1 Automatic Code Generation for Real-Time Convex Optimization

Jacob Mattingley and Stephen Boyd

Information Systems Laboratory, Electrical Engineering Department, Stanford University

To appear in *Convex Optimization in Signal Processing and Communications*, Y. Eldar and D. P. Palomar, Eds. Cambridge University Press, 2009.

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.

Contents

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 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 solutions proposed 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 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 size 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 (page 3):

> 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 on 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, *i.e.*, 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 inver-

sion 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, 9, 10, 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; see, *e.g.*, [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 §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 §1.3. In §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 §1.5, briefly describe a preliminary implementation of a code generator in §1.6, and report some numerical results in §1.7. We give a summary and conclusions in §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, 16, 17], or basic model predictive control [18, 19, 20, 21, 22, 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, 28, 29]. Others discuss various real-time applications [30, 31], especially those in robotics [32, 33, 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, 46, 47, 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; see, *e.g.*, [55, 56, 57, 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

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

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

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

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, *i.e.*, 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

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

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. We will refer to the more general problem families as variable dimension problem families.

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 write down 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 [64, 1, 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, *i.e.*, when the solver is invoked. This is extremely useful, since a single solver can handle a very wide class 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; see, *e.g.*, [66, 6].

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 which 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, 78, 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

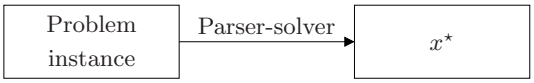

Figure 1.1 A parser-solver processes and solves a single problem instance.

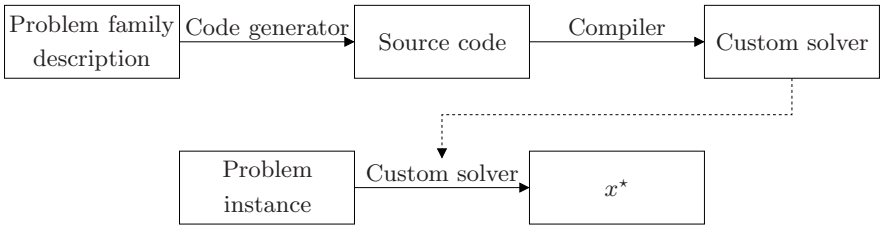

Figure 1.2 A code generator processes a problem family, generating a fast, custom solver, which is used to rapidly solve problem instances.

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 attribute 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.
- Documentation describing the problem family and how to use the code.
- Documentation describing any problem transformations.
- An automated test framework.
- Custom functions for converting problem data to or from a range of formats or environments.
- A system for automatically building and testing the code (such as a `Makefile`).

1.2.6 Example from CVXMOD

In this section we focus on the preliminary code generator in CVXMOD, which generates solver code for the C programming language. To generate code for the problem family described in `qpfam`, we use

```
qpfam.codegen('qpfam/')
```

This tells CVXMOD to generate code and auxiliary files and place them in the `qpfam/` directory. Upon completion, the directory will contain the following files:

- `solver.c`, which includes the actual solver function `solve`, and three initialization functions (`initparams`, `initvars` and `initwork`).
- `template.c`, a simple file illustrating basic usage of the solver and initialization functions.
- `README`, which contains code generation information, including a list of the generated files and information about the targeted problem family.
- `doc.tex`, which provides LaTeX source for a document that describes the problem family, transformations performed to generate the internal standard form, and reference information about how to use the solver.
- `Makefile`, which has rules for compiling and testing the solver.

The file `template.c` contains the following:

```
#include solver.h
int main(int argc, char **argv) {
    Params params = initparams();
    Vars vars = initvars();
    Workspace work = initwork(vars);
    for (;;) { // Real-time loop.
        // Get new parameter values here.
        status = solve(params, vars, work);
        // Test status, export variables, etc here.
    }
```

```
}
```

The main real-time loop (here represented, crudely, as an asynchronous infinite loop) repeatedly carries out the following:

1. Get a new problem instance, *i.e.*, get new parameter values.
2. Solve this instance of the problem, set new values for the variables, and return an indication of solver success (feasibility, infeasibility, failure).
3. Test the status and respond appropriately. If optimization succeeded, export the variables to the particular application.

Some complexity is hidden from the user. For example, the allocated optimization variables include not just the variables specified by the user in the problem specification, but also other, automatically generated intermediate variables, such as slack variables. Similarly, the workspace variables stored within `work` need not concern someone wanting to just get the algorithm working—they are only relevant when the user wants to adjust the various configurable algorithm parameters.

1.3 Examples

In this section we describe several examples of real-time optimization applications. Some we describe in a general setting (*e.g.*, model predictive control); others we decribe in a more specific setting (*e.g.*, optimal order execution). We first list some broad categories of applications, which are not meant to be exclusive or exhaustive.

Real-time adaptation
Real-time optimization is used to optimally allocate multiple resources, as the amounts of resources available, the system requirements or objective, or the system model dynamically change. Here real-time optimization is used to adapt the system to the changes, to maintain optimal performance. In simple adaptation, we ignore any effect the current choice has on future resource availability or requirements. In this case we are simply solving a sequence of independent optimization problem instances, with different data. If the changes in data are modest, warm-start can be used. To be effective, real-time optimization has to be carried out at a rate fast enough to track the changes. Real-time adaptation can be either event-driven (say, whenever the parameters have shifted significantly) or synchronous, with re-optimization occuring at regular time intervals.

Real-time trajectory planning
In trajectory planning we choose a sequence of inputs to a dynamical system that optimizes some objective, while observing some constraints. (This is also called input generation or shaping, or open-loop control.) Typically this is done

asynchronously: A higher level task planner occasionally issues a command such as 'sell this number of shares of this asset over this time period' or 'move the robot end effector to this position at this time'. An optimization problem is then solved, with parameters that depend on the current state of the system, the particular command issued, and other relevant data; the result is a sequence of inputs to the system that will (optimally) carry out the high level command.

Feedback control
In feedback control, real-time optimization is used to determine actions to be taken, based on periodic measurements of some dynamic system, in which current actions *do* affect the future. This task is sometimes divided into two conceptual parts: Optimally sensing or estimating the system state, given the measurements, and choosing an optimal action, based on the estimated system state. (Each of these can be carried out by real-time optimization.) To be effective, the feedback control updates should occur on a time scale at least as fast as the underlying dynamics of the system being controlled. Feedback control is typically synchronous.

Real-time sensing, estimation, or detection
Real-time optimization is used to estimate quantities, or detect events, based on sensor measurements or other periodically-arriving information. In a static system, the quantities to be estimated at each step are independent, so we simply solve an independent problem instance with each new set of measurements. In a dynamic system, the quantities to be estimated are related by some underlying dynamics. In a dynamic system we can have a delay (or look-ahead): We form an estimate of the quantities at time period $t - d$ (where d is the delay), based on measurements up to time period t, or the measurements in some sliding time window.

Real-time system identification
Real-time optimization is used to estimate the parameters in a dynamical model of a system, based on recent measurements of the system outputs (and, possibly, inputs). Here the optimization is used to track changes in the dynamic system; the resulting time-varying dynamic model can in turn be used for prediction, control, or dynamic optimization.

1.3.1 Adaptive filtering and equalization

In adaptive filtering or equalization, a high rate signal is processed in real-time by some (typically linear) operation, parameterized by some coefficients, weights, or gains, that can change with time. The simplest example is a static linear combining filter,

$$y_t = w_t^T u_t,$$

where $u_t \in \mathbf{R}^n$ and $y_t \in \mathbf{R}$ are the vector input and (filtered or equalized) scalar output signals, and $w_t \in \mathbf{R}^n$ is the filter parameter vector, at time $t \in \mathbf{Z}$. The filter parameter w_t is found by solving an (often convex) optimization problem that depends on changing data, such as estimates of noise covariances or channel gains. The filter parameter can be updated (*i.e.*, re-optimized) every step, synchronously every K steps, or asynchronously in an event driven scheme.

When the problem is sufficiently simple, *e.g.*, unconstrained quadratic minimization, the weight updates can be carried out by an analytical method [7, 80, 81]. Subgradient-type or stochastic gradient methods, in which the parameters are updated (usually, slightly) in each step, can also be used [82, 83]. These methods have low update complexity, but only find the optimal weight in the limit of (many) iterations, by which time the data that determined the weight design have already changed. The weight updates could instead be carried out by real-time convex optimization.

To give a specific example, suppose that w_t is chosen to solve the problem

$$\begin{array}{ll} \text{maximize} & w_t^T f_t \\ \text{subject to} & |w_t^T g_t^{(i)}| \leq 1, \quad i = 1, \ldots, m, \end{array}$$

with data f_t, $g_t^{(1)}, \ldots, g_t^{(m)}$. Here f_t is a direction associated with the desired signal, while $g_t^{(i)}$ are directions associated with interference or noise signals. This convex problem can be solved every K steps, say, based on the most recent data available.

1.3.2 Optimal order execution

A sell or buy order, for some number of shares of some asset, is to be executed over a (usually short) time interval, which we divide into T discrete time periods. We have a statistical model of the price in each period, which includes a random component, as well as the effect on the prices due to the amounts sold in the current and previous periods. We may also add constraints, such as a limit on the amount sold per period. The goal is to maximize the expected total revenue from the sale. We can also maximize a variance-adjusted revenue.

In the open-loop version of this problem, we commit to the sales in all periods beforehand. In the closed-loop version, we have recourse: In each period we are told the price (without the current sales impact), and can then adjust the amount we sell. While some forms of this problem have analytical solutions [84, 85], we consider here a more general form.

To give a specific example, suppose that the prices $p = (p_1, \ldots, p_T)$ are modeled as

$$p = p_0 - As,$$

where $s = (s_1, \ldots, s_T)$ are sales, the matrix A (which is lower triangular with nonnegative elements) describes the effect of sales on current and future prices,

and $p_0 \sim \mathcal{N}(\bar{p}, \Sigma)$ is a random price component. The total achieved sales revenue is

$$R = p^T s \sim \mathcal{N}(\bar{p}^T s - s^T A s, s^T \Sigma s).$$

We will choose how to sell $\mathbf{1}^T s = S$ shares, subject to per-period sales limits $0 \leq s \leq S^{\max}$, to maximize the risk-adjusted total revenue,

$$\mathbf{E}\, R - \gamma\, \mathbf{var}\, R = \bar{p}^T s - s^T Q s,$$

where $\gamma > 0$ is a risk aversion parameter, and

$$Q = \gamma\Sigma + (1/2)(A + A^T).$$

(We can assume that $Q \succeq 0$, *i.e.*, Q is positive semidefinite.)

In the open-loop setting, this results in the (convex) QP

$$\begin{array}{ll} \text{maximize} & \bar{p}^T s - s^T Q s \\ \text{subject to} & 0 \leq s \leq S^{\max}, \quad \mathbf{1}^T s = S, \end{array}$$

with variable $s \in \mathbf{R}^T$. The parameters are $\bar{p}$, Q (which depends on the original problem data Σ, A, and γ), $S^{\max}$, and S. An obvious initialization is $s = (S/T)\mathbf{1}$, *i.e.*, constant sales over the time interval.

Real-time optimization for this problem might work as follows. When an order is placed, the problem parameters are determined, and the above QP is solved to find the sales schedule. At least some of these parameters will depend (in part) on the most recently available data; for example, $\bar{p}$, which is a prediction of the mean prices over the next T periods, if no sales occured.

The basic technique in MPC can be used as a very good heuristic for the closed-loop problem. At each time step t, we solve the problem again, using the most recent values of the parameters, and fixing the values of the previous sales $s_1, \ldots, s_{t-1}$ to their (already chosen) values. We then sell the amount s_t from the solution. At the last step no optimization is needed: We simply sell $s_T = S - \sum_{t=1}^{T-1}$, *i.e.*, the remaining unsold shares.

1.3.3 Sliding window smoothing

We are given a noise corrupted scalar signal y_t, $t \in \mathbf{Z}$, and want to form an estimate of the underlying signal, which we denote x_t, $t \in \mathbf{Z}$. We form our estimate $\hat{x}_t$ by examining a window of the corrupted signal, $(y_{t-p}, \ldots, y_{t+q})$, and solving the problem

$$\begin{array}{ll} \text{minimize} & \sum_{\tau=t-p}^{t+q} (y_\tau - \tilde{x}_\tau)^2 + \lambda\phi(\tilde{x}_{t-p}, \ldots, \tilde{x}_{t+q}) \\ \text{subject to} & (\tilde{x}_{t-p}, \ldots, \tilde{x}_{t+q}) \in \mathcal{C}, \end{array}$$

with variables $(\tilde{x}_{t-p}, \ldots, \tilde{x}_{t+q}) \in \mathbf{R}^{p+1+1}$. Here $\phi : \mathbf{R}^{p+q+1} \to \mathbf{R}$ is a (typically convex) function that measures the implausibility of $(\tilde{x}_{t-p}, \ldots, \tilde{x}_{t+q})$, and $\mathcal{C} \subset \mathbf{R}^{p+q+1}$ is a (typically convex) constraint set representing prior information about the signal. The parameter $\lambda > 0$ is used to trade-off fit and implausibility.

The integer $p \geq 0$ is the look-behind length, *i.e.*, how far back in time we look at the corrupted signal in forming our estimate; $q \geq 0$ is the look-ahead length, *i.e.*, how far forward in time we look at the corrupted signal. Our estimate of x_t is $\hat{x}_t = \tilde{x}_t^\star$, where $\tilde{x}^\star$ is a solution of the problem above.

The implausibility function ϕ is often chosen to penalize rapidly varying signals, in which case the estimated signal $\hat{x}$ can be interpreted as a smoothed version of y. One interesting case is $\phi(z) = \sum_{i=1}^{p+q} |z_{t+1} - z_t|$, the total variation of z [86]. Another interesting case is $\phi(z) = \sum_{i=1}^{p+q} |z_{t+1} - 2z_t + z_{t-1}|$, the ℓ_1 norm of the second order difference (or Laplacian); the resulting filter is called an ℓ_1-trend filter [87].

One simple initialization for the problem above is $\tilde{x}_\tau = y_\tau$, $\tau = t-p, \ldots, t+q$; another one is to shift the previous solution in time.

1.3.4 Sliding window estimation

Sliding window estimation, also known as moving horizon estimation (MHE) uses optimization to form an estimate of the state of a dynamical system [88, 21, 89].

A linear dynamical system is modeled as

$$x_{t+1} = Ax_t + w_t,$$

where $x_t \in \mathcal{X} \subset \mathbf{R}^n$ is the state and w_t is a process noise at time period $t \in \mathbf{Z}$. We have linear noise corrupted measurements of the state,

$$y_t = Cx_t + v_t,$$

where $y_t \in \mathbf{R}^p$ is the measured signal and v_t is measurement noise. The goal is to estimate x_t, based on prior information, *i.e.*, A, C, $\mathcal{X}$, and the last T measurements, *i.e.*, $y_{t-T+1}, \ldots, y_t$, along with our estimate of x_{t-T}.

A sliding window estimator chooses the estimate of x_t, which we denote as $\hat{x}_t$, as follows. We first solve the problem

$$\begin{array}{ll} \text{minimize} & \sum_{\tau=t-T+1}^{t} \left(\phi_w(\tilde{x}_\tau - A\tilde{x}_{\tau-1}) + \phi_v(y_\tau - C\tilde{x}_\tau)\right) \\ \text{subject to} & \tilde{x}_{t-T} = \hat{x}_{t-T}, \quad \tilde{x}_\tau \in \mathcal{X}, \quad \tau = t-T+1, \ldots, t, \end{array}$$

with variables $\tilde{x}_{t-T}, \ldots, \tilde{x}_t$. Our estimate is then $\hat{x}_t = \tilde{x}_t^\star$, where $\tilde{x}^\star$ is a solution of the problem above. When $\mathcal{X}$, ϕ_w, and ϕ_v are convex, the problem above is convex.

Several variations of this problem are also used. We can add a cost term associated with $\tilde{x}$, meant to express prior information we have about the state. We can replace the equality constraint $\tilde{x}_{t-T} = \hat{x}_{t-T}$ (which corresponds to the assumption that our estimate of x_{t-T} is perfect) with a cost function term that penalizes deviation of $\tilde{x}_{t-T}$ from $\hat{x}_{t-T}$.

We interpret the cost function term $\phi_w(w)$ as measuring the implausibility of the process noise taking on the value w. Similarly, $\phi_v(v)$ measures the implausibility of the measurement noise taking on the value v. One common choice for these functions is the negative logarithm of the densities of w_t and v_t, respec-

tively, in which case the sliding-window estimate is the maximum likelihood estimate of x_t (assuming the estimate of x_{t-T} was perfect, and the noises w_t are IID, and v_t are IID).

One particular example is $\phi_w(w) = (1/2)\|w\|_2^2$, $\phi_v(v) = (1/2\sigma^2)\|v\|_2^2$, which corresponds to the statistical assumptions $w_t \sim \mathcal{N}(0, I)$, $v_t \sim \mathcal{N}(0, \sigma^2 I)$. We can also use cost functions that give robust estimates, *i.e.*, estimates of x_t that are not greatly affected by occasional large values of w_t and v_t. (These correspond to sudden unexpected changes in the state trajectory, or outliers in the measurements, respectively.) For example, using the (vector) Huber measurement cost function

$$\phi_v(v) = \begin{cases} (1/2)\|v\|_2^2 & \|v\|_2 \leq 1 \\ \|v\|_1 - 1/2 & \|v\|_2 \geq 1 \end{cases}$$

yields state estimates that are surprisingly immune to occasional large values of the measurement noise v_t. (See, *e.g.*, [1, §6.1.2].)

We can initialize the problem above with the previously computed state trajectory, shifted in time, or with one obtained by a linear estimation method, such as Kalman filtering, that ignores the state constraints and, if needed, approximates the cost functions as quadratic.

1.3.5 Real-time input design

We consider a linear dynamical system

$$x_{t+1} = Ax_t + Bu_t,$$

where $x_t \in \mathbf{R}^n$ is the state, and $u_t \in \mathbf{R}^m$ is the control input at time period $t \in \mathbf{Z}$. We are interested in choosing $u_t, \ldots, u_{t+T-1}$, given x_t (the current state) and some convex constraints and objective on $u_t, \ldots, u_{t+T-1}$ and $x_{t+1}, \ldots, x_T$.

As a specific example, we consider minimum norm state transfer to a desired state x^{des}, with input and state bounds. This can be formulated as the QP

$$\begin{array}{ll} \text{minimize} & \sum_{\tau=t}^{T-1} \|u_\tau\|_2^2 \\ \text{subject to} & x_{\tau+1} = Ax_\tau + Bu_\tau, \quad \tau = t, \ldots, t+T-1 \\ & u^{\min} \leq u_\tau \leq u^{\max}, \quad \tau = t, \ldots, t+T-1 \\ & x^{\min} \leq x_\tau \leq x^{\max}, \quad \tau = t, \ldots, t+T, \\ & x_T = x^{\mathrm{des}}, \end{array}$$

with variables $u_t, \ldots, u_{t+T-1}$, $x_t, \ldots, x_{t+T}$. (The inequalities on u_τ and x_τ are componentwise.)

1.3.6 Model predictive control

We consider a linear dynamical system

$$x_{t+1} = Ax_t + Bu_t + w_t, \quad t = 1, 2, \ldots,$$

where $x_t \in \mathbf{R}^n$ is the state, $u_t \in \mathcal{U} \subset \mathbf{R}^m$ is the control input, and $w_t \in \mathbf{R}^n$ is a zero mean random process noise, at time period $t \in \mathbf{Z}_+$. The set $\mathcal{U}$, which is called the input constraint set, is defined by a set of linear inequalities; a typical case is a box,

$$\mathcal{U} = \{v \mid \|v\|_\infty \leq U^{\max}\}.$$

We use a state feedback function (control policy) $\varphi : \mathbf{R}^n \to \mathcal{U}$, with $u(t) = \varphi(x_t)$, so the 'closed-loop' system dynamics are

$$x_{t+1} = Ax_t + B\varphi(x_t) + w_t, \quad t = 1, 2, \ldots.$$

The goal is to choose the control policy φ to minimize the average stage cost, defined as

$$J = \lim_{T\to\infty} \frac{1}{T} \sum_{t=1}^{T} \mathbf{E}\left(x_t^T Q x_t + u_t^T R u_t\right),$$

where $Q \succeq 0$ and $R \succeq 0$. The expectation here is over the process noise.

Model predictive control is a general method for finding a good (if not optimal) control policy. To find $u_t = \varphi^{\mathrm{mpc}}(x_t)$, we first solve the optimization problem

$$\begin{array}{ll} \text{minimize} & \frac{1}{T}\sum_{t=1}^{T}\left(\tilde{z}_t^T Q z_t + v_t^T R v_t\right) + z_{T+1}^T Q_f z_{T+1} \\ \text{subject to} & z_{t+1} = Az_t + Bv_t, \quad t = 1, \ldots, T \\ & v_t \in \mathcal{U}, \quad t = 1, \ldots, T \\ & z_1 = x_t, \end{array} \tag{1.3}$$

with variables $v_1, \ldots, v_T \in \mathbf{R}^m$ and $z_1, \ldots, z_{T+1} \in \mathbf{R}^n$. Here T is called the MPC horizon, and $Q_f \succeq 0$ defines the final state cost. We can interpret the solution to this problem as a plan for the next T time steps, starting from the current state, and ignoring the disturbance. Our control policy is

$$u_t = \varphi^{\mathrm{mpc}}(x_t) = v_1^\star,$$

where $v^\star$ is a solution of the problem (1.3). Roughly speaking, in MPC we compute a *plan of action* for the next T steps, but then execute only the *first control input* from the plan.

The difference between real-time trajectory planning and MPC is *recourse* (or feedback). In real-time trajectory planning an input sequence is chosen, and then executed. In MPC, a trajectory plan is carried out at each step, based on the most current information. In trajectory planning, the system model is deterministic, so no recourse is needed.

One important special case of MPC is when the MPC horizon is $T = 1$, in which case the control policy is

$$u_t = \operatorname*{argmin}_{v\in\mathcal{U}} \left(v^T R v + (Ax_t + Bv)^T Q_f (Ax_t + Bv)\right). \tag{1.4}$$

In this case the control policy is refered to as a control-Lyapunov policy [90, 91].

To evaluate $\varphi(x_t)$, we must solve instances of the QP (1.3) or (1.4). The only parameter in these problem families is x_t; the other problem data (A, B, $\mathcal{U}$, Q, R, Q_f, T) are fixed and known.

There are several useful initializations for the QP (1.3) [6]. One option is to use a linear state feedback gain for an associated unconstrained control problem. Another is to propagate a solution from the previous time step forward.

1.3.7 Optimal network flow rates

This is an example of a resource allocation or resource sharing problem, where the resource to be allocated is the bandwidth over each of a set of links (see, for example, [92, 93], [94, §8]). We consider a network with m edges or links, labeled $1, \ldots, m$, and n flows, labeled $1, \ldots, n$. Each flow has an associated nonnegative flow rate f_j; each edge or link has an associated positive capacity c_i. Each flow passes over a fixed set of links (its route); the total traffic t_i on link i is the sum of the flow rates over all flows that pass through link i. The flow routes are described by a routing matrix $R \in \{0,1\}^{m\times n}$, defined as

$$R_{ij} = \begin{cases} 1 & \text{flow } j \text{ passes through link } i \\ 0 & \text{otherwise.} \end{cases}$$

Thus, the vector of link traffic, $t \in \mathbf{R}^m$, is given by $t = Rf$. The link capacity constraints can be expressed as $Rf \leq c$.

With a given flow vector f, we associate a total utility

$$U(f) = U_1(f_1) + \cdots + U_n(f_n),$$

where U_i is the utility for flow i, which we assume is concave and nondecreasing. We will choose flow rates that maximize total utility, *i.e.*, that are solutions of

$$\begin{array}{l} \text{maximize } U(f) \\ \text{subject to } Rf \leq c, \quad f \geq 0, \end{array}$$

with variable f. This is called the network utility maximization (NUM) problem.

Typical utility functions include linear, with $U_i(f_i) = w_i f_i$, where w_i is a positive constant; logarithmic, with $U_i(f_i) = w_i \log f_i$, and saturated linear, with $U_i(f_i) = w_i \min\{f_i, s_i\}$, w_i a positive weight and s_i a positive satiation level. With saturated linear utilities, there is no reason for any flow to exceed its satiation level, so the NUM problem can be cast as

$$\begin{array}{l} \text{maximize } w^T f \\ \text{subject to } Rf \leq c, \quad 0 \leq f \leq s, \end{array} \tag{1.5}$$

with variable f.

In a real-time setting, we can imagine that R, and the form of each utility function, are fixed; the link capacities and flow utility weights or satiation flow rates change with time. We solve the NUM problem repeatedly, to adapt the flow rates to changes in link capacities or in the utility functions.

Several initializations for (1.5) can be used. One simple one is $f = \alpha \mathbf{1}$, with $\alpha = \min_i c_i/k_i$, where k_i is the number of flows that pass over link i.

1.3.8 Optimal power generation and distribution

This is an example of a single commodity network flow optimization problem. We consider a single commodity network, such as an electrical power network, with n nodes, labeled $1, \ldots, n$, and m directed edges, labeled $1, \ldots, m$. Sources (generators) are connected to a subset $\mathcal{G}$ of the nodes, and sinks (loads) are connected to a subset $\mathcal{L}$ of the nodes. Power can flow along the edges (lines), with a loss that depends on the flow.

We let p_j^{in} denote the (nonnegative) power that enters the tail of edge j; p_j^{out} will denote the (nonnegative) power that emerges from the head of edge j. These are related by

$$p_j^{\text{in}} = p_j^{\text{out}} + \ell_j(p_j^{\text{in}}), \quad j = 1, \ldots, m, \tag{1.6}$$

where $\ell_j(p_j^{\text{in}})$ is the loss on edge j. We assume that ℓ_j is a nonnegative, increasing, and convex function. Each line also has a maximum allowed input power: $p_j^{\text{in}} \leq P_j^{\max}$, $j = 1, \ldots, m$.

At each node the total incoming power, from lines entering the node and a generator, if one is attached to the node, is converted and routed to the outgoing nodes, and to any attached loads. We assume the conversion has an efficiency $\eta_i \in (0, 1]$. Thus we have

$$l_i + \sum_{j \in \mathcal{I}(i)} p_j^{\text{out}} = \eta_i \left(g_i + \sum_{j \in \mathcal{O}(j)} p_j^{\text{in}} \right), \quad i = 1, \ldots, n, \tag{1.7}$$

where l_i is the load power at node i, g_i is the generator input power at node i, $\mathcal{I}(i)$ is the set of incoming edges to node i, and $\mathcal{O}(i)$ is the set of outgoing edges from node i. We take $l_i = 0$ if $i \notin \mathcal{L}$, and $g_i = 0$ if $i \notin \mathcal{G}$.

In the problem of optimal generation and distribution, the node loads l_i are given; the goal is find generator powers $g_i \leq G_i^{\max}$, and line power flows p_i^{in} and p_j^{out}, that minimize the total generating cost, which we take to be a linear function of the powers, $c^T g$. Here c_i is the (positive) cost per watt for generator i. The problem is thus

$$\begin{array}{ll} \text{minimize} & c^T g \\ \text{subject to} & (1.6),\ (1.7) \\ & 0 \leq g \leq G^{\max} \\ & 0 \leq p^{\text{in}} \leq P^{\max}, \quad 0 \leq p^{\text{out}} \end{array}$$

with variables g_i, for $i \in \mathcal{G}$; $p^{\text{in}} \in \mathbf{R}^m$, and $p^{\text{out}} \in \mathbf{R}^m$. (We take $g_i = 0$ for $i \notin \mathcal{G}$.)

Relaxing the line equations (1.6) to the inequalities

$$p_j^{\text{in}} \geq p_j^{\text{out}} + \ell_j(p_j^{\text{in}}), \quad j = 1, \ldots, m,$$

we obtain a convex optimization problem. (It can be shown that every solution of the relaxed problem satisfies the line loss equations (1.6).)

The problem described above is the basic static version of the problem. There are several interesting dynamic versions of the problem. In the simplest, the problem data (*e.g.*, the loads and generation costs) vary with time; in each time period, the optimal generation and power flows are to be determined by solving the static problem. We can add constraints that couple the variables across time periods; for example, we can add a constraint that limits the increase or decrease of each generator power in each time period. We can also add energy storage elements at some nodes, with various inefficiencies, costs, and limits; the resulting problem could be handled by (say) model predictive control.

1.3.9 Processor speed scheduling

We first describe the deterministic finite-horizon version of the problem. We must choose the speed of a processor in each of T time periods, which we denote $s_1, \ldots, s_T$. These must lie between given minimum and maximum values, $s^{\min}$ and $s^{\max}$. The energy consumed by the processor in period t is given by $\phi(s_t)$, where $\phi : \mathbf{R} \to \mathbf{R}$ is increasing and convex. (A very common model, based on simultaneously adjusting the processor voltage with its speed, is quadratic: $\phi(s_t) = \alpha s_t^2$.) The total energy consumed over all the periods is $E = \sum_{t=1}^{T} \phi(s_t)$.

Over the T time periods, the processor must handle a set of n jobs. Each job has an availability time $A_i \in \{1, \ldots, T\}$, and a deadline $D_i \in \{1, \ldots, T\}$, with $D_i \geq A_i$. The processor cannot start work on job i until period $t = A_i$, and must complete the job by the end of period D_i. Each job i involves a (nonnegative) total work W_i.

In period t, the processor allocates its total speed s_t across the n jobs as

$$s_t = S_{t1} + \cdots + S_{tn},$$

where $S_{ti} \geq 0$ is the effective speed the processor devotes to job i during period t. To complete the jobs we must have

$$\sum_{t=A_i}^{D_i} S_{ti} \geq W_i, \quad i = 1, \ldots, n. \tag{1.8}$$

(The optimal allocation will automatically respect the availability and deadline constraints, *i.e.*, satisfy $S_{ti} = 0$ for $t < A_i$ or $t > D_i$.)

We will choose the processor speeds, and job allocations, to minimize the total energy consumed:

$$\begin{array}{ll} \text{minimize} & E = \sum_{t=1}^{T} \phi(s_t) \\ \text{subject to} & s^{\min} \leq s \leq s^{\max}, \quad s = S\mathbf{1}, \quad S \geq 0 \\ & (1.8), \end{array}$$

with variables $s \in \mathbf{R}^T$ and $S \in \mathbf{R}^{T \times n}$. (The inequalities here are all elementwise.)

In the simplest embedded real-time setting, the speeds and allocations are found for consecutive blocks of time, each T periods long, with no jobs spanning two blocks of periods. The speed allocation problem is solved for each block separately; these optimization problems have differing job data (availability time, deadline, and total work).

We can also schedule the speed over a rolling horizon, that extends T periods into the future. At time period t, we schedule processor speed and allocation for the periods $t, t+1, \ldots, t+T$. We interpret n as the maximum number of jobs that can be simultaneously active over such a horizon. Jobs are dynamically added and deleted from the list of active jobs. When a job is finished, it is removed; if a job has already been allocated speed in previous periods, we simply set its availability time to t, and change its required work to be the remaining work to be done. For jobs with deadlines beyond our horizon, we set the deadline to be $t+T$ (the end of our rolling horizon), and linearly interpolate the required work. This gives us a model predictive control method, where we solve the resulting (changing) processor speed and allocation problems in each period, and use the processor speed and allocation corresponding to the current time period. Such a method can dynamically adapt to changing job workloads, new jobs, jobs that are cancelled, or changes in availability and deadlines. This scheme requires the solution of a scheduling problem in each period.

1.4 Algorithm considerations

1.4.1 Requirements

The requirements and desirable features of algorithms for real-time embedded optimization applications differ from those for traditional applications. We first list some important requirements for algorithms used in real-time applications.

Stability and reliability
The algorithm should work well on all, or almost all, $a \in \mathcal{A}$. In contrast, a small failure rate is expected and tolerated in traditional generic algorithms, as a price paid for the ability to efficiently solve a wide range of problems.

Graceful handling of infeasibility
When the particular problem instance is infeasible, or near the feasible-infeasible boundary, a point that is closest to feasible, in some sense, is typically needed. Such points can be found with a traditional Phase I method [1, §11.4], which minimizes the maximum constraint violation, or a sum of constraint violations. In industrial implementations of MPC controllers, for example, the state bound constraints are replaced with what are called soft constraints, *i.e.*, penalties for violating the state constraints that added to the objective function; see, *e.g.*, [21, §3.4]. Another option is to use an infeasible Newton-based method ([1, §10.3]),

in which all iterates satisfy the inequality constraints, but not necessarily the equality constraints, and simply terminate this after a fixed number of steps, whether or not the equality constraints are satisfied [6].

Guaranteed run time bounds
Algorithms used in a real-time setting must be fast, with execution time that is *predictable* and *bounded*. Any algorithm in a real-time loop must have a finite maximum execution time, so results become available in time for the rest of the real-time loop to proceed. Most traditional optimization algorithms have variable run times, since they exit only when certain residuals are small enough.

Another option, that can be useful in synchronous or asynchronous real-time optimization applications, is to employ an *any-time* algorithm, *i.e.*, an algorithm which can be interrupted at any time (after some minimum), and shortly thereafter returns a reasonable approximation of the solution [95, 96].

1.4.2 Exploitable features

On the other hand, real-time applications present us with several features that can work to our advantage, compared to traditional generic applications.

Known (and often modest) accuracy requirements
Most general purpose solvers provide high levels of accuracy, commonly providing optimal values accurate to six or more significant figures. In a real-time setting, such high accuracy is usually unnecessary. For any specific real-time application, the required accuracy is usually known, and typically far smaller than six figures. There are several reasons that high accuracy is often not needed in real-time applications. The variables might represent actions that can be only carried out with some finite fixed resolution (as in a control actuator), so accuracy beyond this resolution is meaningless. As another example, the problem data might be (or come from) physical measurements, which themselves have relatively low accuracy; solving the optimization problem to high accuracy when the data itself has low accuracy is unnecessary. And finally, the model (such as a linear dynamical system model or a statistical model) used to form the real-time optimization problem might not hold to high accuracy, so once again solving the problem to high accuracy is unnecessary.

In many real-time applications, the optimization problem can be solved to low or even very low accuracy, without substantial deterioration in the performance of the overall system. This is especially the case in real-time feedback control, or systems that have recourse, where feedback helps to correct errors from solving previous problem instances inaccurately. For example, Wang and Boyd recently found that, even when the QPs arising in MPC are solved very crudely, high quality control is still achieved [6].

Good initializations are often available
In real-time optimization applications, we often have access to a good initial guess for $x^\star$. In some problems, this comes from a heuristic or approximation specific to the application. For example, in MPC we can initialize the trajectory with one found (quickly) from a classical control method, with a simple projection to ensure the inequality constraints are statisfied. In other real-time optimization applications, the successive problem instances to be solved are near each other, so the optimal point from the last solved problem instance is a good starting point for the current problem instance. MPC provides a good example here, as noted earlier: The most recently computed trajectory can be shifted by one time step, with the boundaries suitably adjusted.

Using a previous solution, or any other good guess, as an initialization for a new problem instance is called *warm starting* [97], and in some cases can dramatically reduce the time required to compute the new solution.

Variable ranges can be determined
A generic solver must work well for data (and solutions) that vary over large ranges of values, that are not typically specified ahead of time. In any particular real-time embedded application, however, we can obtain rather good data about the range of values of variables and parameters. This can be done through simulation of the system, with historical data, or randomly generated data from an appropriate distribution. The knowledge that a variable lies between 0 and 10, for example, can be used to impose (inactive) bounds on it, even when no bounds are present in the original problem statement. Adding bounds like this, which are meant to be inactive, can considerably improve the reliability of, for example, interior-point methods. Other solution methods can use these bounds to tune algorithm parameters.

1.4.3 Interior-point methods

Many methods can be used to solve optimization problems in a real-time setting. For example, Diehl et al [28, 29, 98] have used active set methods for real-time nonlinear MPC. First order methods, such as classical projected gradient methods (see, *e.g.*, [99]), or the more recently developed mirror-descent methods [100], can also be attractive, especially when warm-started, since the accuracy requirements for embedded applications can sometimes be low. The authors have had several successful experiences with interior-point methods. These methods typically require several tens of steps, each of which involves solving a set of equations associated with Newton's method.

Simple primal barrier methods solve a sequence of smooth, equality constrained problems using Newton's method, with a barrier parameter κ that controls the accuracy or duality gap (see, for example, [1, §11] or [101]). For some real-time embedded applications, we can fix the accuracy parameter κ at some suitable value, and limit the number of Newton steps taken. With proper

choice of κ, and warm-start initialization, good application performance can be obtained with just a few Newton steps. This approach is used in [32] to compute optimal robot grasping forces, and in [6] for MPC.

More sophisticated interior-point methods, such as primal-dual methods ([64, §19], [65]) are also very good candidates for real-time embedded applications. These methods can reliably solve problem instances to high accuracy in several tens of steps, but we have found that in many cases, accuracy that is more than adequate for real-time embedded applications is obtained in just a few steps.

1.4.4 Solving systems with KKT-like structure

The dominant effort required in each iteration of an interior-point method is typically the calculation of the search direction, which is found by solving one or two sets of linear equations with KKT (Karush-Kuhn-Tucker) structure:

$$\begin{bmatrix} H & A^T \\ A & 0 \end{bmatrix} \begin{bmatrix} \Delta x \\ \Delta y \end{bmatrix} = \begin{bmatrix} r_1 \\ r_2 \end{bmatrix}. \tag{1.9}$$

Here H is positive semidefinite, A is full rank and fat (*i.e.*, has fewer rows than columns), and Δx and Δy are (or are used to find) the search direction for the primal and dual variables. The data in the KKT system change in each iteration, but the sparsity patterns in H and A are often the same for all iterations, and all problem instances to be solved. This common sparsity pattern derives from the original problem family.

The KKT equations (1.9) can be solved by several general methods. An iterative solver can provide good performance for very large problems, or when extreme speed is needed, but can require substantial tuning of the parameters (see, *e.g.*, [102, 103]). For small and medium size problems, though, we can employ a direct method, such as LDL^T ('signed Cholesky') factorization, possibly using block elimination [104]. We find a permutation P (also called an elimination ordering or pivot sequence), a lower triangular matrix L, and a diagonal matrix D (both invertible) such that

$$P \begin{bmatrix} H & A^T \\ A & 0 \end{bmatrix} P^T = LDL^T. \tag{1.10}$$

Once the LDL^T factorization has been found, we use backward and forward elimination to solve the KKT system (1.9) [1, §C.4.2], [105]. The overall effort required depends on the sparsity pattern of L; more specifically, the number of nonzero entries. This number is always at least as large as the number of nonzero entries in the lower triangular part of the KKT matrix; additional nonzero entries in L, that are not in the KKT matrix, are called fill-in entries, and the number of fill-in entries is referred to as the fill-in. The smaller the fill-in, the more efficiently the KKT system can be solved.

The permutation P is chosen to reduce fill-in, while avoiding numerical instability (such as dividing by a very small number) in the factorization, *i.e.*, the

computation of L and D. In an extreme case, we can encounter a divide-by-zero in our attempt to compute the LDL^T factorization (1.10), which means that such a factorization does not exist for that particular KKT matrix instance, and that choice of permutation. (The factorization exists if and only if every leading principal submatrix of the permuted KKT matrix is nonsingular.)

Static pivoting or *symbolic permutation* refers to the case when P is chosen based only the sparsity pattern of the KKT matrix. In constrast, *dynamic pivoting* refers to the case when P is chosen in part based on the numeric values in the partially factorized matrix. Most general purpose sparse equation solvers use dynamic pivoting; static pivoting is used in some special cases, such as when H is positive definite and A is absent. For real-time embedded applications, static pivoting has several advantages. It results in a simple algorithm with no conditionals, which allows us to bound the run-time (and memory requirements) reliably, and allows much compiler optimization (since the algorithm is branch free). So we proceed assuming that static permutation will be employed. In other words, we will choose one permutation P and use it to solve the KKT system arising in each interior-point iteration in each problem instance to be solved.

Methods for choosing P, based on the sparsity pattern in the KKT matrix, generally use a heuristic for minimizing fill-in, while guaranteeing that the LDL^T factorization exists. KKT matrices have special structure, which may be exploited when selecting the permutation [106, 107, 108]. A recent example is the `KKTDirect` package, developed by Bridson [109], which chooses a permutation that guarantees existence of the LDL^T factorization, provided H is positive definite and A is full rank, and tends to achieve low fill-in. Other methods include approximate minimum degree ordering [110], or METIS [111], which may be applied to the positive definite portion of the KKT matrix, and again after a block reduction. While existence of the factorization does not guarantee numerical stability, it has been observed in practice. (Additional methods, described below, can be used to guard against numerical instability.)

One pathology that can occur is when H is singular (but still positive semidefinite). One solution is to solve the (equivalent) linear equations

$$\begin{bmatrix} H + A^TQA & A^T \\ A & 0 \end{bmatrix} \begin{bmatrix} \Delta x \\ \Delta y \end{bmatrix} = \begin{bmatrix} r_1 + A^TQr_2 \\ r_2 \end{bmatrix},$$

where Q is any positive semidefinite matrix for which $H + A^TQA$ is positive definite [1, §10.4.2]. The key here is to choose Q (if possible) so that the number of nonzero entries in $H + A^TQA$ is not too much more than the number of nonzero entries in A.

Some standard tricks used in optimization computations can also be used in the context of real-time embedded applications. One is to dynamically add diagonal elements to the KKT matrix, during factorization, to avoid division by small numbers (including zero); see, *e.g.*, [112]. In this case we end up with the factorization of a matrix that is close to, but not the same as, the KKT matrix; the search direction computed using this approximate factorization is

only an approximate solution of the original KKT system. One option is to simply use the resulting approximate search direction in the interior-point method, as if it were the exact search direction [65, §11]. Another option is to use a few steps of iterative refinement, using the exact KKT matrix and the approximate factorization [113].

1.5 Code generation

1.5.1 Custom KKT solver

To generate a custom solver based on an interior-point method, we start by generating code to carry out the LDL^T factorization. The most important point here is that the sparsity pattern of L can be determined symbolically, from the sparsity patterns of H and A (which, in turn, derive from the structure of the original problem family), and the permutation matrix P. In particular, the sparsity pattern of L, as well as the exact list of operations to be carried out in the factorization, are known at code generation time. Thus, at code generation time, we can carry out the following tasks.

1. *Choose the permutation P.* This is done to (approximately) minimize fill-in while ensuring existence of the factorization. Considerable effort can be expended in this task, since it is done at code generation time.
2. *Determine storage schemes.* Once the permuation is fixed, we can choose a storage scheme for the permuted KKT matrix (if we in fact form it explicitly), and its factor L.
3. *Generate code.* We can now generate code to perform the following tasks.
   - Fill the entries of the permuted KKT matrix, from the parameter a and the current primal and dual variables.
   - Factor the permuted KKT matrix, *i.e.*, compute the values of L and D.
   - Solve the permuted KKT system, by backward and forward substitution.

Thus, we generate custom code that quickly solves the KKT system (1.9). Note that once the code is generated, we know the exact number of floating point operations required to solve the KKT system.

1.5.2 Additional optimizations

The biggest reduction in solution time comes from careful choice of algorithm, problem transformations, and permutations used in computing the search directions. Together, these fix the number of floating point operations (flops) that a solver requires. Floating point arithmetic is typically computationally expensive, even when dedicated hardware is available.

A secondary goal, then, is to maximize utilization of a computer's floating point unit or units. This is a standard code optimization task, some parts of which a

good code generator can perform automatically. Here are some techniques that may be worth exploring. For general information about compilers, see, *e.g.*, [114].

- *Memory optimization.* We want to store parameter and working problem data as efficiently as possible in memory, to access it quickly, maximize locality of reference and minimize total memory used. The code generator should choose the most appropriate scheme ahead of time.
  Traditional methods for working with sparse matrices, for example in packages like UMFPACK [115] and CHOLMOD [116], require indices to be stored along with problem data. At code generation time we already know all locations of nonzero entries, so we have the choice of removing explicit indices, designing a specific storage structure and then referring to the nonzero entries directly.
- *Unrolling code.* Similar to unrolling loops, we can 'unroll' factorizations and multiplies. For example, when multiplying two sparse matrices, one option is to write each (scalar) operation explicitly in code. This eliminates loops and branches to make fast, linear code, but also makes the code more verbose.
- *Caching arithmetic results.* If certain intermediate arithmetic results are used multiple times, it may be worth trading additional storage for reduced floating point computations.
- *Re-ordering arithmetic.* As long as the algorithm remains mathematically correct, it may be helpful to re-order arithmetic instructions to better use caches. It may also be worth replacing costly divides (say) with additional multiplies.
- *Targeting specific hardware.* Targeting specific hardware features may allow performance gains. Some processors may include particular instruction sets (like SSE [117], which allows faster floating point operations). This typically requires carefully arranged memory access patterns, which the code generator may be able to provide. Using more exotic hardware is possible too; graphics processors allow high speed parallel operations [118], and some recent work has investigated using FPGAs for MPC [119, 120].
- *Parallelism.* Interior-point algorithms offer many opportunities for parallelism, especially because all instructions can be scheduled at code generation time. A powerful code generator may be able to generate parallel code to efficiently use multiple cores or other parallel hardware.
- *Empirical optimization.* At the expense of extra code generation time, a code generator may have the opportunity to use empirical optimization to find the best of multiple options.
- *Compiler selection.* A good compiler with carefully chosen optimization flags can significantly improve the speed of program code.

We emphasize that all of this optimization can take place at code generation time, and thus, can be done in relative leisure. In a real-time optimization setting, longer code generation and compile times can be tolerated, especially when the benefit is solver code that runs very fast.

1.6 CVXMOD: a preliminary implementation

We have implemented a code generator, within the CVXMOD framework, to test some of these ideas. It is a work in progress; we report here only a preliminary implementation. It can handle any problem family that is representable via disciplined convex programming [121, 122, 123] as a QP (including, in particular, LP). Problems are expressed naturally in CVXMOD, using QP-representable functions such as `min`, `max`, `norm1`, and `norminf`.

A wide variety of algorithms can be used to solve QPs. We briefly describe here the particular primal-dual method we use in our current implementation. While it has excellent performance (as we will see from the experimental results), we do not claim that it is any better than other, similar methods.

1.6.1 Algorithm

CVXMOD begins by transforming the given problem into the standard QP form (1.2). The optimization variable therein includes the optimization variables in the original problem, and possibly other, automatically introduced variables. Code is then prepared for a Mehrotra predictor-corrector primal-dual interior point method [124].

We start by introducing a slack variable $s \in \mathbf{R}^m$, which results in the problem

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

with (primal) variables x and s. Introducing dual variables $y \in \mathbf{R}^n$ and $z \in \mathbf{R}^m$, and defining $X = \mathbf{diag}(x)$ and $S = \mathbf{diag}(s)$, the KKT optimality conditions for this problem are

$$\begin{array}{l} Px + q + G^T z + A^T y = 0 \\ Gx + s = h, \quad Ax = b \\ s \geq 0, \quad z \geq 0 \\ ZS = 0. \end{array}$$

The first and second lines (which are linear equations) correspond to dual and primal feasibility, respectively. The third gives the inequality constraints, and the last line (a set of nonlinear equations) is the complementary slackness condition.

In a primal-dual interior-point method, the optimality conditions are solved by a modified Newton method, maintaining strictly positive s and z (including by appropriate choice of step length), and linearizing the complementary slackness condition at each step. The linearized equations are

$$\begin{bmatrix} P & 0 & G^T & A^T \\ 0 & Z & S & 0 \\ G & I & 0 & 0 \\ A & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} \Delta x^{(i)} \\ \Delta s^{(i)} \\ \Delta z^{(i)} \\ \Delta y^{(i)} \end{bmatrix} = \begin{bmatrix} r_1^{(i)} \\ r_2^{(i)} \\ r_3^{(i)} \\ r_4^{(i)} \end{bmatrix},$$

which, in a Mehrotra predictor-corrector scheme, we need to solve with two different right-hand sides [124]. This system of linear equations is nonsymmetric, but can be put in standard KKT form (1.9) by a simple scaling of variables:

$$\left[\begin{array}{cc|cc} P & 0 & G^T & A^T \\ 0 & S^{-1}Z & I & 0 \\ \hline G & I & 0 & 0 \\ A & 0 & 0 & 0 \end{array}\right] \left[\begin{array}{c} \Delta x^{(i)} \\ \Delta s^{(i)} \\ \Delta z^{(i)} \\ \Delta y^{(i)} \end{array}\right] = \left[\begin{array}{c} r_1^{(i)} \\ S^{-1}r_2^{(i)} \\ r_3^{(i)} \\ r_4^{(i)} \end{array}\right].$$

The (1,1) block here is positive semidefinite.

The current implementation of CVXMOD performs two steps of block elimination on this block 4×4 system of equations, which results in a reduced block 2×2 system, also of KKT form. We determine a permutation P for the reduced system using `KKTDirect` [109].

The remainder of the implementation of the primal-dual algorithm is straightforward, and is described elsewhere [124, 65, 64]. Significant performance improvements are achieved by using many of the additional optimizations described in §1.5.2.

1.7 Numerical examples

To give a rough idea of the performance achieved by our preliminary code generation implementation, we conducted numerical experiments. These were performed on an unloaded Intel Core Duo 1.7 GHz, with 2 GB of RAM and Debian GNU Linux 2.6. The results for several different examples are given below.

1.7.1 Model predictive control

We consider a model predictive control problem as described in §1.3.6, with state dimension $n = 10$, $m = 3$ actuators, and horizon $T = 10$. We generate A and B with $\mathcal{N}(0, 1)$ entries, and then scale A so that its spectral radius is one (which makes the control challenging, and therefore interesting). The input constraint set $\mathcal{U}$ is a box, with $U^{\max} = 0.15$. (This causes about half of the control inputs to saturate.) Our objective is defined by $Q = I$, $R = I$. We take $Q_f = 0$, instead adding the constraint that $x(10) = 0$. All of these data are constants in the problem family (1.3); the only parameter is $x(1)$, the initial state. We generate 10 000 instances of the problem family by choosing the components of $x(1)$ from a (uniform) $\mathcal{U}[-1, 1]$ distribution.

The resulting QP (1.3), both before and after transformation to CVXMOD's standard form, has 140 variables, 120 equality, and 70 inequality constraints. The lower triangular portion of the KKT matrix has 1740 nonzeros; after ordering, the factor L has 3140 nonzeros. (This represents a fill-in factor of just under two).

Table 1.1. Model predictive control. Performance results for 10 000 solves.

Original problem		**Transformed problem**		**Performance** (per solve)	
Variables	140	n	140	Step limit	4
Parameters	140	p	120	Steps (avg)	3.3
Equalities	120	m	70	Final gap (avg)	0.9%
Inequalities	60	$\mathbf{nnz}(KKT)$	1740	Time (avg)	425 μs
		$\mathbf{nnz}(L)$	3140	Time (max)	515 μs

We set the maximum number of iterations to be four, terminating early if sufficient accuracy is attained in fewer steps. (The level of accuracy obtained is more than adequate to provide excellent control performance; see, *e.g.*, [6].) The performance results are summarized in Table 1.1.

The resulting QP may be solved at well over 1000 times per second, meaning that MPC can run at over 1 kHz. (Kilohertz rates can be obtained using explicit MPC methods, but this problem is too large for that approach.)

1.7.2 Optimal order execution

We consider the open-loop optimal execution problem described in §1.3.2. We use a simple affine model for the mean price trajectory. We fix the mean starting price $\bar{p}_1$ and set

$$\bar{p}_i = \bar{p}_1 + (d\bar{p}_1/(T-1))(i-1), \quad i = 2, \ldots, T,$$

where d is a parameter (the price drift). The final mean price $\bar{p}_T$ is a factor $1+d$ times the mean starting price.

We model price variation as a random walk, parameterized by the single-step variance σ^2/T. This corresponds to the covariance matrix with

$$\Sigma_{ij} = (\sigma^2/T)\min(i,j), \quad i,j = 1, \ldots, T.$$

The standard deviation of the final price is σ.

We model the effect of sales on prices with

$$A_{ij} = \begin{cases} (\alpha\bar{p}_1/S_{\max})e^{(j-i)/\beta} & i \geq j \\ 0 & i < j, \end{cases}$$

where α and β are parameters. The parameter α gives the immediate decrease in price, relative to the initial mean price, when we sell the maximum number of shares, $S_{\max}$. The parameter β, which has units of periods, determines (roughly) the number of periods over which a sale impacts prices: After around β periods, the price impact is around $1/e \approx 38\%$ of the initial impact.

To test the performance of the generated code, we generate 1 000 problem instances, for a problem family with $T = 20$ periods. We fix the starting price as $\bar{p}_1 = 10$, the risk aversion parameter as $\gamma = 0.5$, the final price standard deviation

Table 1.2. Optimal execution problem. Performance results for 1 000 solves.

Original problem		**Transformed problem**		**Performance** (per solve)	
Variables	20	n	20	Step limit	4
Parameters	232	p	1	Steps (avg)	3.0
Equalities	1	m	40	Final gap (avg)	0.05%
Inequalities	40	**nnz**(KKT)	231	Time (avg)	49 μs
		nnz(L)	231	Time (max)	65 μs

as $\sigma = 4$, and the maximum shares sold per period as $S_{\max} = 10\,000$. For each problem instance we randomly generate parameters as follows. We take $d \sim \mathcal{U}[-0.3, 0.3]$ (representing a mean price movement between 30% decrease and 30% increase) and choose $S \sim \mathcal{U}[0, 10\,000]$. We take $\alpha \sim \mathcal{U}[0.05, 0.2]$ (meaning an immediate price impact for the maximum sale ranges between 5% and 20%), and $\beta \sim \mathcal{U}[1, 10]$ (meaning the price increase effect persists between 1 and 10 periods).

CVXMOD transforms the original problem, which has 20 variables, one equality constraint, and 20 inequality constraints, into a standard form problem with 20 variables and one equality constraint. The lower triangular portion of the KKT matrix has a total of 231 nonzero entries. After ordering, L also has 231 entries, so there is no fill-in.

We fix the maximum number of iterations at four, terminating earlier if the duality gap is less than 500. (The average objective value is around 250 000, so this represents a required accuracy around 0.2%.) The performance results are summarized in Table 1.2. We can see that well over 100 000 problem instances can be solved in one second.

We should mention that the solve speed obtained is far faster than what is needed in any real-time implementation. One practical use of the very fast solve time is for Monte Carlo simulation, which might be used to test the optimal execution algorithm, tune various model parameters (*e.g.*, A), and so on.

1.7.3 Optimal network flow rates

We consider the NUM problem with satiation (1.5). We choose (and fix) a random routing matrix $R \in \{0, 1\}^{50\times 50}$ by setting three randomly chosen entries in each column to 1. This corresponds to a network with 50 flows and 50 links, with each flow passing over 3 links.

CVXMOD transforms the original problem, which has 50 scalar variables and 150 inequality constraints, into a problem in standard form, which also has 50 scalar variables and 150 inequality constraints. The lower triangular portion of the KKT matrix has a total of 244 nonzero entries. After ordering, L has 796 nonzero entries. This corresponds to a fill-in factor of a little over three.

Table 1.3. NUM problem. Performance results for 100 000 solves.

Original problem		**Transformed problem**		**Performance** (per solve)	
Variables	50	n	50	Step limit	6
Parameters	300	p	0	Steps (avg)	5.7
Equalities	0	m	150	Final gap (avg)	0.8%
Inequalities	150	**nnz**(KKT)	244	Time (avg)	230 μs
		nnz(L)	496	Time (max)	245 μs

To test the performance of the generated code, we generate 100 000 problem instances, with random parameters, chosen as follows. The weights are uniform on $[1, 5]$, the satiation levels are uniform on $[5\,000, 15\,000]$, and the link capacities are uniform on $[10\,000, 30\,000]$.

We fix a step limit of six, terminating earlier if sufficient accuracy is attained (in this case, if the duality gap passes below 2000, which typically represents about 1.5% accuracy). The performance results are summarized in Table 1.3.

We can see that this optimal flow problem can be solved several thousand times per second, making it capable of responding to changing link capacities, satiation levels, or flow utilities on the millisecond level. As far as we know, flow control on networks is not currently done by explicitly solving optimization problems; instead, it is carried out using protocols that adjust rates based on a simple feedback mechanism, using the number of lost packets, or round-trip delay times, for each flow. Our high solver speed suggests that, in some cases, flow control could be done by explicit solution of an optimization problem.

1.7.4 Real-time actuator optimization

A rigid body has n actuators, each of which applies a force (or torque) at a particular position, in a particular direction. These forces and torques must lie in given intervals. For example, a thruster, or a force transfered by tension in a cable, can only apply a nonnegative force, which corresponds to a lower limit of zero. The bounds can also represent limits on the magnitude of the force or torque for each actuator. The forces and torques yield a net force ($\in \mathbf{R}^3$) and net moment ($\in \mathbf{R}^3$) that are linear functions of the actuator values. The goal is to achieve a given desired net force and moment with minimum actuator cost.

Taking the lower limit to be zero (*i.e.*, all forces must be nonnegative), and a linear cost function, we have the problem

$$\begin{array}{ll} \text{minimize} & c^T f \\ \text{subject to} & F^{\text{des}} = Af, \quad \Omega^{\text{des}} = Bf, \\ & 0 \leq f \leq F^{\max}, \end{array}$$

with variable $f \in \mathbf{R}^n$. Here $c \in \mathbf{R}^n$ defines the cost, and $A \in \mathbf{R}^{3\times n}$ ($B \in \mathbf{R}^{3\times n}$) relates the applied forces (moments) to the net force (moment). (These matrices

Table 1.4. Actuator problem. Performance results for 100 000 solves.

Original problem		**Transformed problem**		**Performance** (per solve)	
Variables	50	n	50	Step limit	7
Parameters	300	p	6	Steps (avg)	6.4
Equalities	6	m	100	Final gap (avg)	0.4%
Inequalities	100	**nnz**(KKT)	356	Time (avg)	170 μs
		nnz(L)	1317	Time (max)	190 μs

depend on the location and direction of the actuator forces.) The problem data are A, B, F^{des}, and Ω^{des}.

This actuator optimization problem can be used to determine the necessary actuator signals in real-time. A high level control algorithm determines the desired net force and moment to be applied at each sampling interval, and the problem above is solved to determine how best the actuators should achieve the required net values.

To test the performance of the generated code, we solve 100 000 problem instances. At each instance we choose an $A, B \in \mathbf{R}^{3\times 100}$ with entries uniform on $[-2.5, 2.5]$, set the entries of c uniform on $[0.1, 1]$, the entries of F^{net} and Ω^{net} uniform on $[-5, 5]$ and $F^{\mathrm{max}} = 1$.

We fix a step limit of seven, terminating earlier if sufficient accuracy is attained (in this case, if the duality gap passes below 0.007, which typically represents about 1% accuracy). The performance results are summarized in Table 1.4.

We see that the problem instances can be solved at a rate of several thousand per second, which means that this actuator optimization method can be embedded in a system running at several kHz.

1.8 Summary, conclusions, and implications

In real-time embedded optimization we must solve many instances of an optimization problem from a given family, often at regular time intervals. A generic solver can be used if the time intervals are long enough, and the problems small enough. But for many interesting applications, particularly those in which problem instances must be solved in milliseconds (or faster), a generic solver is not fast enough. In these cases a custom solver can always be developed 'by hand', but this requires much time and expertise.

We propose that code generation should be used to rapidly generate source code for custom solvers for specific problem families. Much optimization of the algorithm (for example the ordering of variable elimination), can be carried out automatically during code generation. While code generation and subsequent compilation can be slow, the resulting custom solver is very fast, and has a well defined maximum run-time, which makes it suitable for real-time applications.

We have implemented a basic code generator for convex optimization problems that can be transformed to QPs, and demonstrated that extremely fast (worst-case) execution times can be reliably achieved. On a 1.7 GHz processor, the generated code solves problem instances with tens, or even more than a hundred variables, in times measured in microseconds. (We believe that these times can be further reduced by various improvements in the code generation.) These are execution times several orders of magnitude faster than those achieved by generic solvers.

There are several other uses for very fast custom solvers, even in cases where the raw speed is not needed in the real-time application. One obvious example is simulation, where many instances of the problem must be solved. Suppose for example that a real-time trading system requires solution of a QP instance each second, which (assuming the problem size is modest) is easily achieved with a generic QP solver. If a custom solver can solve this QP in 100 μs, we can carry out simulations of the trading system 10 000 times faster than real-time. This makes it easier to judge performance and tune algorithm parameters.

Our main interest is in real-time embedded applications, in which problem instances are solved repeatedly, and very quickly. An immediate application is MPC, a well developed real-time control method that relies on the solution of a QP in each time step. Until now, however, general MPC was mostly considered practical only for 'slow' systems, in which the time steps are long, say, seconds or minutes. (One obvious exception is the recent development of explicit MPC, which is limited to systems to with a small number of states, actuators, and constraints, and a short horizon.) We believe that MPC should be much more widely used than it currently is, especially in applications with update times measured in milliseconds.

We can think of many other potential applications of real-time embedded optimization, in areas such as robotics, signal processing, and machine learning, to name just a few. Many of the real-time methods in wide use today are simple, requiring only a few matrix-vector (or even vector-vector) operations at run-time. The parameters or weights, however, are calculated with considerable effort, often by optimization, and off-line. We suspect that methods that solve optimization problems on-line can out-perform methods that use only simple calculations on-line. (This is certainly the case in control, which suggests that it should be the case in other application areas as well.)

There might seem to be a clear dividing line between traditional real-time control and signal processing methods, which rely on simple calculations in each step, and 'computational' methods, that carry out what appear to be more complex calculations in each step, such as solving a QP. The classical example here is the distinction between a classical feedback control law, which requires a handful of matrix-matrix and matrix-vector operations in each step, and MPC, which requires the solution of a QP in each time step. We argue that no such clean dividing line exists: We can often solve a QP (well enough to obtain good performance) in a time comparable to that required to compute a classical control law,

and, in any case, fast enough for a wide variety of applications. We think that a smearing of the boundaries between real-time optimization and control will occur, with many applications benefitting from solution of optimization problems in real-time.

We should make a comment concerning the particular choices we made in the implementation of our prototype code generator. These should be interpreted merely as incidental selections, and not as an assertion that these are the best choices for all applications. In particular, we do not claim that primal-dual interior-point methods are better than active set or first order methods; we do not claim that dynamic pivoting should always be avoided; and we do not claim that using an LDL^T factorization of the reduced KKT system is the best way to solve for the search direction. We do claim that these choices appear to be good enough to result in very fast, and very reliable solvers, suitable for use in embedded real-time optimization applications.

Finally, we mention that our preliminary code generator, and numerical examples, focus on relatively small problems, with tens of variables, and very fast solve times (measured in microseconds). But many of the same ideas apply to much larger problems, say with thousands of variables. Custom code automatically generated for such a problem family would be much faster than a generic solver.

Acknowledgments

We are grateful to several people for very helpful discussions, including Arkadi Nemirovsky, Eric Feron, Yang Wang and Behcet Acikmese.

The research reported here was supported in part by NSF award 0529426, AFOSR award FA9550-06-1-0514, and by DARPA contract N66001-08-1-2066. Jacob Mattingley was supported in part by a Lucent Technologies Stanford Graduate Fellowship.

References

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

[2] G. F. Franklin, J. D. Powell, and A. Emami-Naeini, *Feedback Control of Dynamic Systems*. Prentice Hall, 1991.

[3] G. F. Franklin, M. L. Workman, and D. Powell, *Digital Control of Dynamic Systems*. Addison-Wesley, 1997.

[4] S. J. Qin and T. A. Badgwell, "A survey of industrial model predictive control technology," *Control Engineering Practice*, vol. 11, no. 7, pp. 733–764, 2003.

[5] A. Bemporad and C. Filippi, "Suboptimal explicit receding horizon control via approximate multiparametric quadratic programming," *Journal of Optimization Theory and Applications*, vol. 117, no. 1, pp. 9–38, Nov. 2004.

[6] Y. Wang and S. Boyd, "Fast model predictive control using online optimization," in *Proceedings IFAC World Congress*, Jul. 2008, pp. 6974–6997.

[7] A. H. Sayed, *Fundamentals of Adaptive Filtering*. IEEE Press, 2003.

[8] E. J. Candès and T. Tao, "Decoding by linear programming," *IEEE Transactions on Information Theory*, vol. 51, no. 12, pp. 4203–4215, 2005.

[9] J. Feldman, D. R. Karger, and M. J. Wainwright, "LP decoding," in *Proceedings, Annual Allerton Conference on Communication Control and Computing*, vol. 41, no. 2, 2003, pp. 951–960.

[10] J. Feldman, "Decoding error-correcting codes via linear programming," Ph.D. dissertation, Massachusetts Institute of Technology, 2003.

[11] J. Jalden, C. Martin, and B. Ottersten, "Semidefinite programming for detection in linear systems—optimality conditions and space-time decoding," *IEEE International Conference on Acoustics, Speech, and Signal Processing*, vol. 4, 2003.

[12] M. Kisialiou and Z.-Q. Luo, "Performance analysis of quasi-maximum-likelihood detector based on semi-definite programming," *IEEE International Conference on Acoustics, Speech, and Signal Processing*, vol. 3, pp. 433–436, 2005.

[13] S. Joshi and S. Boyd, "Sensor selection via convex optimization," *To appear, IEEE Transactions on Signal Processing*, 2008.

[14] A. Zymnis, S. Boyd, and D. Gorinevsky, "Relaxed maximum a posteriori fault identification," Available online: http://www.stanford.edu/~boyd/

papers/fault_det.html, May 2008.

[15] I. M. Ross and F. Fahroo, "Issues in the real-time computation of optimal control," *Mathematical and Computer Modelling*, vol. 43, no. 9-10, pp. 1172–1188, 2006.

[16] G. C. Goodwin, M. Seron, and J. D. Dona, *Constrained Control and Estimation: An Optimisation Approach*. Springer, 2005.

[17] B. Bäuml and G. Hirzinger, "When hard realtime matters: Software for complex mechatronic systems," *Journal of Robotics and Autonomous Systems*, vol. 56, pp. 5–13, 2008.

[18] E. F. Camacho and C. Bordons, *Model Predictive Control*. Springer, 2004.

[19] D. E. Seborg, T. F. Edgar, and D. A. Mellichamp, *Process dynamics and control*. Wiley New York, 1989.

[20] F. Allgöwer and A. Zheng, *Nonlinear Model Predictive Control*. Birkhauser, 2000.

[21] J. M. Maciejowski, *Predictive Control: With Constraints*. Prentice Hall, 2002.

[22] I. Dzafic, S. Tesnjak, and M. Glavic, "Automatic object-oriented code generation to power system on-line optimization and analysis," in *21st IASTED International Conference on Modeling, Identification and Control*, 2002.

[23] R. Soeterboek, *Predictive Control: A Unified Approach*. Prentice Hall, 1992.

[24] A. Bemporad, M. Morari, V. Dua, and E. N. Pistikopoulos, "The explicit linear quadratic regulator for constrained systems," *Automatica*, vol. 38, no. 1, pp. 3–20, 2002.

[25] C. V. Rao, S. J. Wright, and J. B. Rawlings, "Application of interior-point methods to model predictive control," *Journal of Optimization Theory and Applications*, vol. 99, no. 3, pp. 723–757, 1998.

[26] Y. Wang and S. Boyd, "Performance bounds for linear stochastic control," Working manuscript, Aug. 2008.

[27] V. M. Zavala and L. T. Biegler, "The advanced-step NMPC controller: optimality, stability and robustness," *Automatica*, 2007, submitted.

[28] M. Diehl, H. G. Bock, and J. P. Schlöder, "A real-time iteration scheme for nonlinear optimization in optimal feedback control," *SIAM Journal on control and optimization*, vol. 43, no. 5, pp. 1714–1736, 2005.

[29] M. Diehl, R. Findeisen, F. Allgöwer, H. G. Bock, and J. P. Schlöder, "Nominal stability of the real-time iteration scheme for nonlinear model predictive control," *Proceedings Control Theory Applications*, vol. 152, no. 3, pp. 296–308, May 2005.

[30] E. Frazzoli, Z. H. Mao, J. H. Oh, and E. Feron, "Aircraft conflict resolution via semidefinite programming," *AIAA Journal of Guidance, Control, and Dynamics*, vol. 24, no. 1, pp. 79–86, 2001.

[31] T. Ohtsuka and H. A. Fujii, "Nonlinear receding-horizon state estimation by real-time optimization technique," *Journal of Guidance, Control, and Dynamics*, vol. 19, no. 4, pp. 863–870, 1996.

[32] S. P. Boyd and B. Wegbreit, "Fast computation of optimal contact forces," *IEEE Transactions on Robotics*, vol. 23, no. 6, pp. 1117–1132, 2007.

[33] D. Verscheure, B. Demeulenaere, J. Swevers, J. De Schutter, and M. Diehl, "Practical time-optimal trajectory planning for robots: a convex optimization approach," *IEEE Transactions on Automatic Control*, 2008, submitted.

[34] J. Zhao, M. Diehl, R. W. Longman, H. G. Bock, and J. P. Schloder, "Nonlinear model predictive control of robots using real-time optimization," *Advances in the Astronautical Sciences*, 2004.

[35] E. J. Candes, M. B. Wakin, and S. Boyd, "Enhancing sparsity by reweighted l1 minimization," *Journal of Fourier Analysis and Applications*, 2008.

[36] J. A. Tropp, "Just relax: convex programming methods for identifying sparse signals in noise," *IEEE Transactions on Information Theory*, vol. 52, no. 3, pp. 1030–1051, 2006.

[37] E. Candès, M. Rudelson, T. Tao, and R. Vershynin, "Error correction via linear programming," *Annual symposium on foundations of computer science*, vol. 46, p. 295, 2005.

[38] D. Goldfarb and W. Yin, "Second-order cone programming methods for total variation-based image restoration," *SIAM Journal on Scientific Computing*, vol. 27, no. 2, pp. 622–645, 2005.

[39] A. Zymnis, S. Boyd, and D. Gorinevsky, "Mixed state estimation for a linear Gaussian Markov model," *To appear, IEEE Conference on Decision and Control*, Dec. 2008.

[40] K. Collins-Thompson, "Estimating robust query models with convex optimization," *Advances in Neural Information Processing Systems*, 2008, submitted.

[41] G. Ginis and J. M. Cioffi, "Vectored transmission for digital subscriber line systems," *IEEE Journal on Selected Areas in Communications*, vol. 20, no. 5, pp. 1085–1104, 2002.

[42] P. Stoica, J. Li, and Y. Xie, "On probing signal design for MIMO radar," *IEEE Transactions on Signal Processing*, vol. 55, no. 8, pp. 4151–4161, Aug. 2007.

[43] W.-K. Ma, T. N. Davidson, K. M. Wong, Z.-Q. Luo, and P.-C. Ching, "Quasi-maximum-likelihood multiuser detection using semi-definite relaxation with application to synchronous CDMA," *IEEE Transactions on Signal Processing*, vol. 50, pp. 912–922, 2002.

[44] F. P. Kelly, A. K. Maulloo, and D. K. H. Tan, "Rate control for communication networks: Shadow prices, proportional fairness and stability," *Journal of Operational Research Society*, vol. 49, no. 3, pp. 237–252, 1998.

[45] D. X. Wei, C. Jin, S. H. Low, and S. Hegde, "FAST TCP: motivation, architecture, algorithms, performance," *IEEE/ACM Transactions on Networking*, vol. 14, no. 6, pp. 1246–1259, 2006.

[46] M. Chiang, S. H. Low, A. R. Calderbank, and J. C. Doyle, "Layering as optimization decomposition: A mathematical theory of network architectures," *Proceedings of the IEEE*, vol. 95, no. 1, pp. 255–312, 2007.

[47] S. Shakkottai and R. Srikant, *Network Optimization and Control*. Now Publishers, 2008.

[48] S. Meyn, "Stability and optimization of queueing networks and their fluid models," *Mathematics of Stochastic Manufacturing Systems*, 1997.

[49] M. Chiang, C. W. Tan, D. P. Palomar, D. O'Neill, and D. Julian, "Power control by geometric programming," *IEEE Transactions on Wireless Communications*, vol. 6, no. 7, pp. 2640–2651, 2007.

[50] D. O'Neill, A. Goldsmith, and S. Boyd, "Optimizing adaptive modulation in wireless networks via utility maximization," *Proceedings IEEE International Conference on Communications*, pp. 3372–3377, 2008.

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

[52] C. Donnely and R. Stallman, *Bison version 2.3*, 2006.

[53] The Mathworks, Inc., "Simulink: Simulation and model-based design," http://www.mathworks.com/products/simulink/, Oct. 2008.

[54] J. Eker, J. W. Janneck, E. A. Lee, J. Liu, X. Liu, J. Ludvig, S. Neuendorffer, S. Sachs, and Y. Xiong, "Taming heterogeneity—the Ptolemy approach," *Proceedings of the IEEE*, vol. 91, no. 1, pp. 127–144, 2003.

[55] E. Kant, "Synthesis of mathematical-modeling software," *IEEE Software*, vol. 10, no. 3, pp. 30–41, 1993.

[56] R. Bacher, "Automatic generation of optimization code based on symbolic non-linear domain formulation," *Proceedings International Symposium on Symbolic and Algebraic Computation*, pp. 283–291, 1996.

[57] ——, "Combining symbolic and numeric tools for power system network optimization," *Maple Technical Newsletter*, vol. 4, no. 2, pp. 41–51, 1997.

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

[59] T. Oohori and A. Ohuchi, "An efficient implementation of Karmarkar's algorithm for large sparse linear programs," *Proceedings IEEE Conference on Systems, Man, and Cybernetics*, vol. 2, pp. 1389–1392, 1988.

[60] L. K. McGovern, "Computational analysis of real-time convex optimization for control systems," Ph.D. dissertation, Massachusetts Institute of Technology, 2000.

[61] E. Hazan, "Efficient algorithms for online convex optimization and their applications," Ph.D. dissertation, Department of Computer Science, Princeton University, Sep. 2006.

[62] I. Das and J. W. Fuller, "Real-time quadratic programming for control of dynamical systems," 2008, US Patent 7,328,074.

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

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

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

[66] S. Boyd and B. Wegbreit, "Fast computation of optimal contact forces," *IEEE Transactions on Robotics*, vol. 23, no. 6, pp. 1117–1132, 2006.

[67] R. Fourer, D. Gay, and B. Kernighan, *AMPL: A Modeling Language for Mathematical Programming*. Duxbury Press, Dec. 1999.

[68] A. Brooke, D. Kendrick, A. Meeraus, and R. Raman, *GAMS: A User's Guide*. The Scientific Press, South San Francisco, 1998.

[69] S.-P. Wu and S. Boyd, "SDPSOL: A parser/solver for semidefinite programs with matrix structure," in *Recent Advances in LMI Methods for Control*, L. El Ghaoui and S.-I. Niculescu, Eds. SIAM, 2000, ch. 4, pp. 79–91.

[70] I. The Mathworks, "LMI control toolbox 1.0.8 (software package)," Web site: http://www.mathworks.com/products/lmi, Aug. 2002.

[71] L. El Ghaoui, J.-L. Commeau, F. Delebecque, and R. Nikoukhah, "LMITOOL 2.1 (software package)," Web site: http://robotics.eecs.berkeley.edu/~elghaoui/lmitool/lmitool.html, Mar. 1999.

[72] A. Mutapcic, K. Koh, S.-J. Kim, L. Vandenberghe, and S. Boyd, *GGPLAB: A Simple Matlab Toolbox for Geometric Programming*, 2005, available from www.stanford.edu/~boyd/ggplab/.

[73] 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.

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

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

[76] W. E. Hart, "Python optimization modeling objects (Pyomo)," in *To appear, Proceedings, INFORMS Computing Society Conference*, 2009.

[77] D. Orban and B. Fourer, "Dr. Ampl: A meta solver for optimization," CORS/INFORMS Joint International Meeting, 2004.

[78] I. P. Nenov, D. H. Fylstra, and L. V. Kolev, "Convexity determination in the Microsoft Excel solver using automatic differentiation techniques," *International Workshop on Automatic Differentiation*, 2004.

[79] Y. Lucet, H. H. Bauschke, and M. Trienis, "The piecewise linear-quadratic model for computational convex analysis," *Computational Optimization and Applications*, pp. 1–24, 2007.

[80] B. Widrow, J. M. McCool, M. G. Larimore, and C. R. Johnson, Jr., "Stationary and nonstationary learning characteristics of the LMS adaptive filter," *Proceedings of the IEEE*, vol. 64, no. 8, pp. 1151–1162, 1976.

[81] S. Haykin, *Adaptive filter theory*. Prentice Hall, 1996.

[82] A. T. Erdogan and C. Kizilkale, "Fast and low complexity blind equalization via subgradient projections," *IEEE Transactions on Signal Processing*, vol. 53, no. 7, pp. 2513–2524, 2005.

[83] R. L. G. Cavalcante and I. Yamada, "Multiaccess interference suppression in orthogonal space-time block coded mimo systems by adaptive projected subgradient method," *IEEE Transactions on Signal Processing*, vol. 56, no. 3, pp. 1028–1042, 2008.

[84] D. Bertsimas and A. W. Lo, "Optimal control of execution costs," *Journal of Financial Markets*, vol. 1, no. 1, pp. 1–50, 1998.

[85] R. Almgren and N. Chriss, "Optimal execution of portfolio transactions," *Journal of Risk*, vol. 3, no. 2, pp. 5–39, 2000.

[86] L. Rudin, S. Osher, and E. Fatemi, "Nonlinear total variation based noise removal algorithms," *Physica D*, vol. 60, no. 1-4, pp. 259–268, 1992.

[87] S.-J. Kim, K. Koh, M. Lustig, S. Boyd, and D. Gorinevsky, "A method for large-scale l1-regularized least squares," *IEEE Journal on Selected Topics in Signal Processing*, vol. 3, pp. 117–120, 2007.

[88] A. Bemporad, D. Mignone, and M. Morari, "Moving horizon estimation for hybrid systems and fault detection," in *Proceedings of the American Control Conference*, vol. 4, 1999.

[89] B. Kouvaritakis and M. Cannon, *Nonlinear Predictive Control: Theory and Practice*. IET, 2001.

[90] M. Corless and G. Leitmann, "Controller design for uncertain system via Lyapunov functions," in *Proceedings of the American Control Conference*, vol. 3, 1988, pp. 2019–2025.

[91] E. D. Sontag, "A Lyapunov-like characterization of asymptotic controllability," *SIAM Journal on Control and Optimization*, vol. 21, no. 3, pp. 462–471, 1983.

[92] A. Zymnis, N. Trichakis, S. Boyd, and D. ONeill, "An interior-point method for large scale network utility maximization," *Proceedings of the Allerton Conference on Communication, Control, and Computing*, sep 2007.

[93] R. Srikant, *The Mathematics of Internet Congestion Control*. Birkhäuser, 2004.

[94] D. Bertsekas, *Network Optimization: Continuous and Discrete Models*. Athena Scientific, 1998.

[95] M. Boddy and T. L. Dean, "Solving time-dependent planning problems," Technical report, 1989.

[96] E. A. Hansen and S. Zilberstein, "Monitoring and control of anytime algorithms: A dynamic programming approach," *Artificial Intelligence*, vol. 126, pp. 139–157, 2001.

[97] E. A. Yıldırım and S. J. Wright, "Warm-start strategies in interior-point methods for linear programming," *SIAM Journal on Optimization*, vol. 12, no. 3, pp. 782–810, 2002.

[98] M. Diehl, "Real-time optimization for large scale nonlinear processes," Ph.D. dissertation, University of Heidelberg, 2001.

[99] D. Bertsekas, *Nonlinear Programming*, 2nd ed. Athena Scientific, 1999.

[100] A. Nemirovski and D. Yudin, *Problem complexity and method efficiency in optimization*. Wiley, 1983.

[101] A. V. Fiacco and G. P. McCormick, "Nonlinear programming: sequential unconstrained minimization techniques," DTIC Research Report AD0679036, 1968.

[102] A. Aggarwal and T. H. Meng, "A convex interior-point method for optimal OFDM PAR reduction," *IEEE International Conference on Communications*, vol. 3, 2005.

[103] Y. Y. Saad and H. A. van der Vorst, "Iterative solution of linear systems in the 20th century," *Journal of Computational and Applied Mathematics*, vol. 123, no. 1-2, pp. 1–33, 2000.

[104] M. A. Saunders and J. A. Tomlin, "Stable reduction to KKT systems in barrier methods for linear and quadratic programming," *International Symposium on Optimization and Computation*, 2000.

[105] G. Golub and C. F. V. Loan, *Matrix Computations*, 2nd ed. Johns Hopkins University Press, 1989.

[106] M. Tuma, "A note on the LDL^T decomposition of matrices from saddle-point problems," *SIAM Journal on Matrix Analysis and Applications*, vol. 23, no. 4, pp. 903–915, 2002.

[107] R. J. Vanderbei, "Symmetric quasi-definite matrices," *SIAM Journal on Optimization*, vol. 5, no. 1, pp. 100–113, 1995.

[108] R. J. Vanderbei and T. J. Carpenter, "Symmetric indefinite systems for interior point methods," *Mathematical Programming*, vol. 58, no. 1, pp. 1–32, 1993.

[109] R. Bridson, "An ordering method for the direct solution of saddle-point matrices," Preprint, 2007.

[110] P. R. Amestoy, T. A. Davis, and I. S. Duff, "Algorithm 837: AMD, an approximate minimum degree ordering algorithm," *ACM Transactions on Mathematical Software (TOMS)*, vol. 30, no. 3, pp. 381–388, 2004.

[111] G. Karypis and V. Kumar, "A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs," *SIAM journal on scientific computing*, vol. 20, pp. 359–392, 1999.

[112] S. J. Wright, "Modified Cholesky factorizations in interior-point algorithms for linear programming," *SIAM Journal of Optimization*, vol. 9, pp. 1159–1191, 1999.

[113] P. E. Gill, W. Murray, D. B. Ponceleon, and M. A. Saunders, "Solving reduced KKT systems in barrier methods for linear and quadratic pro-

gramming," Technical Report SOL 91-7, 1991.

[114] A. Aho, M. Lam, R. Sethi, and J. Ullman, "Compilers: Principles, techniques, & tools," 2007.

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

[116] ——, *CHOLMOD User Guide*, 2006, available from http://www.cise.ufl.edu/research/sparse/cholmod/.

[117] D. Aberdeen and J. Baxter, "Emmerald: a fast matrix-matrix multiply using Intel's SSE instructions," *Concurrency and Computation: Practice and Experience*, vol. 13, no. 2, pp. 103–119, 2001.

[118] E. S. Larsen and D. McAllister, "Fast matrix multiplies using graphics hardware," in *Proceedings of the ACM/IEEE conference on Supercomputing*. ACM, 2001, pp. 55–60.

[119] K. V. Ling, B. F. Wu, and J. M. Maciejowski, "Embedded model predictive control (MPC) using a FPGA," in *Proceedings IFAC World Congress*, Jul. 2008, pp. 15 250–15 255.

[120] M. S. K. Lau, S. P. Yue, K. V. Ling, and J. M. Maciejowski, "A comparison of interior point and active set methods for FPGA implementation of model predictive control," *Proceedings, ECC*, 2009, submitted.

[121] M. Grant, "Disciplined convex programming," Ph.D. dissertation, Department of Electrical Engineering, Stanford University, Dec. 2004.

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

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

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