a general purpose solver before code generation (using symbolic `p`), but a very specific solver afterwards (using numeric `100`).

For a more involved example, consider this segment, which is a function for multiplying the KKT matrix and `source` and storing the `result`:

```
void kkt_multiply(double *result, double *source) {
  - kkt.rows.times do |i|
    result[#{i}] = 0;
    - kkt.neighbors(i).each do |j|
      - if kkt.nonzero? i, j
        result += #{kkt[i,j]}*source[#{j}];
}
```

Here, we see plain C code (in black), control statements (in green) and in-text substitutions (in blue). The control statements allow us to loop, at development time, over the nonzero entries of the symbolic `kkt` structure, determine the non-zero products, and emit code that describes exactly how to multiply with the given `kkt` structure. In fact, the segment `#{kkt[i,j]}` will be replaced with an expression that could be anything from `1`, describing a constant, a parameter reference such as `params.A[12]`, or even a multiplication such as `2*params.lambda[0]*vars.s_inv[15]`. Thus, with a very short description length in the templating language, we get extremely explicit, highly optimizable code ready for processing by the compiler.

5.4.2 Explicit coding style

The simple KKT multiplication code above illustrates a further point: In CVXGEN, the generated code is extremely explicit. Conventional solvers use sparse matrix libraries like UMFPACK and CHOLMOD [Dav03, Dav06] to perform matrix operations and factorizations. These require only small amounts of code, and are well tested, but carry significant overhead, since the sparse structures must be repeatedly unpacked, evaluated to determine necessary operations, then repacked. By contrast, CVXGEN determines the necessary operations at code development time, then uses flat data structures and explicit references to individual data elements. This means verbose, explicit code, which can be bulky for larger problems, but, after compilation by an optimizing compiler, performs faster than standard libraries.

6 Numerical examples

In this section, we give a series of examples to demonstrate the speed of CVXGEN solvers. For each of the four examples, we create several problem families with differing dimensions, then test performance for 10 000 to 1 million problem instances (depending on solve time). The data for each problem instance are generated randomly, but in a way that would be plausible for each application, and that guarantees the feasibility and boundedness of each problem instance.

We will show results with two different computers. The first is a powerful desktop, running an Intel Core i7-860 with maximum single-core clock speed of 3.46 GHz, an 8 MB Level 2 cache and 95 W peak processor power consumption. The second is an Intel Atom Z530 at 1.60 GHz, with 512 kB of Level 2 Cache and just 2 W of peak power consumption.

6.1 Simple quadratic program

For the first example, we consider the basic quadratic program

$$\begin{array}{ll} \text{minimize} & x^T Q x + c^T x \\ \text{subject to} & Ax = b, \quad 0 \leq x \leq 1, \end{array}$$

with optimization variable $x \in \mathbf{R}^n$ and parameters $A \in \mathbf{R}^{m \times n}$, $b \in \mathbf{R}^m$, $c \in \mathbf{R}^n$ and $Q \in \mathbf{S}^n_+$. Results for three different problem sizes are shown in Table 1.

6.2 Support vector machine

This example, from machine learning, demonstrates the creation of a support vector machine [BV04, §8.6.1]. In this problem, we are given observations $(x_i, y_i) \in \mathbf{R}^n \times \{-1, 1\}$, for $i = 1, \ldots, N$, and a parameter $\lambda \in \mathbf{R}_+$. We wish to choose two optimization variables: a weight vector $w \in \mathbf{R}^n$, and an offset $b \in \mathbf{R}$ that solve the optimization problem

$$\text{minimize} \quad \|w\|_2^2 + \lambda \textstyle\sum_{i=1,\ldots,N} \left(1 - y_i(w^T x_i - b)\right)_+ .$$

Size (m, n)	Small $(3, 10)$	Medium $(6, 20)$	Large $(12, 40)$
CVX and SeDuMi	230 ms	260 ms	340 ms
Scalar parameters	143	546	2132
Variables, original	10	20	40
Variables, transformed	10	20	40
Equalities, transformed	3	6	12
Inequalities, transformed	20	40	80
KKT matrix dimension	53	106	212
KKT matrix nonzeros	165	490	1620
KKT factor fill-in	1.00	1.00	1.00
Code size	123 kB	377 kB	1891 kB
Generation time, i7	0.6 s	5.6 s	95 s
Compilation time, i7	1.1 s	4.2 s	56 s
Binary size, i7	67 kB	231 kB	1256 kB
CVXGEN, i7	26 μs	110 μs	720 μs
CVXGEN, Atom	250 μs	860 μs	4.6 ms
Maximum iterations required, 99.9%	8	9	11
Maximum iterations required, all	20	17	12

Table 1: Performance results for the simple quadratic program example.

Size (N, n)	Medium $(50, 10)$	Large $(100, 20)$
CVX and SeDuMi	750 ms	1400 ms
Scalar parameters	551	2101
Variables, original	11	21
Variables, transformed	61	121
Equalities, transformed	0	0
Inequalities, transformed	100	200
KKT matrix dimension	261	521
KKT matrix nonzeros	960	2920
KKT factor fill-in	1.11	1.11
Code size	712 kB	2334 kB
Generation time, i7	25 s	420 s
Compilation time, i7	8.3 s	54 s
Binary size, i7	367 kB	1424 kB
CVXGEN, i7	250 μs	1.1 ms
CVXGEN, Atom	2.4 ms	9.3 ms
Maximum iterations required, 99.9%	11	12
Maximum iterations required, all	15	18

Table 2: Performance results for the support vector machine example.

Table 2 shows the results for two problem families of different sizes.

6.3 Lasso

This example, from statistics, demonstrates the lasso procedure (ℓ_1-regularized least squares) [BV04, §6.3.2]. Here we wish to solve the optimization problem

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

with parameters $A \in \mathbf{R}^{m \times n}$, $b \in \mathbf{R}^m$ and $\lambda \in \mathbf{R}_+$, and optimization variable $x \in \mathbf{R}^n$. The problem is interesting both when $m < n$, and when $m > n$. Table 3 shows performance results for an example from each case.

6.4 Model predictive control

This example, from control systems, is for model predictive control (MPC). See [MWB10] for several detailed CVXGEN MPC examples. For this example, we will solve the optimization problem

$$\begin{array}{ll} \text{minimize} & \sum_{t=0}^{T} \left(x_t^T Q x_t + u_t^T R u_t \right) + x_{T+1}^T Q_{\text{final}} x_{T+1} \\ \text{subject to} & x_{t+1} = A x_t + B u_t, \quad t = 0, \ldots, T \\ & |u_t| \leq u_{\max}, \quad t = 0, \ldots, T, \end{array}$$

Family (N, n)	Overdetermined $(100, 10)$	Underdetermined $(10, 100)$
CVX and SeDuMi	170 ms	200 ms
Scalar parameters	1101	1011
Variables, original	10	100
Variables, transformed	20	210
Equalities, transformed	0	10
Inequalities, transformed	20	200
KKT matrix dimension	60	620
KKT matrix nonzeros	155	3050
KKT factor fill-in	1.06	1.13
Code size	454 kB	1089 kB
Generation time, i7	18 s	130 s
Compilation time, i7	4.8 s	23 s
Binary size, i7	215 kB	631 kB
CVXGEN, i7	33 μs	660 μs
CVXGEN, Atom	280 μs	4.2 ms
Maximum iterations required, 99.9%	7	9
Maximum iterations required, all	7	10

Table 3: Performance results for the lasso example.

Size (m,n,T)	Small $(2,3,10)$	Medium $(3,5,10)$	Large $(4,8,20)$
CVX and SeDuMi	870 ms	880 ms	1.6 s
Scalar parameters	41	105	249
Variables, original	55	88	252
Variables, transformed	77	121	336
Equalities, transformed	33	55	168
Inequalities, transformed	66	99	252
KKT matrix dimension	242	374	1008
KKT matrix nonzeros	552	1025	3568
KKT factor fill-in	1.30	1.44	1.60
Code size	337 kB	622 kB	2370 kB
Generation time, i7	4.3 s	13 s	200 s
Compilation time, i7	3.6 s	9.4 s	41 s
Binary size, i7	175 kB	351 kB	1445 kB
CVXGEN, i7	85 μs	230 μs	970 μs
CVXGEN, Atom	1.7 ms	3.3 ms	13 ms
Maximum iterations required, 99.9%	13	13	12
Maximum iterations required, all	23	24	24

Table 4: Performance results for the model predictive control example.

with optimization variables $x_1, \ldots, x_{T+1} \in \mathbf{R}^n$ (state variables) and $u_0, \ldots, u_T \in \mathbf{R}^m$ (input variables); and problem data consisting of system dynamics matrices $A \in \mathbf{R}^{n\times n}$ and $B \in \mathbf{R}^{n\times m}$; input cost diagonal $R \in \mathbf{S}_+^{m\times m}$, state and final state costs diagonal $Q \in \mathbf{S}_+^{n\times n}$ and dense $Q_{\text{final}} \in \mathbf{S}_+^{n\times n}$; amplitude and slew rate limits $u_{\max} \in \mathbf{R}_+$ and $S \in \mathbf{R}_+$; and initial state $x_0 \in \mathbf{R}^n$. Table 4 shows performance results for three problem families of varying sizes.

6.5 Settings and reliability

To explore and verify the performance of CVXGEN solvers, we tested many different problem families, with at least millions, and sometimes hundreds of millions, of problem instances for each family. Since the computation time of the solver is almost exactly proportional to the number of iterations, we verified both reliability and performance by recording the number of iterations for every problem instance. Failures, in this case, are not merely problem instances for which the algorithm would never converge, but problem instances which take more than some fixed limit of iterations (say, 20).

CVXGEN solvers demonstrate reliable performance with default solver settings, with minimal dependence on the particular algorithm settings. As an example of this type of analysis, in this section we demonstrate the behavior of a single CVXGEN solver as we vary the solver settings. In each case, we solve the same $100,000$ problem instances, recording the number of iterations required to achieve relatively high accuracy in both provable optimality

gap and equality and inequality residuals.

The problem we will solve is

$$\begin{array}{ll} \text{minimize} & \|Ax - b\|_1 \\ \text{subject to} & -1 \leq x \leq 1, \end{array}$$

with optimization variable $x \in \mathbf{R}^{15}$ and problem data $A \in \mathbf{R}^{8 \times 15}$ and $b \in \mathbf{R}^8$. We will generate data by setting each element $A_{ij} \sim \mathcal{N}(0, 1)$ and $b_{ij} \sim \mathcal{N}(0, 9)$. (With these problem instances, at optimality, approximately 50% of the constraints are active.) The optimal value is nearly always between 1 and 10, and problems are solved to a relatively tight tolerance of 10^{-4} (approximately 0.01%), with constraint residual norms required to be less than 10^{-6}.

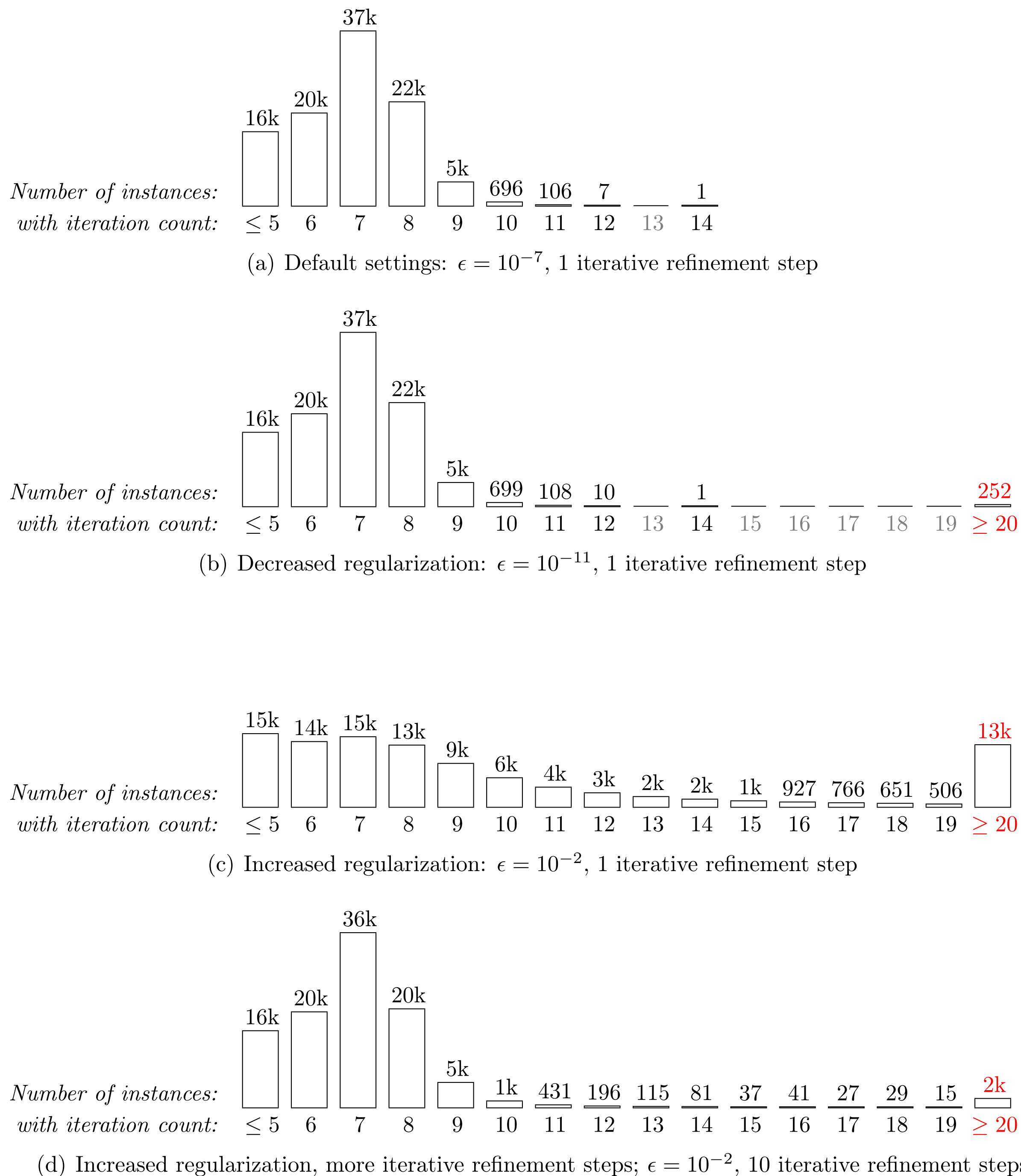
Figure 4(a) shows the performance of the solver with default settings. All problems were solved within 14 iterations, so it would be reasonable to set a maximum iteration limit of 10, at the cost of slightly poorer accuracy in less than 1% of cases.

If the regularization is removed, by setting the regularization $\epsilon = 0$, the solver fails in every case. This is because no factorization of K is possible with the permutation chosen by CVXGEN. However, as long as some regularization is present, solution is still successful. Figure 4(b) shows the behavior of the solver with the regularization decreased by a factor of 10^4, to $\epsilon = 10^{-11}$. This is a major change, yet the solver works nearly as well, encountering the iteration limit in less than 0.2% of cases.

The CVXGEN generated solver shows similarly excellent behavior with increased regularization. To illustrate the point, however, figure 4(c) shows what happens when regularization is increased too much, by a factor of 10^5 to $\epsilon = 10^{-2}$. Even with this excessive regularization, however, the solver still only reaches the iteration limit in 13% of cases.

This extreme case gives us an opportunity to show the effect of iterative refinement. With this excessively high $\epsilon = 10^{-2}$, using 10 iterative refinement steps, as shown in figure 4(d) means the iteration limit is only reached in 2% of cases.

Similar testing was carried out for a much wider selection of problems and solver configurations, and demonstrates that CVXGEN solvers are robust, and perform nearly independently of their exact configuration.



(a) Default settings: $\epsilon = 10^{-7}$, 1 iterative refinement step

(b) Decreased regularization: $\epsilon = 10^{-11}$, 1 iterative refinement step

(c) Increased regularization: $\epsilon = 10^{-2}$, 1 iterative refinement step

(d) Increased regularization, more iterative refinement steps; $\epsilon = 10^{-2}$, 10 iterative refinement steps

Figure 4: Iteration counts for $100,000$ problem instances, with varying solver settings. Labels on x-axis are iteration counts; bar heights and labels are the number of problem instances requiring that many iterations.

7 Conclusion

CVXGEN is, as far as we are aware, the first automatic code generator for convex optimization. It shows the feasibility of automatically generating extremely fast solvers, directly from a high level problem family description. In addition to high speed, the generated solvers have no library dependencies, and are almost branch free, making them suitable for embedding in real-time applications.

The current implementation is limited to small and medium sized problems, that can be transformed to QPs. The size limitation is mostly due to our choice of generating explicit factorization code; handling dense blocks separately would go a long way towards alleviating this short-coming. Our choice of QP as the target, as opposed to a more general form such as second-order cone program (SOCP), was for simplicity. The changes needed to handle such problems are (in principle) not very difficult. The language needs to be extended, and the solver would need to be modified to handle SOCPs. Fortunately, the current methods for solving the KKT system would work almost without change.

Historically, embedded convex optimization has been challenging and time consuming to use. CVXGEN makes this process much simpler by letting users move from a high level problem description to a fast, robust solver, with minimal effort. We hope that CVXGEN (or similar tools) will greatly increase the interest and use of embedded convex optimization.

Acknowledgments

We are grateful to Lieven Vandenberghe for some very helpful discussions, including suggesting the initialization method and algorithm of §5.2. We also thank early users of CVXGEN, including Yang Wang and Craig Beal, for important bug reports and suggestions.

The research reported here was supported in part by JPL contract 1400723, and NASA grant NNX07AEIIA. Jacob Mattingley was supported in part by a Lucent Technologies Stanford Graduate Fellowship.

References

[Bac96] R. Bacher. Automatic generation of optimization code based on symbolic nonlinear domain formulation. *Proceedings International Symposium on Symbolic and Algebraic Computation*, pages 283–291, 1996.

[Bac97] R. Bacher. Combining symbolic and numeric tools for power system network optimization. *Maple Technical Newsletter*, 4(2):41–51, 1997.

[BB91] S. Boyd and C. Barratt. *Linear Controller Design: Limits of Performance*. Prentice-Hall, 1991.

[BEFB94] S. Boyd, L. El Ghaoui, E. Feron, and V. Balakrishnan. *Linear Matrix Inequalities in System and Control Theory*. Society for Industrial and Applied Mathematics, 1994.

[Ber75] D. P. Bertsekas. Necessary and sufficient conditions for a penalty method to be exact. *Mathematical Programming*, 9(1):87–99, 1975.

[BKPH05] S. Boyd, S.-J Kim, D. Patil, and M. A. Horowitz. Digital circuit optimization via geometric programming. *Operations Research*, 53(6):899–932, 2005.

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

[CRR69] R. Calvin, C. Ray, and V. Rhyne. The design of optimal convolutional filters via linear programming. *IEEE Trans. Geoscience Elec.*, 7(3):142–145, July 1969.

[CST00] N. Cristianini and J. Shawe-Taylor. *An introduction to Support Vector Machines and other kernel-based learning methods*. Cambridge University Press, 2000.

[CT07] G. Cornuejols and R. Tütüncü. *Optimization methods in finance*. Cambridge University Press, 2007.

[Dav03] T. A. Davis. *UMFPACK User Guide*, 2003. Available from `http://www.cise.ufl.edu/research/sparse/umfpack`.

[Dav06] T. A. Davis. *CHOLMOD User Guide*, 2006. Available from `http://www.cise.ufl.edu/research/sparse/cholmod/`.

[DDB95] M. A. Dahleh and I. J. Diaz-Bobillo. *Control of Uncertain Systems: A Linear Programming Approach*. Prentice-Hall, 1995.

[DER89] I. S. Duff, A. M. Erisman, and J. K. Reid. *Direct methods for sparse matrices*. Oxford University Press, USA, 1989.

[EMV03] Y. C. Eldar, A. Megretski, and G. C. Verghese. Designing optimal quantum detectors via semidefinite programming. *IEEE Transactions on Information Theory*, 49(4):1007–1012, 2003.

[GB08a] M. Grant and S. Boyd. CVX: Matlab software for disciplined convex programming (web page and software). `http://www.stanford.edu/~boyd/cvx/`, July 2008.

[GB08b] M. Grant and S. Boyd. Graph implementations for nonsmooth convex programs. In V. Blondel, S. Boyd, and H. Kimura, editors, *Recent advances in Learning and Control (a tribute to M. Vidyasagar)*, pages 95–110. Springer, 2008.

[GBY06] M. Grant, S. Boyd, and Y. Ye. Disciplined convex programming. In L. Liberti and N. Maculan, editors, *Global Optimization: from Theory to Implementation*, Nonconvex Optimization and Its Applications, pages 155–210. Springer Science & Business Media, Inc., New York, 2006.

[GGL96] R. Graham, M. Grötschel, and L. Lovász. *Handbook of combinatorics*, volume 2, chapter 28. MIT Press, 1996.

[Gra04] M. Grant. *Disciplined Convex Programming*. PhD thesis, Department of Electrical Engineering, Stanford University, December 2004.

[GSS96] P. E. Gill, M. A. Saunders, and J. R. Shinnerl. On the stability of Cholesky factorization for symmetric quasidefinite systems. *SIAM J. Matrix Anal. Appl*, 17(1):35–46, 1996.

[HBL01] M. del Mar Hershenson, S. Boyd, and T. H. Lee. Optimal design of a CMOS op-amp via geometric programming. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 20(1):1–21, 2001.

[HMBL99] M. del Mar Hershenson, S. S. Mohan, S. Boyd, and T. H. Lee. Optimization of inductor circuits via geometric programming. In *Design Automation Conference*, pages 994–998. IEEE Computer Society, 1999.

[Joh75] S. C. Johnson. Yacc: Yet another compiler-compiler. *Computing Science Technical Report*, 32, 1975.

[Kan93] E. Kant. Synthesis of mathematical-modeling software. *IEEE Software*, 10(3):30–41, 1993.

[KMT98] F. P. Kelly, A. K. Maulloo, and D. K. H. Tan. Rate control for communication networks: shadow prices, proportional fairness and stability. *Journal of the Operational Research society*, pages 237–252, 1998.

[LÖ4] J. Löfberg. YALMIP: A toolbox for modeling and optimization in MATLAB. In *Proceedings of the CACSD Conference*, Taipei, Taiwan, 2004. `http://control.ee.ethz.ch/~joloef/yalmip.php`.

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

[MB08] J. Mattingley and S. Boyd. CVXMOD: Convex optimization software in Python (web page and software). `http://cvxmod.net/`, August 2008.

[MB09a] J. E. Mattingley and S. Boyd. Automatic code generation for real-time convex optimization. In D. P. Palomar and Y. C. Eldar, editors, *Convex optimization in signal processing and communications*. Cambridge University Press, 2009.

[MB09b] J. E. Mattingley and S. Boyd. Real-time convex optimization in signal processing. *IEEE Signal Processing Magazine*, 23(3):50–61, 2009.

[Meh92] S. Mehrotra. On the implementation of a primal-dual interior point method. *SIAM Journal on Optimization*, 2:575, 1992.

[MWB10] J. E. Mattingley, Y. Wang, and S. Boyd. Code generation for receding horizon control. In *Proceedings IEEE Multi-Conference on Systems and Control*, pages 985–992, September 2010.

[NN94] Y. Nesterov and A. Nemirovskii. *Interior Point Polynomial Algorithms in Convex Programming*, volume 13. SIAM, 1994.

[NW99] J. Nocedal and S. J. Wright. *Numerical Optimization*. Springer, 1999.

[SB04] C. Shi and R. W. Brodersen. Automated fixed-point data-type optimization tool for signal processing and communication systems. *ACM IEEE Design Automation Conference*, pages 478–483, 2004.

[Spe07] IEEE Journal of Selected Topics in Signal Processing, December 2007. Special Issue on Convex Optimization Methods for Signal Processing.

[Stu02] J. F. Sturm. Implementation of interior point methods for mixed semidefinite and second order cone optimization problems. *Optimization Methods and Software*, 17(6):1105–1154, 2002.

[Tum02] M. Tuma. A note on the LDL^T decomposition of matrices from saddle-point problems. *SIAM Journal on Matrix Analysis and Applications*, 23(4):903–915, 2002.

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

[Van10] L. Vandenberghe. The cvxopt linear and quadratic cone program solvers. `http://abel.ee.ucla.edu/cvxopt/documentation/coneprog.pdf`, March 2010.

[Vap00] V. N. Vapnik. *The Nature of Statistical Learning Theory*. Springer, second edition, 2000.

[VC93] R. J. Vanderbei and T. J. Carpenter. Symmetric indefinite systems for interior point methods. *Mathematical Programming*, 58(1):1–32, 1993.

[WB08] Y. Wang and S. Boyd. Fast model predictive control using online optimization. In *Proceedings IFAC World Congress*, pages 6974–6997, July 2008.

[WJLH06] D. X. Wei, C. Jin, S. H. Low, and S. Hegde. FAST TCP: motivation, architecture, algorithms, performance. *IEEE/ACM Transactions on Networking*, 14(6):1246–1259, 2006.

[Wri97] S. J. Wright. *Primal-Dual Interior-Point Methods*. SIAM, 1997.

[Ye97] Y. Ye. *Interior Point Algorithms: Theory and Analysis*. Citeseer, 1997.