1 Automatic code generation for real-time convex optimization

Jacob Mattingley and Stephen Boyd

This chapter concerns the use of convex optimization in real-time embedded systems, in areas such as signal processing, automatic control, real-time estimation, real-time resource allocation and decision making, and fast automated trading. By "embedded" we mean that the optimization algorithm is part of a larger, fully automated system, that executes automatically with newly arriving data or changing conditions, and without any human intervention or action. By "real-time" we mean that the optimization algorithm executes much faster than a typical or generic method with a human in the loop, in times measured in milliseconds or microseconds for small and medium size problems, and (a few) seconds for larger problems. In real-time embedded convex optimization the same optimization problem is solved many times, with different data, often with a hard real-time deadline. In this chapter we propose an automatic code generation system for real-time embedded convex optimization. Such a system scans a description of the problem family, and performs much of the analysis and optimization of the algorithm, such as choosing variable orderings used with sparse factorizations and determining storage structures, at code generation time. Compiling the generated source code yields an extremely efficient custom solver for the problem family. We describe a preliminary implementation, built on the Python-based modeling framework CVXMOD, and give some timing results for several examples.

1.1 Introduction

1.1.1 Advisory optimization

Mathematical optimization is traditionally thought of as an aid to human decision making. For example, a tool for portfolio optimization *suggests* a portfolio to a human decision maker, who possibly carries out the proposed trades. Optimization is also used in many aspects of engineering design; in most cases, an engineer is in the decision loop, continually reviewing the proposed designs and changing parameters in the problem specification, if needed.

When optimization is used in an advisory role, the solution algorithms do not need to be especially fast; an acceptable time might be a few seconds (for example, when analyzing scenarios with a spreadsheet), or even tens of minutes or hours for very large

Convex Optimization in Signal Processing and Communication, eds. Daniel P. Palomar and Yonina C. Eldar.
Published by Cambridge University Press © Cambridge University Press, 2010.

problems (e.g., engineering design synthesis, or scheduling). Some unreliability in the solution methods can be tolerated, since the human decision maker will review the proposed solutions, and hopefully catch problems.

Much effort has gone into the development of optimization algorithms for these settings. For adequate performance, they must detect and exploit a generic problem structure not known (to the algorithm) until the particular problem instance is solved. A good generic *linear programming* (LP) solver, for example, can solve, on human-based time scales, large problems in digital circuit design, supply chain management, filter design, or automatic control. Such solvers are often coupled with optimization modeling languages, which allow the user to efficiently describe optimization problems in a high level format. This permits the user to rapidly see the effect of new terms or constraints.

This is all based on the conceptual model of a human in the loop, with most previous and current solver development effort focusing on scaling to *large* problem instances. Not much effort, by contrast, goes into developing algorithms that solve small- or medium-sized problems on fast (millisecond or microsecond) time scales, and with great reliability.

1.1.2 Embedded optimization

In this chapter we focus on embedded optimization, where solving optimization problems is part of a wider, automated algorithm. Here the optimization is deeply embedded in the application, and no human is in the loop. In the introduction to the book *Convex Optimization* [1], Boyd and Vandenberghe state:

> A relatively recent phenomenon opens the possibility of many other applications for mathematical optimization. With the proliferation of computers embedded in products, we have seen a rapid growth in *embedded optimization*. In these embedded applications, optimization is used to automatically make real-time choices, and even carry out the associated actions, with no (or little) human intervention or oversight. In some application areas, this blending of traditional automatic control systems and embedded optimization is well under way; in others, it is just starting. Embedded real-time optimization raises some new challenges: in particular, it requires solution methods that are extremely reliable, and solve problems in a predictable amount of time (and memory).

In real-time embedded optimization, different instances of the same small- or medium-size problem must be solved extremely quickly, for example, on millisecond or microsecond time scales; in many cases the result must be obtained before a strict real-time deadline. This is in direct contrast to generic algorithms, which take a variable amount of time, and exit only when a certain precision has been achieved.

An early example of this kind of embedded optimization, though not on the time scales that we envision, is *model predictive control* (MPC), a form of feedback control system. Traditional (but still widely used) control schemes have relatively simple control policies, requiring only a few basic operations like matrix-vector multiplies and lookup table searches at each time step [2, 3]. This allows traditional control policies to be executed rapidly, with strict time constraints and high reliability. While the control policies themselves are simple, great effort is expended in developing and tuning (i.e., choosing parameters in) them. By contrast, with MPC, at each step the control action is determined by solving an optimization problem, typically a (convex) *quadratic program* (QP). It was

first deployed in the late 1980s in the chemical process industry, where the hard real-time deadlines were in the order of 15 minutes to an hour per optimization problem [4]. Since then, we have seen huge computer processing power increases, as well as substantial advances in algorithms, which allow MPC to be carried out on the same fast time scales as many conventional control methods [5, 6]. Still, MPC is generally not considered by most control engineers, even though there is much evidence that MPC provides better control performance than conventional algorithms, especially when the control inputs are constrained.

Another example of embedded optimization is program or algorithmic trading, in which computers initiate stock trades without human intervention. While it is hard to find out what is used in practice due to trade secrets, we can assume that at least some of these algorithms involve the repeated solution of linear or quadratic programs, on short, if not sub-second, time scales. The trading algorithms that run on faster time scales are presumably just like those used in automatic control; in other words, simple and quickly executable. As with traditional automatic control, huge design effort is expended to develop and tune the algorithms.

In signal processing, an algorithm is used to extract some desired signal or information from a received noisy or corrupted signal. In *off-line signal processing*, the entire noisy signal is available, and while faster processing is better, there are no hard real-time deadlines. This is the case, for example, in the restoration of audio from wax cylinder recordings, image enhancement, or geophysics inversion problems, where optimization is already widely used. In *on-line* or *real-time signal processing*, the data signal samples arrive continuously, typically at regular time intervals, and the results must be computed within some fixed time (typically, a fixed number of samples). In these applications, the algorithms in use, like those in traditional control, are still relatively simple [7].

Another relevant field is communications. Here a noise-corrupted signal is received, and a decision as to which bit string was transmitted (i.e., the decoding) must be made within some fixed (and often small) period of time. Typical algorithms are simple, and hence fast. Recent theoretical studies suggest that decoding methods based on convex optimization can deliver improved performance [8–11], but the standard methods for these problems are too slow for most practical applications. One approach has been the development of custom solvers for communications decoding, which can execute far faster than generic methods [12].

We also envisage real-time optimization being used in statistics and machine learning. At the moment, most statistical analysis has a human in the loop. But we are starting to see some real-time applications, e.g., spam filtering, web search, and automatic fault detection. Optimization techniques, such as support vector machines (SVMs), are heavily used in such applications, but much like in traditional control design, the optimization problems are solved on long time scales to produce a set of model parameters or weights. These parameters are then used in the real-time algorithm, which typically involves not much more than computing a weighted sum of features, and so can be done quickly. We can imagine applications where the weights are updated rapidly, using some real-time, optimization-based method. Another setting in which an optimization problem might be solved on a fast time scale is real-time statistical inference, in which estimates of the

probabilities of unknown variables are formed soon after new information (in the form of some known variables) arrives.

Finally, we note that the ideas behind real-time embedded optimization could also be useful in more conventional situations with no real-time deadlines. The ability to extremely rapidly solve problem instances from a specific problem family gives us the ability to solve large numbers of similar problem instances quickly. Some example uses of this are listed below.

- *Trade-off analysis.* An engineer formulating a design problem as an optimization problem solves a large number of instances of the problem, while varying the constraints, to obtain a sampling of the optimal trade-off surface. This provides useful design guidelines.
- *Global optimization.* A combinatorial optimization problem is solved using branch-and-bound or a similar global optimization method. Such methods require the solution of a large number of problem instances from a (typically convex, often LP) problem family. Being able to solve each instance very quickly makes it possible to solve the overall problem much faster.
- *Monte Carlo performance analysis.* With Monte Carlo simulation, we can find the distribution of minimum cost of an optimization problem that depends on some random parameters. These parameters (e.g., prices of some resources or demands for products) are random with some given distribution, but will be known before the optimization is carried out. To find the distribution of optimized costs, we use Monte Carlo: we generate a large number of samples of the price vector (say), and for each one we carry out optimization to find the minimal cost. Here, too, we end up solving a large number of instances of a given problem family.

1.1.3 Convex optimization

Convex optimization has many advantages over general nonlinear optimization, such as the existence of efficient algorithms that can reliably find a globally optimal solution. A less appreciated advantage is that algorithms for specific convex optimization problem families can be highly robust and reliable; unlike many general purpose optimization algorithms, they do not have parameters that must be manually tuned for particular problem instances. Convex optimization problems are, therefore, ideally suited to real-time embedded applications, because they can be reliably solved.

A large number of problems arising in application areas like signal processing, control, finance, statistics and machine learning, and network operation can be cast (exactly, or with reasonable approximations) as convex problems. In many other problems, convex optimization can provide a good heuristic for approximate solution of the problem [13, 14].

In any case, much of what we say in this chapter carries over to local optimization methods for nonconvex problems, although without the global optimality guarantee, and with some loss in reliability. Even simple methods of extending the methods of convex optimization can work very well in pratice. For example, we can use a basic interior-point

method as if the problem were convex, replacing nonconvex portions with appropriate convex approximations at each iteration.

1.1.4 Outline

In Section 1.2, we describe problem families and the specification languages used to formally model them, and two general approaches to solving problem instances described this way: via a parser-solver, and via code generation. We list some specific example applications of real-time convex optimization in Section 1.3. In Section 1.4 we describe, in general terms, some requirements on solvers used in real-time optimization applications, along with some of the attributes of real-time optimization problems that we can exploit. We give a more detailed description of how a code generator can be constructed in Section 1.5, briefly describe a preliminary implementation of a code generator in Section 1.6, and report some numerical results in Section 1.7. We give a summary and conclusions in Section 1.8.

1.1.5 Previous and related work

Here we list some representative references that focus on various aspects of real-time embedded optimization or closely related areas.

Control

Plenty of work focuses on traditional real-time control [15–17], or basic model predictive control [18–23]. Several recent papers describe methods for solving various associated QPs quickly. One approach is *explicit MPC*, pioneered by Bemporad and Morari [24], who exploit the fact that the solution of the QP is a piecewise linear function of the problem data, which can be determined analytically ahead of time. Solving instances of the QP then reduces to evaluating a piecewise linear function. Interior-point methods [25], including fast custom interior-point methods [6], can also be used to provide rapid solutions. For fast solution of the QPs arising in evaluation of control-Lyapunov policies (a special case of MPC), see [26]. Several authors consider fast solution of nonlinear control problems using an MPC framework [27–29]. Others discuss various real-time applications [30, 31], especially those in robotics [32–34].

Signal processing, communications, and networking

Work on convex optimization in signal processing includes ℓ_1-norm minimization for sparse signal recovery, recovery of noisy signals, or statistical estimation [35, 36], or linear programming for error correction [37]. Goldfarb and Yin discuss interior-point algorithms for solving total variation image restoration problems [38]. Some combinatorial optimization problems in signal processing that are approximately, and very quickly, solved using convex relaxations and local search are static fault detection [14], dynamic fault detection [39], query model estimation [40] and sensor selection [13]. In communications, convex optimization is used in DSL [41], radar [42], and CDMA [43], to list just a few examples.

Since the publication of the paper by Kelly *et al.* [44], which poses the optimal network flow control as a convex optimization problem, many authors have looked at optimization-based network flow methods [45–48], or optimization of power and bandwidth [49, 50].

Code generation

The idea of automatic generation of source code is quite old. Parser-generators such as Yacc [51], or more recent tools like GNU Bison [52], are commonly used to simplify the writing of compilers. For engineering problems, in particular, there are a range of code generators: one widely used commercial tool is Simulink [53], while the open-source Ptolemy project [54] provides a modeling environment for embedded systems. Domain-specific code generators are found in many different fields [55–58].

Generating source code for optimization solvers is nothing new either; in 1988 Oohori and Ohuchi [59] explored code generation for LPs, and generated explicit Cholesky factorization code ahead of time. Various researchers have focused on code generation for convex optimization. McGovern, in his PhD thesis [60], gives a computational complexity analysis of real-time convex optimization. Hazan considers algorithms for on-line convex optimization [61], and Das and Fuller [62] hold a patent on an active-set method for real-time QP.

1.2 Solvers and specification languages

It will be important for us to carefully distinguish between an instance of an optimization problem, and a parameterized family of optimization problems, since one of the key features of real-time embedded optimization applications is that each of the specific problems to be solved comes from a single family.

1.2.1 Problem families and instances

We consider continuously parameterized families of optimization problems, of the form

$$\begin{array}{ll} \text{minimize} & F_0(x,a) \\ \text{subject to} & F_i(x,a) \le 0, \quad i = 1,\ldots,m \\ & H_i(x,a) = 0, \quad i = 1,\ldots,p, \end{array} \tag{1.1}$$

where $x \in \mathbf{R}^n$ is the (vector) optimization variable, and $a \in \mathcal{A} \subset \mathbf{R}^\ell$ is a parameter or data vector that specifies the problem instance. To specify the problem family (1.1), we need descriptions of the functions $F_0, \ldots, F_m, H_1, \ldots, H_p$, and the parameter set $\mathcal{A}$. When we fix the value of the parameters, by fixing the value of a, we obtain a problem *instance*.

As a simple example, consider the QP

$$\begin{array}{ll} \text{minimize} & (1/2)x^T P x + q^T x \\ \text{subject to} & Gx \le h, \quad Ax = b, \end{array} \tag{1.2}$$

with variable $x \in \mathbf{R}^n$, where the inequality between vectors means componentwise. Let us assume that in all instances we care about, the equality constraints are the same, that is, A and b are fixed. The matrices and vectors P, q, G, and h can vary, although P must be symmetric positive semidefinite. For this problem family we have

$$a = (P, q, G, h) \in \mathcal{A} = \mathbf{S}_+^n \times \mathbf{R}^n \times \mathbf{R}^{m \times n} \times \mathbf{R}^m,$$

where $\mathbf{S}_+^n$ denotes the set of symmetric $n \times n$ positive semidefinite matrices. We can identify a with an element of $\mathbf{R}^\ell$, with total dimension

$$\ell = \underbrace{n(n+1)/2}_{P} + \underbrace{n}_{q} + \underbrace{mn}_{G} + \underbrace{m}_{h}.$$

In this example, we have

$$\begin{aligned} F_0(x, a) &= (1/2)x^T P x + q^T x, \\ F_i(x, a) &= g_i^T x - h_i, \quad i = 1, \ldots, m, \\ H_i(x, a) &= \tilde{a}_i^T x - b_i, \quad i = 1, \ldots, p, \end{aligned}$$

where g_i^T is the ith row of G, and $\tilde{a}_i^T$ is the ith row of A. Note that the equality constraint functions H_i do not depend on the parameter vector a; the matrix A and vector b are constants in the problem family (1.2).

Here we assume that the data matrices have no structure, such as sparsity. But in many cases, problem families do have structure. For example, suppose that we are interested in the problem family in which P is tridiagonal, and the matrix G has some specific sparsity pattern, with N (possibly) nonzero entries. Then $\mathcal{A}$ changes, as does the total parameter dimension, which becomes

$$\ell = \underbrace{2n-1}_{P} + \underbrace{n}_{q} + \underbrace{N}_{G} + \underbrace{m}_{h}.$$

In a more general treatment, we could also consider the dimensions and sparsity patterns as (discrete) parameters that one specifies when fixing a particular problem instance. Certainly when we refer to QP generally, we refer to families of QPs with any dimensions, and not just a family of QPs with some specific set of dimensions and sparsity patterns. In this chapter, however, we restrict our attention to continuously parameterized problem families, as described above; in particular, the dimensions n, m, and p are fixed, as are the sparsity patterns in the data.

The idea of a parameterized problem family is a central concept in optimization (although in most cases, a family is considered to have variable dimensions). For example, the idea of a solution algorithm for a problem family is sensible, but the idea of a solution algorithm for a problem instance is not. (The best solution algorithm for a problem instance is, of course, to output a pre-computed solution.)

Nesterov and Nemirovsky refer to families of convex optimization problems, with constant structure and parameterized by finite dimensional parameter vectors as *well structured problem (families)* [63].

1.2.2 Solvers

A *solver* or *solution method* for a problem family is an algorithm that, given the parameter value $a \in \mathcal{A}$, finds an optimal point $x^\star(a)$ for the problem instance, or determines that the problem instance is infeasible or unbounded.

Traditional solvers [1,64,65] can handle problem families with a range of dimensions (e.g., QPs with the form (1.2), any values for m, n, and p, and any sparsity patterns in the data matrices). With traditional solvers, the dimensions, sparsity patterns and all other problem data a are specified only at solve time, that is, when the solver is invoked. This is extremely useful, since a single solver can handle a very wide range of problems, and exploit (for efficiency) a wide variety of sparsity patterns. The disadvantage is that analysis and utilization of problem structure can only be carried out as each problem instance is solved, which is then included in the per-instance solve time. This also limits the reasonable scope of efficiency gains: there is no point in spending longer looking for an efficient method than it would take to solve the problem with a simpler method.

This traditional approach is far from ideal for real-time embedded applications, in which a very large number of problems, from the same continuously parameterized family, will be solved, hopefully very quickly. For such problems, the dimensions and sparsity patterns are known ahead of time, so much of the problem and efficiency analysis can be done ahead of time (and in relative leisure).

It is possible to develop a custom solver for a specific, continuously parameterized problem family. This is typically done by hand, in which case the development effort can be substantial. On the other hand, the problem structure and other attributes of the particular problem family can be exploited, so the resulting solver can be far more efficient than a generic solver [6,66].

1.2.3 Specification languages

A *specification language* allows a user to describe a problem instance or problem family to a computer, in a convenient, high-level algebraic form. All specification languages have the ability to declare optimization variables; some also have the ability to declare parameters. Expressions involving variables, parameters, constants, supported operators, and functions from a library can be formed; these can be used to specify objectives and constraints. When the specification language supports the declaration of parameters, it can also be used to describe $\mathcal{A}$, the set of valid parameters. (The domains of functions used in the specification may also implicitly impose constraints on the parameters.)

Some specification languages impose few restrictions on the expressions that can be formed, and the objective and constraints that can be specified. Others impose strong restrictions to ensure that specified problems have some useful property such as

convexity, or are transformable to some standard form such as an LP or a *semidefinite program* (SDP).

1.2.4 Parser-solvers

A *parser-solver* is a system that scans a specification language description of a problem *instance*, checks its validity, carries out problem transformations, calls an appropriate solver, and transforms the solution back to the original form. Parser-solvers accept directives that specify which solver to use, or that override algorithm parameter defaults, such as required accuracy.

Parser-solvers are widely used. Early (and still widely used) parser-solvers include AMPL [67] and GAMS [68], which are general purpose. Parser-solvers that handle more restricted problem types include SDPSOL [69], LMILAB [70], and LMITOOL [71] for SDPs and *linear matrix inequalities* (LMIs), and GGPLAB [72] for generalized geometric programs. More recent examples, which focus on convex optimization, include YALMIP [73], CVX [74], CVXMOD [75], and Pyomo [76]. Some tools [77–79] are used as post-processors, and attempt to detect convexity of a problem expressed in a general purpose modeling language.

As an example, an *instance* of the QP problem (1.2) can be specified in CVXMOD as

```
P = matrix(...); q = matrix(...); A = matrix(...)
b = matrix(...); G = matrix(...); h = matrix(...)
x = optvar('x', n)
qpinst = problem(minimize(0.5*quadform(x, P) + tp(q)*x),
                 [G*x <= h, A*x == b])
```

The first two (only partially shown) lines assign names to specific numeric values, with appropriate dimensions and values. The third line declares x to be an optimization variable of dimension n, which we presume has a fixed numeric value. The last line generates the problem instance itself (but does not solve it), and assigns it the name `qpinst`. This problem instance can then be solved with

```
qpinst.solve()
```

which returns either `'optimal'` or `'infeasible'`, and, if optimal, sets `x.value` to an optimal value $x^\star$.

For specification languages that support parameter declaration, numeric values must be attached to the parameters before the solver is called. For example, the QP problem *family* (1.2) is specified in CVXMOD as

```
A = matrix(...); b = matrix(...)
P = param('P', n, n, psd=True); q = param('q', n)
G = param('G', m, n); h = param('h', m)
x = optvar('x', n)
qpfam = problem(minimize(0.5*quadform(x, P) + tp(q)*x),
                [G*x <= h, A*x == b])
```

In this code segment, as in the example above, m and n are fixed integers. In the first line, A and b are still assigned fixed values, but in the second and third lines, P, q, G, and h are declared instead as parameters with appropriate dimensions. Additionally, P is specified as symmetric positive semidefinite. As before, x is declared to be an optimization variable. In the final line, the QP problem family is constructed (with identical syntax), and assigned the name `qpfam`.

If we called `qpfam.solve()` right away, it would fail, since the parameters have no numeric values. However (with an overloading of semantics), if values are attached to each parameter first, `qpfam.solve()` will create a problem instance and solve that:

```
P.value = matrix(...); q.value = matrix(...)
G.value = matrix(...); h.value = matrix(...)
qpfam.solve()   # Instantiates, then solves.
```

This works since the `solve` method will solve the particular instance of a problem family specified by the numeric values in the value attributes of the parameters.

1.2.5 Code generators

A *code generator* takes a description of a problem family, scans it and checks its validity, carries out various problem transformations, and then generates source code that compiles into a (hopefully very efficient) solver for that problem family. Figures 1.1 and 1.2 show the difference between code generators and parser-solvers.

A code generator will have options configuring the type of code it generates, including, for example, the target language and libraries, the solution algorithm (and algorithm parameters) to use, and the handling of infeasible problem instances. In addition to source code for solving the optimization problem family, the output might also include:

- Auxiliary functions for checking parameter validity, setting up problem instances, preparing a workspace in memory, and cleaning up after problem solution.

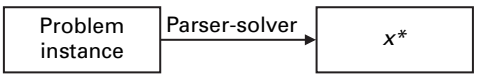

Figure 1.1 A parser-solver processes and solves a single problem instance.

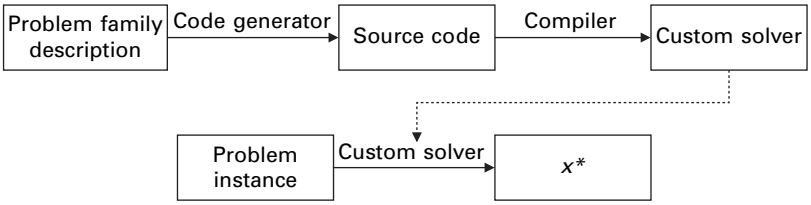

Figure 1.2 A code generator processes a problem family, generating a fast, custom solver, which is used to rapidly solve problem instances.