CODE GENERATION FOR
EMBEDDED CONVEX OPTIMIZATION

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF ELECTRICAL ENGINEERING
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

Jacob Elliot Mattingley
February 2011
© 2011 by Jacob Elliot Mattingley. All Rights Reserved.
Re-distributed by Stanford University under license with the author.

This dissertation is online at: http://purl.stanford.edu/qd834ss0073
I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Stephen Boyd, Primary Adviser

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Michael Saunders

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Yinyu Ye

Approved for the Stanford University Committee on Graduate Studies.

Patricia J. Gumport, Vice Provost Graduate Education

This signature page was generated electronically upon submission of this dissertation in electronic format. An original signed hard copy of the signature page is on file in University Archives.
Abstract

Convex optimization is widely used, in many fields, but is nearly always constrained to problems solved in a few minutes or seconds, and even then, nearly always with a human in the loop. The advent of parser-solvers has made convex optimization simpler and more accessible, and greatly increased the number of people using convex optimization. Most current applications, however, are for the design of systems or analysis of data.

It is possible to use convex optimization for real-time or embedded applications, where the optimization solver is a part of a larger system. Here, the optimization algorithm must find solutions much faster than a generic solver, and often has a hard, real-time deadline. Use in embedded applications additionally means that the solver cannot fail, and must be robust even in the presence of relatively poor quality data. For ease of embedding, the solver should be simple, and have minimal dependencies on external libraries. Convex optimization has been successfully applied in such settings in the past. However, they have usually necessitated a custom, hand-written solver. This requires significant time and expertise, and has been a major factor preventing the adoption of convex optimization in embedded applications.

This work describes the implementation and use of a prototype code generator for convex optimization, CVXGEN, that creates high-speed solvers automatically. Using the principles of disciplined convex programming, CVXGEN allows the user to describe an optimization problem in a convenient, high-level language, then receive code for compilation into an extremely fast, robust, embeddable solver.
Acknowledgments

It is with gratitude that I look back over my time at Stanford and think of the many people who have helped me get this far.

A few weeks after arriving, I was captivated by Professor Stephen Boyd’s teaching, and realized how much I could learn from him. When the opportunity arose, it was an easy choice to join his group. His teaching, assistance, and mentoring made the graduate school experience extremely satisfying. I do not think I could have had a better advisor.

Other professors have helped as well; in particular, I would like to thank Professors Antony Fraser-Smith, Yinyu Ye, Sanjay Lall, Michael Saunders and Lieven Vandenberghe. Each has played an important part. The various members of my research group have been inspiring, helpful and supportive: Michael, Alessandro, Almir, Siddharth, Joëlle, Kwangmoo, Argyris, Yang, Ekine, Brendan, Eric, Matt, and Arezou.

I have made many close friends at Stanford and look forward to keeping those relationships for life. And finally, I would like to thank my family for their love and support: Barry, Lorelei, Ruth, and Daniel. This thesis is dedicated to them.
Contents

Abstract iv
Acknowledgments v

1 Overview 1

2 Code Generation for Real-Time Convex Optimization 4
  2.1 Introduction ................................................................. 4
  2.2 Solvers and specification languages .................................. 10
  2.3 Examples ................................................................. 16
  2.4 Algorithm considerations ............................................. 26
  2.5 Summary ................................................................. 29

3 Implementation of a Convex Optimization Code Generator 30
  3.1 Introduction ................................................................. 30
  3.2 CVXGEN example .......................................................... 32
  3.3 Problem specification ................................................... 35
  3.4 Using CVXGEN ............................................................ 38
  3.5 Implementation ........................................................... 41
  3.6 Numerical examples ..................................................... 52
  3.7 Conclusion ............................................................... 58

4 Code Generation for Signal Processing 60
  4.1 Introduction ................................................................. 60
  4.2 Code generation .......................................................... 64
  4.3 Linearizing pre-equalization .......................................... 65
  4.4 Robust Kalman filtering ............................................... 68
Chapter 1

Overview

We start in Chapter 2 by introducing 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 Chapter 2 we introduce the idea of using 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. This chapter is based on [MB10].

Chapter 3 looks at a particular example of an automatic code generator, CVXGEN, which is a prototype code generator we developed to implement and test these ideas. CVXGEN is a software tool that takes a high-level description of a convex optimization problem family and automatically generates custom C code that compiles into a reliable, high-speed solver for the problem family. The current implementation targets problem families that can be transformed, using disciplined convex programming techniques, to convex quadratic programs of modest size. CVXGEN generates simple, flat, library-free code suitable for
embedding in real-time applications. The generated code is almost branch free, so has highly predictable run-time behavior. The combination of regularization (both static and dynamic) and iterative refinement in the search direction computation yields reliable performance, even with poor quality data. We describe how CVXGEN is implemented, and give some results on the speed and reliability of the automatically generated solvers. This chapter is based on [MB11].

Chapter 4 discusses signal processing applications of embedded convex optimization. Convex optimization has been used in signal processing for a long time, to choose coefficients for use in fast (linear) algorithms, such as in filter or array design; more recently, it has been used to carry out (nonlinear) processing on the signal itself. Examples of the latter case include total variation de-noising, compressed sensing, fault detection, and image classification. In both scenarios, the optimization is carried out on timescales of seconds or minutes, and without strict time constraints. Convex optimization has traditionally been considered computationally expensive, so its use has been limited to applications where plenty of time is available. Such restrictions are no longer justified. The combination of dramatically increased computing power, modern algorithms, and new coding approaches has delivered an enormous speed increase, which makes it possible to solve modest-sized convex optimization problems on microsecond or millisecond timescales, and with strict deadlines. This enables real-time convex optimization in signal processing. This chapter is based on [MB09].

The final chapter, Chapter 5, considers another important early application for code generation, receding horizon control (RHC), also known as model predictive control (MPC). RHC is a general-purpose control scheme that involves repeatedly solving a constrained optimization problem, using predictions of future costs, disturbances, and constraints over a moving time horizon to choose the control action. RHC handles constraints, such as limits on control variables, in a direct and natural way, and generates sophisticated feed-forward actions. The main disadvantage of RHC is that an optimization problem has to be solved at each step, which leads many control engineers to think that it can only be used for systems with slow sampling (say, less than one hertz). Several techniques have recently been developed to overcome this problem. In one approach, called explicit MPC, the optimization problem is solved analytically and explicitly, so evaluating the control policy requires only a table lookup. Another approach, which is our focus, is to exploit the structure in the optimization problem to solve it efficiently, using automatic code generation. With code generation, the RHC policy is specified in a high-level language, then automatically transformed into source code for a custom solver. The custom solver
is typically orders of magnitude faster than a generic solver, solving in milliseconds or microseconds on standard processors, making it possible to use RHC policies at even kilohertz rates. In this chapter we demonstrate code generation with four simple control examples. They show a range of problems that may be handled by RHC. In every case, we show a speedup of several hundred times relative to generic parser-solvers. This chapter is based on [MWB10].
Chapter 2

Code Generation for Real-Time Convex Optimization

2.1 Introduction

2.1.1 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 because 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
timescales, 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 describe optimization problems efficiently in a high-level format. This permits the user to see the effect of new terms or constraints rapidly. [BV04, NN94, Ye97, NW06]. Current application areas include control [BB91, BEFB94, DDB95], circuit design [HBL01, HMBL99, BKPH05], economics and finance [Mar52, CT07], networking [KMT98, WJLH06], statistics and machine learning [Vap00, CST00], quantum information theory [EMV03], combinatorial optimization [GGL96], and signal processing [Sig07, CRR69].

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) timescales and with great reliability.

### 2.1.2 Embedded optimization

Here, though, we focus on embedded optimization, where solving optimization problems is part of a wider, automated algorithm. The optimization is deeply embedded in the application, and no human is in the loop. In the introduction to the book *Convex Optimization* [BV04], 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 timescales; 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 timescales that we envision, is model predictive control (MPC), a form of feedback control system. (See Chapter 5.) Traditional (but still widely used) control schemes have relatively simple control policies, requiring only a few basic operations like matrix-vector multiplies and table lookups at each time step [FPEN91, FWP97]. 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 them. By contrast, with MPC, at each step the control action is determined by solving an optimization problem, typically a (convex) quadratic program (QP). MPC 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 [QB03]. 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 timescales as many conventional control methods [BF04, WB08]. 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 because of 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, timescales. The trading algorithms that run on faster timescales 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 offline signal processing, the entire noisy signal is available, and while faster processing is better, there are no hard real-time deadlines. This is the case, for example, in the restoration of audio from wax cylinder recordings, image enhancement, or geophysics inversion problems, where optimization is already widely used. In online 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 [Say03].
CHAPTER 2. REAL-TIME CONVEX OPTIMIZATION

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 [CT05, FKW03, Fel03, JMO03], 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 [KL05].

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 timescales 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 timescale 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. Some examples where it is helpful to solve problem instances from a specific problem family extremely rapidly 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.

2.1.3 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 situations, convex optimization can provide a good heuristic for approximate solution of the problem; see, e.g., [JB08, ZBG09].

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 practice. 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.

2.1.4 Outline

In §2.2, we describe problem families and the specification languages used to model them formally, 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 §2.3. In §2.4 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. This material sets the stage for subsequent chapters, which explore the implementation and application of a code generator.

2.1.5 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 [RF06, GSD05, BH08], or basic model predictive control [CB04, SEM89, AZ00, Mac02, DTG02, Soe92]. Several recent papers describe methods for solving various associated QPs quickly. One approach is explicit MPC, pioneered by Bemporad and Morari [BMDP02], who exploit the fact that the solution of the QP is a piecewise linear function of the problem data that can be determined analytically and ahead of time. Solving instances of the QP then reduces to evaluating a piecewise linear function. Interior-point methods [RWR98], including fast custom interior-point methods [WB08], 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 [WB09a]. Several authors consider fast solution of nonlinear control problems using an MPC framework [ZB07, DBS05, DFA05]. Others discuss various real-time applications [FMOF01, OF96], especially those in robotics [BW07, VDS+08, ZDL+04].

**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 [CWB08, Tro06], or linear programming for error correction [CRTV05]. Goldfarb and Yin discuss interior-point algorithms for solving total variation image restoration problems [GY05]. Some combinatorial optimization problems in signal processing that are approximately, and very quickly, solved using convex relaxations and local search are static fault detection [ZBG09], dynamic fault detection [ZBG08], query model estimation [CT08], and sensor selection [JB08]. In communications, convex optimization is used in DSL [GC02], radar [SLX07], and CDMA [MDW02], to list just a few examples.

Since the publication of the paper by Kelly *et al.* [KMT98], which poses the optimal network flow control as a convex optimization problem, many authors have looked at optimization-based network flow methods [WJLH06, CLCD07, SS08, Mey97], or optimization of power and bandwidth [CTP07, OGB08].
CHAPTER 2. REAL-TIME CONVEX OPTIMIZATION

**Code generation.** The idea of automatic generation of source code is quite old. Parser-generators such as Yacc [Joh75], or more recent tools like GNU Bison [DS06], are commonly used to simplify the writing of compilers. For engineering problems, in particular, there is a range of code generators: One widely used commercial tool is Simulink [The08], while the open-source Ptolemy project [EJL+03] provides a modeling environment for embedded systems. Domain-specific code generators are found in many different fields; see, e.g., [Kan93, Bac96, Bac97, SB04].

Generating source code for optimization solvers is nothing new either; in 1988 Oohori and Ohuchi [OO88] 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 [McG00] gives a computational complexity analysis of real-time convex optimization. Hazan considers algorithms for online convex optimization [Haz06], and Das and Fuller [DF08] hold a patent on an active-set method for real-time QP.

### 2.2 Solvers and specification languages

It will be important for us to distinguish carefully an instance of an optimization problem and a parameterized family of optimization problems because 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.

#### 2.2.1 Problem families and instances

We consider continuously parameterized families of optimization problems of the form

\[
\begin{align*}
\text{minimize} & \quad F_0(x, a) \\
\text{subject to} & \quad F_i(x, a) \leq 0, \quad i = 1, \ldots, m \\
& \quad H_i(x, a) = 0, \quad i = 1, \ldots, p,
\end{align*}
\]

where \( x \in \mathbb{R}^n \) is the (vector) optimization variable, and \( a \in \mathcal{A} \subset \mathbb{R}^f \) is a parameter or data vector that specifies the problem instance. To specify the problem family (2.1), 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

\[
\begin{align*}
\text{minimize} & \quad (1/2)x^TPx + q^Tx \\
\text{subject to} & \quad Gx \leq h, \quad Ax = b,
\end{align*}
\]

with variable \( x \in \mathbb{R}^n \), 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

\[
a = (P, q, G, h) \in \mathcal{A} = \mathbb{S}_n^+ \times \mathbb{R}^n \times \mathbb{R}^{m \times n} \times \mathbb{R}^m,
\]

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

\[
\ell = n(n + 1)/2 + n + mn + m.
\]

In this example, we have

\[
\begin{align*}
F_0(x, a) & = (1/2)x^TPx + q^Tx, \\
F_i(x, a) & = g_i^Tx - h_i, \quad i = 1, \ldots, m, \\
H_i(x, a) & = a_i^Tx - b_i, \quad i = 1, \ldots, p,
\end{align*}
\]

where \( g_i^T \) is the \( i \)th row of \( G \), and \( a_i^T \) is the \( i \)th 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 (2.2).

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

\[
\ell = 2n - 1 + n + N + m.
\]

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

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

Nesterov and Nemirovsky refer to families of convex optimization problems with constant structure and parameterized by finite dimensional parameter vectors as well structured problem (families) [NN94].

### 2.2.2 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 $x^*(a)$ for the problem instance, or determines that the problem instance is infeasible or unbounded.

Traditional solvers [NW06, BV04, Wri97] can handle problem families with a range of dimensions (e.g., QPs with the form (2.2), 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., [BW06, WB08].

2.2.3 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).

2.2.4 Parser-solvers

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

Parser-solvers are widely used. Early (and still widely used) parser-solvers include AMPL [AMP03] and GAMS [BKMR98], which are general-purpose. Parser-solvers that handle more restricted problem types include SDPSOL [WB00], LMLAB [TM02], and LMITOOL [ECDN99] for SDPs and linear matrix inequalities (LMIs), and GGPLAB [MKK++05] for generalized geometric programs. More recent examples, which focus on convex optimization, include YALMIP [Lö04], CVX [GB08a], CVXMOD [MB08] and Pylomo [Har09]. Some tools [OF04, NFK04, LBT07] 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 (2.2) can be specified in CVXMOD as

\[
P = \text{matrix}(...); \quad q = \text{matrix}(...); \quad A = \text{matrix}(...)
\]
CHAPTER 2. REAL-TIME CONVEX OPTIMIZATION

\[\begin{align*}
b &= \text{matrix}(\ldots); \quad G = \text{matrix}(\ldots); \quad h = \text{matrix}(\ldots) \\
x &= \text{optvar}'x', n) \\
qpinst &= \text{problem}(\text{minimize}(0.5*\text{quadform}(x, P) + tp(q)*x), \\
&\quad [G*x <= h, A*x == b])
\end{align*}\]

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

\[\text{qpinst.solve()}\]

which returns either 'optimal' or 'infeasible', and, if optimal, sets \(x\).value to an optimal value \(x^*\).

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 (2.2) is specified in CVXMOD as

\[\begin{align*}
A &= \text{matrix}(\ldots); \quad b = \text{matrix}(\ldots) \\
P &= \text{param}'P', n, n, \text{psd=True}; \quad q = \text{param}'q', n) \\
G &= \text{param}'G', m, n); \quad h = \text{param}'h', m) \\
x &= \text{optvar}'x', n) \\
qpfam &= \text{problem}(\text{minimize}(0.5*\text{quadform}(x, P) + tp(q)*x), \\
&\quad [G*x <= h, A*x == b])
\end{align*}\]

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

If we called \(qpfam.solve()\) right away it would fail because 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:
CHAPTER 2. REAL-TIME CONVEX OPTIMIZATION

![Diagram 2.1](image1.png)

**Figure 2.1:** A parser-solver processes and solves a single problem instance.

![Diagram 2.2](image2.png)

**Figure 2.2:** A code generator processes a problem family, generating a fast, custom solver, which rapidly solves problem instances.

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

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

### 2.2.5 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 2.1 and 2.2 show the difference between code generators and parser-solvers.

A code generator may 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).

While CVXMOD initially had some code generation features, these were later moved to a separate project, CVXGEN. CVXGEN is the subject of Chapter 3.

2.3 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 describe 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 allocate multiple resources optimally 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 in order 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 occurring 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 timescale 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. Sometimes the system is modified directly by a designer; for example, fast embedded optimization could be incorporated into the design and simulation of large, nonlinear models. Here, the optimization could be readily combined with techniques like model reduction [CBMF10].

### 2.3.1 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 \( u_t \in \mathbb{R}^n \) and \( y_t \in \mathbb{R} \) are the vector input and (filtered or equalized) scalar output signals, and \( w_t \in \mathbb{R}^n \) is the filter parameter vector, at time \( t \in \mathbb{Z} \). 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 [Say03, WMLJ76, Hay96]. Subgradient-type or stochastic gradient methods, in which the parameters are updated (usually, slightly) in each step, can also be used [EK05, CY08]. 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

\[
\begin{align*}
\text{maximize} & \quad w_t^T f_t \\
\text{subject to} & \quad |w_t^T g_t^{(i)}| \leq 1, \quad i = 1, \ldots, m,
\end{align*}
\]

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.

### 2.3.2 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 [BL98, AC00], 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

\[
R = p^T s \sim \mathcal{N}(\bar{p}^T s - s^T As, s^T \Sigma s).
\]

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

\[
\mathbb{E} R - \gamma \text{var } R = \bar{p}^T s - s^T Q s,
\]

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

\[
\begin{align*}
\text{maximize} & \quad \bar{p}^T s - s^T Q s \\
\text{subject to} & \quad 0 \leq s \leq S^{\text{max}}, \quad 1^T s = S,
\end{align*}
\]

with variable \( s \in \mathbb{R}^T \). The parameters are \( \bar{p}, Q \) (which depends on the original problem data \( \Sigma, A, \) and \( \gamma \)), \( S^{\max} \), and \( S \). An obvious initialization is \( s = (S/T)1 \), 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 occurred.

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 \( s_T = S - \sum_{t=1}^{T-1} s_t \), i.e., the remaining unsold shares.

### 2.3.3 Sliding window smoothing

We are given a noise-corrupted scalar signal \( y_t, t \in \mathbb{Z} \), and want to form an estimate of the underlying signal, which we denote \( x_t, t \in \mathbb{Z} \). 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

\[
\begin{align*}
\text{minimize} & \quad \sum_{\tau=t-p}^{t+q} (y_\tau - \hat{x}_\tau)^2 + \lambda \phi(\hat{x}_{t-p}, \ldots, \hat{x}_{t+q}) \\
\text{subject to} & \quad (\hat{x}_{t-p}, \ldots, \hat{x}_{t+q}) \in C,
\end{align*}
\]

with variables \( (\hat{x}_{t-p}, \ldots, \hat{x}_{t+q}) \in \mathbb{R}^{p+q+1} \). Here \( \phi : \mathbb{R}^{p+q+1} \rightarrow \mathbb{R} \) is a (typically convex) function that measures the implausibility of \( (\hat{x}_{t-p}, \ldots, \hat{x}_{t+q}) \), and \( C \subset \mathbb{R}^{p+q+1} \) 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 \( \hat{x}_t = \hat{x}_t^* \), where \( \hat{x}_t^* \) is a solution of the problem above.

The implausibility function \( \phi \) 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 \( \phi(z) = \sum_{i=p+q} |z_{t+1} - z_t| \), the total variation of \( z \) [ROF92]. Another interesting case is \( \phi(z) = \sum_{i=p+q} |z_{t+1} - 2z_t + z_{t-1}| \), the \( \ell_1 \) norm of the second-order difference (or Laplacian); the resulting filter is called an \( \ell_1 \)-trend filter [KKL+07].

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

### 2.3.4 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 [BMM99, Mac02, KC01].
A linear dynamical system is modeled as

\[ x_{t+1} = Ax_t + w_t, \]

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

\[ y_t = Cx_t + v_t, \]

where \( y_t \in \mathbb{R}^p \) 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 \( y_{t-T+1}, \ldots, y_t \), 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

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

with variables \( \tilde{x}_{t-T}, \ldots, \tilde{x}_t \). Our estimate is then \( \hat{x}_t = \tilde{x}_t^* \), where \( \tilde{x}^* \) is a solution of the problem above. When \( \mathcal{X}, \phi_w, \) and \( \phi_v \) 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 \( v_t \), respectively, 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

\[
\phi_v(v) = \begin{cases} 
(1/2)\|v\|_2^2 & \text{if } \|v\|_2 \leq 1 \\
\|v\|_1 - 1/2 & \text{if } \|v\|_2 > 1 
\end{cases}
\]

yields state estimates that are surprisingly immune to occasional large values of the measurement noise \(v_t\). (See, e.g., [BV04, §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.

### 2.3.5 Real-time input design

We consider a linear dynamical system

\[
x_{t+1} = Ax_t + Bu_t,
\]

where \(x_t \in \mathbb{R}^n\) is the state and \(u_t \in \mathbb{R}^m\) is the control input at time period \(t \in \mathbb{Z}\). 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 \(x^{\text{des}}\), with input and state bounds. This can be formulated as the QP

\[
\text{minimize} \quad \sum_{\tau=t}^{T-1} \|u_\tau\|_2^2 \\
\text{subject to} \quad x_{\tau+1} = Ax_\tau + Bu_\tau, \quad \tau = t, \ldots, t+T - 1 \\
u_{\tau}^{\text{min}} \leq u_\tau \leq u_{\tau}^{\text{max}}, \quad \tau = t, \ldots, t+T - 1 \\
x_{\tau}^{\text{min}} \leq x_\tau \leq x_{\tau}^{\text{max}}, \quad \tau = t, \ldots, t+T \\
x_T = x^{\text{des}},
\]

with variables \(u_t, \ldots, u_{t+T-1}, x_t, \ldots, x_{t+T}\). (The inequalities on \(u_\tau\) and \(x_\tau\) are component-wise.)

### 2.3.6 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, [ZTBO07, Sri04], [Ber98, §8]). We consider a network with \(m\) edges or links, labeled \(1, \ldots, m\), and
n flows, labeled 1, . . . , 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, \( t \in \mathbb{R}^m \), 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

\[
\begin{align*}
\text{maximize} & \quad U(f) \\
\text{subject to} & \quad Rf \leq c, \quad f \geq 0,
\end{align*}
\]

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

\[
\begin{align*}
\text{maximize} & \quad w^T f \\
\text{subject to} & \quad Rf \leq c, \quad 0 \leq f \leq s,
\end{align*}
\]

(2.3)

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 (2.3) 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 \( i \).
2.3.7 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 \( p^{\text{in}}_j \) denote the (nonnegative) power that enters the tail of edge \( j \); \( p^{\text{out}}_j \) will denote the (nonnegative) power that emerges from the head of edge \( j \). These are related by

\[
p^{\text{in}}_j = p^{\text{out}}_j + \ell_j(p^{\text{in}}_j), \quad j = 1, \ldots, m,
\]

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

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

\[
l_i + \sum_{j \in I(i)} p^{\text{out}}_j = \eta_i \left( g_i + \sum_{j \in O(i)} p^{\text{in}}_j \right), \quad i = 1, \ldots, n,
\]

where \( l_i \) is the load power at node \( i \), \( g_i \) is the generator input power at node \( i \), \( I(i) \) is the set of incoming edges to node \( i \), and \( O(i) \) is the set of outgoing edges from node \( i \). We take \( l_i = 0 \) if \( i \not\in \mathcal{L} \), and \( g_i = 0 \) if \( i \not\in \mathcal{G} \).

In the problem of optimal generation and distribution, the node loads \( l_i \) are given; the goal is find generator powers \( g_i \leq G^{\text{max}}_i \), and line power flows \( p^{\text{in}}_j \) and \( p^{\text{out}}_j \), 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

\[
\begin{align*}
\text{minimize} & \quad c^T g \\
\text{subject to} & \quad \text{(2.4), (2.5)} \\
& \quad 0 \leq g \leq G^{\text{max}}, \\
& \quad 0 \leq p^{\text{in}} \leq P^{\text{max}}, \quad 0 \leq p^{\text{out}}
\end{align*}
\]

with variables \( g_i \), for \( i \in \mathcal{G} \); \( p^{\text{in}} \in \mathbb{R}^m \), and \( p^{\text{out}} \in \mathbb{R}^m \). (We take \( g_i = 0 \) for \( i \not\in \mathcal{G} \).)
Relaxing the line equations (2.4) 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 (2.4).)

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.

### 2.3.8 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, \( s_{\text{min}} \) and \( s_{\text{max}} \). The energy consumed by the processor in period \( t \) is given by \( \phi(s_t) \), where \( \phi : \mathbb{R} \to \mathbb{R} \) 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

\[ \sum_{t=A_i}^{D_i} S_{ti} \geq W_i, \quad i = 1, \ldots, n. \]  \hspace{1cm} (2.6)
CHAPTER 2. REAL-TIME CONVEX OPTIMIZATION

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

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

\[
\text{minimize} \quad E = \sum_{t=1}^{T} \phi(s_t)
\]

subject to \( (2.6) \)

\[s_{\min} \leq s \leq s_{\max}, \quad s = S_{1}, \quad S \geq 0,\]

with variables \( s \in \mathbb{R}^{T} \) and \( S \in \mathbb{R}^{T \times n} \). (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 canceled, or changes in availability and deadlines. This scheme requires the solution of a scheduling problem in each period.

2.4 Algorithm considerations

2.4.1 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 solve a wide range of problems efficiently.

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 [BV04, §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, \(e.g.,\) [Mac02, §3.4]. Another option is to use an infeasible Newton-based method [BV04, §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 [WB08]. (This assumes that we have an initial point that satisfies the inequalities.)

Guaranteed run time bounds. Algorithms used in a real-time setting must be fast, with execution time that is \textit{predictable} and \textit{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 because they exit only when certain residuals are sufficiently small.

Another option that can be useful in synchronous or asynchronous real-time optimization applications is to employ an \textit{any-time} algorithm, \(i.e.,\) an algorithm that can be interrupted at any time (after some minimum) and shortly thereafter returns a reasonable approximation of the solution [Bod89, HZ01].

2.4.2 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 lower 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 [WB08].

2.4.3 Interior-point methods

Many methods can be used to solve optimization problems in a real-time setting. For example, Diehl et al. [DBS05, DFA+05, Die01] have used active-set methods for real-time nonlinear MPC. First-order methods, such as classical projected-gradient methods (see, e.g., [Ber99]), or the more recently developed mirror-descent methods [NY83], can also be attractive, especially when warm-started, because the accuracy requirements for embedded applications can sometimes be low. 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 \( \kappa \) that controls the accuracy or duality gap (see, for example, [BV04, §11] or [FM68]). For some real-time embedded applications, we can even fix \( \kappa \) at some suitable value, and limit the number of Newton steps taken. With proper choice of \( \kappa \), and warm-start initialization, good application performance can be obtained with just a few Newton steps. This approach is used in [BW07] to compute optimal robot grasping forces, and in [WB08] for MPC.

More sophisticated interior-point methods, such as primal-dual methods [NW06, §19], [Wri97] 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 adequate for real-time embedded applications is obtained in just a few steps. A primal-dual method was selected for CVXGEN.
2.5 Summary

This chapter has set the stage for real-time embedded optimization, where we must solve many instances of an optimization problem from a given family. 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 be used to 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.
Chapter 3

Implementation of a Convex Optimization Code Generator

3.1 Introduction

We have seen in Chapter 2 that convex optimization is widely used, and that embedded convex optimization applications show much promise. Parser-solvers like CVX [GB08a] and YALMIP [Lö4] make the process of specifying and solving a convex problem simple, and so are ideal for prototyping an algorithm or method that relies on convex optimization. But the resulting solve times are measured in seconds or minutes, which precludes their use in faster real-time systems. In addition, these tools require extensive libraries and commercial software to run, and so are not suitable for embedding in many applications. Conventionally, however, moving from a general-purpose parser-solver to a high-speed, embeddable solver requires extensive modeling and conversion by hand. This is a time-consuming process that requires significant expertise, so embedded convex optimization applications have so far been limited.

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

As introduced in §2.1.2, the setting we are focusing on here is embedded convex optimization, where finding the solution to convex optimization problems is part of a wider algorithm. In §2.4.1, we described the requirements that this imposes on the solver. These include robustness (that the solver should never cause a ‘fatal error’ such as a division-by-zero or segmentation fault) and speed (because the system sample rate may dictate a solve time measured in milliseconds or microseconds). The solver should have a simple footprint. General-purpose parser-solvers usually depend on either an integrated environment like Matlab, or, at least, extensive pre-built libraries. This makes it difficult to adapt and validate the solver for use in embedded applications. Instead, the solver should ideally use simple, flat code with minimal, or no, library dependencies.

While embedded solvers have certain requirements, they also have certain features that can be exploited to make their design less challenging. These requirements are described in §2.4.2. CVXGEN addresses both the requirements and the exploitable features of embedded solvers via automatic code generation.

3.1.1 Prior work

The other chapters in this document discuss various current applications of real-time optimization (Chapter 2), especially in control systems (Chapter 5) and signal processing (Chapter 4); however, we do not know of any previous general-purpose convex optimization code generators. This is despite the fact that, as mentioned in §2.1.5, the idea of automatic code generation is rather old, and has been used since (at least) the 1970s. CVXGEN is the first tool of its kind.

CVXGEN was originally part of CVXMOD [MB08], a general-purpose convex optimization parser-solver for Python. Rudimentary code generation capability was added to CVXMOD, but this functionality was soon moved into the separate (Ruby) project CVXGEN. We mention this because previous publications [MB10, MB09] refer to CVXMOD as a code generator; however, we consider CVXMOD to be merely an early prototype of CVXGEN.

3.1.2 Overview

The remainder of this chapter discusses the use and implementation of CVXGEN. In §3.2, we give an example showing how CVXGEN looks to the user. In §3.3, we describe
the CVXGEN specification language, which is used to describe each problem family. In §3.4, we show how the generated solver code is embedded in an application. The CVXGEN implementation requires parsing and conversion to a standard form QP, discussed in §3.5.1. Solving the standard form QP is covered in §3.5.2 and §3.5.3, and the code generation process itself is described in §3.5.4. Finally, we report on speed and reliability by showing several examples in §3.6.

3.2 CVXGEN example

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

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

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

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

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

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

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

\[ 1^T(z_t)_- \leq \eta 1^T z_t. \]

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

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

\[
\begin{align*}
\text{maximize} & \quad q_t^T z_t - z_t^T Q_t z_t \\
\text{subject to} & \quad z_t = x_t + u_t \\
& \quad 1^T u_t + b_t^T (u_t)_+ + s_t^T (u_t)_- \leq 0 \\
& \quad 1^T (z_t)_- \leq \eta 1^T z_t,
\end{align*}
\]

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

\[ q_t, \ Q_t, \ x_t, \ b_t, \ s_t, \ \eta. \]

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

Our trading policy works as follows. In period \( t \), we obtain the problem parameters (3.2). Some of these are known (such as the current portfolio \( x_t \)); others are specified or chosen (such as \( \eta \)); and others are estimated by an auxiliary algorithm (\( q_t, Q_t, b_t, s_t \)). We then solve problem (3.1), which is feasible provided \( 1^T x_t \geq 0 \). (If this is not the case, we declare ruin and quit.) We then choose the trade \( u_t \) as a solution of (3.1). By construction, this trade will be self-financing, and will respect the post-trade leverage constraint. (The solution can be \( u_t = 0 \), meaning that no trades should be executed in period \( t \)).

**CVXGEN specification.** The CVXGEN specification of this problem family is shown in Figure 3.1, for the case with \( n = 20 \) assets. The specification is explored in detail in §3.3, but for now we point out the obvious correspondence between the optimization problem (3.1) and the CVXGEN description in Figure 3.1.

For this problem family, code generation takes 24 s, and the generated code requires
 CHAPTER 3. CODE GENERATOR IMPLEMENTATION

dimensions
  n = 20
end

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

variables
  u_t(n);  z_t(n)
end

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

Figure 3.1: CVXGEN problem specification for the multi-period trading problem example.
7.9 s to compile. The generated solver solves instances of the problem (3.1) in 200 µs. For comparison, instances of the problem (3.1) require around 600 ms to solve using CVX, so code generation yields a speed-up of around 3000x.

Even if the actual trading application does not require a 200 µs solve time, a fast solver is still very useful. For example, to test the performance of our trading policy (together with the auxiliary algorithm that provides the parameter estimates in each period), we would need to solve,sequentially, many instances of the problem (3.1). Simulating or testing the trading policy for one year, with trading every 10 minutes (say) and around 2000 hours of trading per year, requires solving 12000 instances of the problem (3.1) (sequentially, so it cannot be done in parallel). Using CVX, the trading policy simulation time would be around two hours (not counting the time required to produce the parameter estimates). Using the CVXGEN-generated solver for this same problem, on the same computer, the year-long simulation can be carried out in a few seconds. The speed-up is important because we may need to simulate the trading policy many times as we adjust the parameters or develop the auxiliary prediction algorithm.

### 3.3 Problem specification

Here we describe CVXGEN’s problem specification language. The language is built on the principles of disciplined convex programming (DCP) [Gra04, GBY06, GB08b]. By imposing several simple rules on the problem specification, we ensure that valid problem statements represent convex problems, which can be transformed to canonical form in a straightforward and automatic way. Figure 3.1 shows the CVXGEN problem specification for problem (3.1).

#### 3.3.1 Symbols

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

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

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

3.3.2 Functions and expressions

Expressions are created from parameters and variables using addition, subtraction, multiplication, division and several additional functions. These expressions can then be used in the objective and constraints. The example in Figure 3.1 shows basic matrix and vector multiplication and addition, transposition, and the use of several different functions. Expressions may also be created with scalar division (although with no optimization variables in the denominator) and vector indexing.

CVXGEN comes with a small set of functions that can be composed to create problem descriptions, when supported by the relevant convex calculus (see §3.3.3). There are two sets of functions provided by CVXGEN. The first set may be used in the objective and constraints, and consists of elementwise absolute value (abs), vector and elementwise maximum and minimum (max and min), \( \ell_1 \) and \( \ell_\infty \) norms (norm_1 and norm_inf), vector summation (sum) and elementwise positive part and negative part (pos and neg). The second set of functions consists of the quadratic and square functions. These can only be used in the objective. This is necessary so that the problem can be transformed to a QP.

3.3.3 Convexity

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

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

• Negation is affine and nonincreasing. It is nonpositive when its argument is nonnegative, and nonnegative when its argument is nonpositive.

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

• A constant expression is one of:
  
  – A literal constant.
  – A parameter.
  – A function of constant expressions.

• An affine expression is one of:
  
  – A constant expression.
  – An optimization variable.
  – An affine function of affine expressions.

• A convex expression is one of:
  
  – An affine expression.
  – A convex function of an affine expression.
  – A convex nondecreasing function of a convex expression.
  – A convex function, nondecreasing when its argument is nonnegative, of a convex nonnegative expression.
  – A convex nonincreasing function of a concave expression.
  – A concave function, nondecreasing when its argument is nonnegative, of a convex nonnegative expression.

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

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

### 3.3.4 Objective and constraints

The objective is a direction (\texttt{minimize} or \texttt{maximize}) and a (respectively, convex or concave) scalar expression. Feasibility problems are specified by omitting the objective. The problem specification in Figure 3.1 is a concave maximization problem.

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

- convex \( \leq \) concave,
- concave \( \leq \) convex or
- affine \( = \) affine.

The CVXGEN specification in Figure 3.1 contains two constraints: an affine equality constraint and a convex-less-than-affine inequality constraint (which is a special case of convex-less-than-concave because affine functions are also concave). In CVXGEN, the square function cannot appear in constraints because the problem is converted to a convex QP.

### 3.4 Using CVXGEN

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

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

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

3.4.2 Using the generated code

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

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

```c
for (;;) { // Main control loop.
    load_data(params);
    // Solve individual problem instance defined in params.
    num_iters = solve(params, vars, work, settings);
    // Solution available in vars; status details in work.
}
```

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

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

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

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

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

- `settings.kkt_reg`, with default $10^{-7}$. This controls the regularization $\epsilon$ added to the KKT matrix. See §3.5.3.

- `settings.refine_steps`, with default 1. This controls the number of steps of iterative refinement. See §3.5.3.

3.4.4 Handling infeasibility and unboundedness

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

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

A (classical) option for handling possible infeasibility is to create an additional ‘phase I solver’, which finds a feasible point if one exists, and otherwise finds a point that minimizes some measure of infeasibility. This solver can be called after the original solver has failed to converge [BV04, §11.4].
To avoid unbounded problems, problems should include additional constraints, such as lower and upper bounds on some or all variables. These should be set sufficiently large so that bounded problems are unaffected, and may be checked for tightness, after solution. (If any bound constraints are tight, we mark the problem instance as likely unbounded.)

3.4.5 Increasing solver speed

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

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

Changing the solver settings can also improve performance. For applications where average solve times are more important than maximum times, we recommend using relaxed constraint satisfaction and duality gap specifications (see §3.4.3), which allow early termination once a good (but not provably optimal) solution is found. Often, a near-optimal point is found early, with subsequent iterations merely confirming the point’s quality.

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

3.5 Implementation

In this portion of the chapter, we describe the techniques used to create CVXGEN solvers and make them fast and robust. While CVXGEN handles only problems that transform to QPs, nearly all of the techniques described would apply, with minimal variation, to more general convex problem families such as second-order cone programs (SOCPs).
3.5.1 Parsing and canonicalization

Before code generation, CVXGEN problem specifications, in the form discussed in §3.3, are parsed and converted to an internal CVXGEN representation. All convexity and dimension checking is performed in this internal layer. Once parsed, the problem family is analyzed to determine the problem transformations required to target a single canonical form. With vector variable $x \in \mathbb{R}^n$, the canonical form is

$$
\begin{align*}
\text{minimize} \quad & (1/2)x^T Q x + q^T x \\
\text{subject to} \quad & G x \leq h, \quad A x = b,
\end{align*}
$$

(3.3)

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

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

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

$$
\|Ay - b\|_1 \leq 3.
$$

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

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

which, crucially, are all affine.

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

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

For completeness, we now describe the algorithm. This standard algorithm is taken from [Van10], but similar treatments may be found in [Wri97, NW06, Stu02], with the Mehrotra predictor-corrector, in particular, described in [Meh92, Wri97].

**Introduce slack variables.** Given a QP in the form (3.3), introduce a slack variable $s \in \mathbb{R}^p$, and solve the equivalent problem

$$\begin{align*}
\text{minimize} & \quad (1/2)x^T Q x + q^T x \\
\text{subject to} & \quad G x + s = h, \quad A x = b, \quad s \geq 0,
\end{align*}$$

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

$$\begin{align*}
G x + s &= h, \quad A x = b, \quad s \geq 0 \\
z &\geq 0 \\
Q x + q + G^T z + A^T y &= 0 \\
s_i z_i &= 0, \quad i = 1, \ldots, p.
\end{align*}$$

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

$$\begin{align*}
\text{minimize} & \quad (1/2)x^T Q x + q^T x + (1/2)\|s\|_2^2 \\
\text{subject to} & \quad G x + s = h, \quad A x = b,
\end{align*}$$

with variables $x$ and $s$, and

$$\begin{align*}
\text{maximize} & \quad -(1/2)w^T Q w - h^T z - b^T y - (1/2)\|z\|_2^2 \\
\text{subject to} & \quad Q w + q + G^T z + A^T y = 0,
\end{align*}$$
with variables $w$, $y$ and $z$. We can solve for the optimality conditions of both problems simultaneously by solving the linear system

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

We use the solution to set the initial primal and dual variables to

\[x^{(0)} = x\]
\[y^{(0)} = y\]
\[z^{(0)} = Gx - h\]
\[\alpha_p = \inf\{\alpha \mid -z + \alpha 1 \geq 0\}\]

as the initial value of $s$. Finally, we set

\[\alpha_d = \inf\{\alpha \mid z + \alpha 1 \geq 0\}\]

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

**Main iterations.**

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

2. Compute affine scaling directions by solving

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

where $S = \text{diag}(s)$ and $Z = \text{diag}(z)$. We will shortly see that we do not solve this system directly.
3. Compute centering-plus-corrector directions by solving

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

where \( \mu = s^T z / p \),

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

and

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

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

\[
\begin{align*}
\Delta x &= \Delta x^{aff} + \Delta x^{cc} \\
\Delta s &= \Delta s^{aff} + \Delta s^{cc} \\
\Delta y &= \Delta y^{aff} + \Delta y^{cc} \\
\Delta z &= \Delta z^{aff} + \Delta z^{cc}
\end{align*}
\]

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

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

5. Update primal and dual variables:

\[
\begin{align*}
x &:= x + \alpha \Delta x \\
s &:= s + \alpha \Delta s \\
y &:= y + \alpha \Delta y \\
z &:= z + \alpha \Delta z
\end{align*}
\]

6. Repeat from step 1.

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

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

$$
K = \begin{bmatrix}
Q & 0 & G^T & A^T \\
0 & S^{-1}Z & I & 0 \\
G & I & 0 & 0 \\
A & 0 & 0 & 0
\end{bmatrix}.
$$

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

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

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

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

**Regularization**

To ensure the factorization always exists and is numerically stable, we modify the linear system. Instead of the original system $K$, we regularize the KKT matrix by choosing $\epsilon > 0$
and work with
\[
\widetilde{K} = \begin{bmatrix}
Q & 0 & G^T & A^T \\
0 & S^{-1} Z & I & 0 \\
G & I & 0 & 0 \\
A & 0 & 0 & 0
\end{bmatrix} + \begin{bmatrix}
\epsilon I & 0 \\
0 & -\epsilon I
\end{bmatrix}.
\]

This new matrix \( \widetilde{K} \) is *quasidefinite*, i.e., symmetric with \( (1,1) \) block diagonal positive definite, and \( (2,2) \) block diagonal negative definite. This means that, for any permutation \( P \) the \( LDL^T \) factorization must exist [Sau95, GSS96, Van95]. In fact, as long as \( \epsilon \) is sufficiently large, the factorization will be adequately stable for any choice of permutation [Sau96, §4.2]. Thus, to find a solution to the system \( \widetilde{K}\ell = r \), we can permute and factor \( \widetilde{K} \) so that
\[
P\widetilde{K}P^T = LDL^T,
\]
then find solutions \( \ell \) via
\[
\ell = \widetilde{K}^{-1}r = P^T L^{-T} D^{-1} L^{-1} P r,
\]
where \((\cdot)^{-1}\) denotes not matrix inversion and multiplication, but the application of backward substitution, scaling, and forward substitution, respectively. This provides solutions to the perturbed system of equations with matrix \( \widetilde{K} \) instead of \( K \). This is not necessarily a problem, since the search direction found via this method is merely a heuristic, and good performance can still be obtained with \( \widetilde{K} \approx K \). However, we now discuss a method that allows us to recover solutions to the original system with matrix \( K \).

**Iterative refinement**

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

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

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

**Dynamic regularization**

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

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

$$D_{ii} = \xi_i((\xi_i\tilde{D}_{ii})_+ + \epsilon),$$

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

**Choosing a permutation**

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

$$PKP^T = LDL^T$$

will exist and will be unique. However, the choice of $P$ is important in another way: It determines the number, and pattern, of nonzero entries in $L$. All nonzero entries in the lower triangular portion of $PKP^T$ will cause corresponding nonzero entries in $L$; additional nonzero entries in $L$ are called *fill-in*. We wish to (approximately) minimize the number of nonzero entries in $L$, as it approximately corresponds to the amount of work required to
CHAPTER 3. CODE GENERATOR IMPLEMENTATION

factorize and solve the linear system. Thus, we use a heuristic to choose $P$ to minimize the nonzero count in $L$. We use a simple, greedy method, called local minimum fill-in [DER89, §7.5]. This technique requires comparatively large amounts of time to determine, but with CVXGEN occurs at code generation time, and thus has no solve time penalty. We now describe the permutation selection algorithm.

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

2. For each node $i \not\in E$, calculate the fill-in if it were eliminated next. This is simply the number of missing links between uneliminated nodes $j, k \not\in E$ for which $L_{jk} = 0$ and $L_{ij} = L_{ik} = 1$.

3. Select the node $i$ for which the fill-in would be lowest, add it to $E$, and make the appropriate changes to $L$.

4. Repeat steps (2) and (3) until all nodes have been eliminated. This gives us the elimination ordering, and the structure of non-zero elements in the factor $L$.

Two example sparsity patterns, after permutation and fill-in, are shown in Figure 3.2. Elements that constitute fill-in are shown in red. The pattern on the left-hand side is for an MPC problem like those described in §3.6.4. There are 398 non-zero entries in the (non-strict) lower triangle of the regularized KKT matrix; after permutation and fill-in, there are 509 non-zero entries in $L$. This gives a fill-in factor of 1.28. The pattern on the right-hand side is for a lasso problem like those of §3.6.3. There are 358 non-zero entries in the KKT matrix; afterward, there are 411, for a fill-in factor of 1.15.

3.5.4 Code generation

The goal of code generation is to describe the structure and implementation of a solver once, then programmatically transform that implementation, any number of times, into a code tailored for a specific problem. This is much like a compiler, which allows programmers to write code in a more powerful, higher-level language, while still getting the performance from (say) assembly code after compilation. CVXGEN uses a templating language to describe the general solver structure, and a modeling and generation layer that fills the holes in each template with detailed code specific to each solver.
CHAPTER 3. CODE GENERATOR IMPLEMENTATION

Figure 3.2: Sparsity patterns of the factor $L$, with red indicating fill-in.

**Templating language**

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

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

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

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

This extremely basic example demonstrates how the template has the flavor of a general-purpose solver before code generation (using symbolic $p$), but a very specific solver afterwards (using numeric 100).
CHAPTER 3. CODE GENERATOR IMPLEMENTATION

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

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

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

Thus, with a very short description length in the templating language, we get extremely explicit, highly optimizable code ready for processing by the compiler.

Explicit coding style

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

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

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

3.6.1 Simple quadratic program

For the first example, we consider the basic quadratic program

\[
\begin{align*}
\text{minimize} & \quad x^T Q x + c^T x \\
\text{subject to} & \quad Ax = b, \quad 0 \leq x \leq 1,
\end{align*}
\]

with optimization variable \( x \in \mathbb{R}^n \) and parameters \( A \in \mathbb{R}^{m \times n}, b \in \mathbb{R}^m, c \in \mathbb{R}^n \) and \( Q \in \mathbb{S}^n_+ \). Results for three different problem sizes are shown in Table 3.1.

3.6.2 Support vector machine

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

\[
\begin{align*}
\text{minimize} & \quad \|w\|^2_2 + \lambda \sum_{i=1}^N \left(1 - y_i (w^T x_i - b)\right)_+.
\end{align*}
\]

Table 3.2 shows the results for two problem families of different sizes.
### Table 3.1: Performance results for the simple quadratic program example.

<table>
<thead>
<tr>
<th>Size $(m, n)$</th>
<th>Small $(3, 10)$</th>
<th>Medium $(6, 20)$</th>
<th>Large $(12, 40)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>CVX and SeDuMi</td>
<td>230 ms</td>
<td>260 ms</td>
<td>340 ms</td>
</tr>
<tr>
<td>Scalar parameters</td>
<td>143</td>
<td>546</td>
<td>2132</td>
</tr>
<tr>
<td>Variables, original</td>
<td>10</td>
<td>20</td>
<td>40</td>
</tr>
<tr>
<td>Variables, transformed</td>
<td>10</td>
<td>20</td>
<td>40</td>
</tr>
<tr>
<td>Equalities, transformed</td>
<td>3</td>
<td>6</td>
<td>12</td>
</tr>
<tr>
<td>Inequalities, transformed</td>
<td>20</td>
<td>40</td>
<td>80</td>
</tr>
<tr>
<td>KKT matrix dimension</td>
<td>53</td>
<td>106</td>
<td>212</td>
</tr>
<tr>
<td>KKT matrix nonzeros</td>
<td>165</td>
<td>490</td>
<td>1620</td>
</tr>
<tr>
<td>KKT factor fill-in</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>Code size</td>
<td>123 kB</td>
<td>377 kB</td>
<td>1891 kB</td>
</tr>
<tr>
<td>Generation time, i7</td>
<td>0.6 s</td>
<td>5.6 s</td>
<td>95 s</td>
</tr>
<tr>
<td>Compilation time, i7</td>
<td>1.1 s</td>
<td>4.2 s</td>
<td>56 s</td>
</tr>
<tr>
<td>Binary size, i7</td>
<td>67 kB</td>
<td>231 kB</td>
<td>1256 kB</td>
</tr>
<tr>
<td>CVXGEN, i7</td>
<td>26 µs</td>
<td>110 µs</td>
<td>720 µs</td>
</tr>
<tr>
<td>CVXGEN, Atom</td>
<td>250 µs</td>
<td>860 µs</td>
<td>4.6 ms</td>
</tr>
<tr>
<td>Maximum iterations required, 99.9%</td>
<td>8</td>
<td>9</td>
<td>11</td>
</tr>
<tr>
<td>Maximum iterations required, all</td>
<td>20</td>
<td>17</td>
<td>12</td>
</tr>
</tbody>
</table>
### Table 3.2: Performance results for the support vector machine example.

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

This example, from statistics, demonstrates the lasso procedure ($\ell_1$-regularized least squares) [BV04, §6.3.2]. Here we wish to solve the optimization problem

$$\min \frac{1}{2} \|Ax - b\|^2_2 + \lambda \|x\|_1,$$

with parameters $A \in \mathbb{R}^{m \times n}, b \in \mathbb{R}^m$ and $\lambda \in \mathbb{R}_+$, and optimization variable $x \in \mathbb{R}^n$. The problem is interesting both when $m < n$ and when $m > n$. Table 3.3 shows performance results for an example from each case.

### 3.6.4 Model predictive control

This example, from control systems, is for model predictive control (MPC), also known as receding horizon control (RHC). See Chapter 5 for several detailed CVXGEN MPC
### Table 3.4: Performance results for the model predictive control example.

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

Examples. For this example, we solve the optimization problem

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

with optimization variables \(x_1, \ldots, x_{T+1} \in \mathbb{R}^n\) (state variables) and \(u_0, \ldots, u_T \in \mathbb{R}^m\) (input variables); and problem data consisting of system dynamics matrices \(A \in \mathbb{R}^{n \times n}\) and \(B \in \mathbb{R}^{n \times m}\); input cost diagonal \(R \in S_+^{m \times m}\), state and final state costs diagonal \(Q \in S_+^{n \times n}\) and dense \(Q_{\text{final}} \in S_+^{n \times n}\); amplitude and slew rate limits \(u_{\text{max}} \in \mathbb{R}_+\) and \(S \in \mathbb{R}_+\); and initial state \(x_0 \in \mathbb{R}^n\). Table 3.4 shows performance results for three problem families of varying sizes.

### 3.6.5 Settings and reliability

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

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

The problem we solve is

\[
\begin{align*}
\text{minimize} & \quad \|Ax - b\|_1 \\
\text{subject to} & \quad -1 \leq x \leq 1,
\end{align*}
\]

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

Figure 3.3(a) shows the performance of the solver with default settings. All problems were solved within 14 iterations, so it would be reasonable to set a maximum iteration limit of 10, at the cost of slightly poorer accuracy in less than 1\% of cases.

If the regularization is removed, by setting the regularization \( \epsilon = 0 \), the solver fails in every case. This is because no factorization of \( K \) is possible with the permutation chosen by CVXGEN. However, as long as some regularization is present, solution is still successful. Figure 3.3(b) shows the behavior of the solver with the regularization decreased by a factor of \( 10^4 \), to \( \epsilon = 10^{-11} \). This is a major change, yet the solver works nearly as well, encountering the iteration limit in less than 0.2\% of cases.

The CVXGEN generated solver shows similarly excellent behavior with increased regularization. To illustrate the point, however, Figure 3.3(c) shows what happens when regularization is increased too much, by a factor of \( 10^5 \) to \( \epsilon = 10^{-2} \). Even with this excessive regularization, however, the solver still only reaches the iteration limit in 13\% of
cases.

This extreme case gives us an opportunity to show the effect of iterative refinement. With this excessively high $\epsilon = 10^{-2}$, using 10 iterative refinement steps means the iteration limit is only reached in 2% of cases, as shown in Figure 3.3(d).

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

### 3.7 Conclusion

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

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

Historically, embedded convex optimization has been challenging and time-consuming to use. CVXGEN makes this process much simpler by letting users move from a high-level problem description to a fast, robust solver, with minimal effort. We hope that CVXGEN (or similar tools) will greatly increase the interest and use of embedded convex optimization.
CHAPTER 3. CODE GENERATOR IMPLEMENTATION

Figure 3.3: Iteration counts for 100,000 problem instances, with varying solver settings. Labels on x-axis are iteration counts; bar heights and labels are the number of problem instances requiring that many iterations.
Chapter 4

Code Generation for Signal Processing

4.1 Introduction

This chapter considers the application of code generation to signal processing. Convex optimization has a long history in signal processing, dating back to the 1960s. The history is described below in a little more detail; for some more recent applications, see for example the special issue of the *IEEE Journal on Selected Topics in Signal Processing* [Sig07].

Signal processing applications may be split into two categories. In the first, optimization is used for design, *i.e.*, to choose the weights or algorithm parameters for later use in a (typically linear) signal processing algorithm. A classical example is the design of finite impulse response (FIR) filter coefficients via linear programming (LP) [CRR69, Rab72] (see also the review article [Dav10]). In these design applications, the optimization must merely be fast enough to satisfy the designer; thus, optimization times measured in seconds, or even minutes, are usually sufficient. In the second category, convex optimization is used to process the signal itself, which (generally) yields a nonlinear algorithm; an early example is \( \ell_1 \) regularization for sparse reconstruction in geophysics [CM73, TBM79]. Most applications in this category are (currently) offline, as in geophysics reconstruction, so while faster is better, the optimization is not subject to the strict real-time deadlines that would arise in an online application. There are some exceptions; an early example is [ALW00], which describes the use of convex optimization in online adaptive filtering.

Recent advances in algorithms for solving convex optimization problems, along with great advances in processor power, have dramatically reduced solution times. Another
significant reduction in solution time may be obtained by using a solver customized for a particular problem family. As a result, convex optimization problems that 20 years ago might have taken minutes to solve can now be solved in microseconds.

This opens up several new possibilities. In the design context, algorithm weights can be re-designed or updated on fast timescales (say, kHz). Perhaps more exciting is the possibility that convex optimization can be embedded directly in signal processing algorithms that run online, with strict real-time deadlines, even at rates of tens of kHz. We will see that solving 10,000 modest sized convex optimization problems per second is entirely possible on a generic processor. This is quite remarkable, since solving an optimization problem is generally considered a computationally challenging task, and few engineers would consider an online algorithm, that requires the solution of an optimization problem at each step, to be feasible for signal rates measured in kHz.

Of course, for high-throughput or fast signal processing (say, an equalizer running at GHz rates) it is not feasible to solve an optimization problem in each step, and it may never be. But a large number of applications are now potentially within reach of new algorithms, in which an optimization problem is solved in each step, or every few steps. We imagine that in the future, more and more signal processing algorithms will involve embedded optimization, running at rates up to or exceeding tens of kHz. (We believe the same trend will take place in automatic control; see, e.g., [WB08, WB09a].)

The code generation techniques introduced in Chapter 2 and described in Chapter 3, apply immediately to a wide range of signal processing problems. In this chapter, we describe how convex optimization can be used in signal processing, and illustrate the idea of real-time embedded convex optimization with two simple examples. In the first example (§4.3), we show how to implement a nonlinear pre-equalizer for a system with input saturation. It pre-distorts the input signal so that the output signal approximately matches the output of a reference linear system. Our equalizer is based on a method called model predictive control [Mac02], which has been widely used in the process control industry for more than a decade. It requires the solution of a QP at each step. It would not surprise anyone to know that such an algorithm could be run at, say, 1 Hz (process control applications typically run with sample times measured in minutes); but we will show that it can easily be run at 1 kHz. This example illustrates how our ability to solve QPs with extreme reliability and speed has made new signal processing methods possible.

In the second example (§4.4), we show how a standard Kalman filter can be modified to handle occasional large sensor noises (such as those due to sensor failure or intentional jamming), using now-standard $\ell_1$-based methods. Those familiar with the ideas behind
compressed sensing (or several other related techniques) will not be surprised at the effectiveness of these methods, which require the solution of a QP at each time step. What is surprising is that such an algorithm can be run at tens of kHz.

In the next two subsections we describe some previous and current applications of convex optimization, in the two just-described categories of weight design and direct signal processing. Before proceeding, we note that the distinction between the two categories—optimization for algorithm weight design versus optimization directly in the algorithm itself—is not sharp. For example, widely used adaptive signal processing techniques [Say03, Hay96] adjust parameters in an algorithm (i.e., carry out re-design) online, based on the data or signals themselves. (Indeed, many adaptive signal processing algorithms can be interpreted as stochastic approximation or stochastic subgradient methods for solving an underlying convex optimization problem.)

### 4.1.1 Weight design via convex optimization

Convex optimization was first used in signal processing in design, *i.e.*, selecting weights or coefficients for use in simple, fast, typically linear, signal processing algorithms. In 1969, [CRR69] showed how to use LP to design symmetric linear phase FIR filters. This was later extended to the design of weights for 2-D filters [Lai07], and filter banks [NK96]. LP and SOCP can be used to design filters with magnitude specifications [AV02, WBV96], via spectral factorization.

Weight design via convex optimization can also be carried out for (some) nonlinear signal processing algorithms; for example, in a decision-feedback equalizer [KJB00]. Convex optimization can also be used to choose the weights in array signal processing, in which multiple sensor outputs are combined linearly to form a composite array output. Here the weights are chosen to give a desirable response pattern [LB97]. More recently, convex optimization has been used to design array weights that are robust to variations in the signal statistics or array response [VGL03, BO01, LSW03, LB05]. For another example of weight design, see [GSS+10].

Many classification algorithms from machine learning involve what is essentially weight design via convex optimization [HTF01]. For example, we might classify objects $x$ (say, images or email messages) into two groups by first computing a vector of features $\phi(x) \in \mathbb{R}^n$, then, in real-time, using a simple linear threshold to classify the objects: we assign $x$ to one group if $w^T \phi(x) \geq v$, and to the other group if not. Here $w \in \mathbb{R}^n$ and $v \in \mathbb{R}$ are weights, chosen by training from objects whose true classification is known. This offline training step often involves convex optimization. One widely used method is the
support vector machine (SVM), in which the weights are found by solving a large QP [Vap00, CST00]. While this involves solving a (possibly large) optimization problem to determine the weights, only minimal computation is required at run-time to compute the features and form the inner product that classifies any given object.

4.1.2 Signal processing via convex optimization

Recently introduced applications use convex optimization to carry out (nonlinear) processing of the signal itself. The crucial difference from the previous category is that speed is now of critical importance. Convex optimization problems are now solved in the main loop of the processing algorithm, and the total processing time depends on how fast these problems can be solved.

With some of these applications, processing time again matters only in the sense that ‘faster is better’. These are offline applications where data is being analyzed without strict time constraints. More challenging applications involve online solution, with strict real-time deadlines. Only recently has the last category become possible, with the development of reliable, efficient solvers and the recent increase in computing power.

One of the first applications where convex optimization was used directly on the signal is in geophysics [CM73, TBM79], where $\ell_1$ minimization was used for sparse reconstruction of signals. Similar $\ell_1$-techniques are widely used in total variation noise removal in image processing [ROF92, CG02, Cha04]. Other image processing applications include deblurring [BBTK08] and, recently, automatic face recognition [Kro09]. Other signal identification algorithms use $\ell_1$ minimization or regularization to recover signals from incomplete or noisy measurements [CRT05, Tro06, Don06]. Within statistics, feature selection via the Lasso [Tib96] uses similar techniques. The same ideas are applied to reconstructing signals with sparse derivative (or gradient, more generally) in total variation de-noising, and in signals with sparse second derivative (or Laplacian) [KKBG09]. A related problem is parameter estimation, where we fit a model to data. One example of this is fitting MA or ARMA models; here parameter estimation can be carried out with convex optimization [SMM99, DTS01, AV00].

Convex optimization is also used as a relaxation technique for problems that are essentially Boolean, as in the detection of faults [ZBG09, ZBG08], or in decoding a bit string from a received noisy signal. In these applications a convex problem is solved, after which some kind of rounding takes place to guess the fault pattern or transmitted bit string [FKW03, CT05, CR08]. For more on convex optimization for non-convex problems, see [LMS+10].
CHAPTER 4. SIGNAL PROCESSING

Many methods of state estimation can be interpreted as involving convex optimization. (Basic Kalman filtering and least-squares fall in this category, but since the objectives are quadratic, the optimization problems can be solved analytically using linear algebra.) In the 1970s, ellipsoidal calculus was used to develop a state estimator less sensitive to statistical assumptions than the Kalman filter, by propagating ellipsoids that contain the state [SS72, KV96]. The standard approach here is to work out a conservative update for the ellipsoid; but the most sophisticated methods for ellipsoidal approximation rely on convex optimization [BV04, §8.4]. Another recently developed estimation method is minimax regret estimation [EBTN04], which relies on convex optimization.

Convex optimization algorithms have also been used in wireless systems. Some examples here include online pulse shape design for reducing the peak or average power of a signal [DLW00], receive antenna selection in MIMO systems [DMP06], and performing demodulation by solving convex problems [SS10].

4.2 Code generation

Designing and prototyping a convex optimization-based algorithm requires choosing a suitable problem format, then testing and adjusting it for good application performance. In this prototyping stage, the speed of the solver is often nearly irrelevant; simulations can usually take place at significantly reduced speeds. In prototyping and algorithm design, the key is the ability to change the problem formulation rapidly while testing the application performance, often using real data. Parser-solvers, as introduced in §2.2.4, are ideal for such use, and reduce development time by freeing users from the need to translate their problem into the restricted standard form required by the solver.

Once prototyping is complete, however, the final code must often run much faster. Thus, a serious challenge in using real-time convex optimization is the creation of a fast, reliable solver for a particular application. It is possible to hand-code solvers that take advantage of the special structure of a problem family, but such work is tedious and difficult to get exactly right.

It is sometimes possible to use the (slow) code from the prototyping stage in the final algorithm. For example, the acceptable time frame for a fault detection algorithm may be measured in minutes, in which case the above prototype is likely adequate. Often, though, there are still advantages in having code that is independent of the particular modeling framework. On the other hand (and as previously mentioned), some applications require timescales that are faster than those achievable even with a very good generic solver; here
explicit methods may be the only option. We are left with a large category of problems
where a fast, automatically-generated solver would be extremely useful.

This is where we can apply automatic code generation, where a user, who is not nec-
essarily an expert in algorithms for convex optimization, can formulate and test a convex
optimization problem within a familiar high-level environment, and then request a custom
solver. The remainder of this chapter, therefore, assumes the existence of an automatic
code generator like that of Chapter 3, and explores how it can be used for signal processing.

4.3 Linearizing pre-equalization

Many types of nonlinear pre- and post-equalizers can be implemented using convex opti-
mization. In this section we focus on one example, a nonlinear pre-equalizer for a nonlinear
system with Hammerstein [HK86] structure: A unit saturation nonlinearity, followed by
a stable linear time-invariant system. It is shown in Figure 4.1. Our equalizer, shown in
Figure 4.2, has access to the scalar input signal \( u \), with a lookahead of \( T \) samples (or,
equivalently, with an additional delay of \( T \) samples), and generates the equalized input
signal \( v \). This signal \( v \) is then applied to the system, and results in the output signal \( y \).
The goal is to choose \( v \) so that the actual output signal \( y \) matches the reference output
\( y^{\text{ref}} \), which is the output signal that would have resulted without the saturation nonlin-
earity. This is shown in the block diagram in Figure 4.3, which includes the error signal
\( e = y - y^{\text{ref}} \). If the error signal is small, then our pre-equalizer, followed by the system,
gives nearly the same output as the reference system. Since the reference system is linear,
our pre-equalizer thus linearizes the system.

When the input peak is smaller than the saturation level of 1, the error signal is zero;
our equalizer only comes into play when the input signal peak exceeds 1. A baseline choice of pre-equalizer is none: We simply take $v = u$. We use this simple equalizer as a basis for comparison with the nonlinear equalizer we describe here. We’ll refer to the output produced without pre-equalization as $y_{\text{none}}$, and the corresponding error as $e_{\text{none}}$.

We now describe the system, and the pre-equalizer, in more detail. We use a state-space model for the linear system,

$$x_{t+1} = Ax_t + Bs(at(v_t)), \quad y_t = Cx_t,$$

with state $x_t \in \mathbb{R}^n$, where the unit saturation function is given by $\text{sat}(z) = z$ for $|z| \leq 1$, $\text{sat}(z) = 1$ for $z > 1$, and $\text{sat}(z) = -1$ for $z < -1$. The reference system is

$$x_{t+1}^{\text{ref}} = Ax_t^{\text{ref}} + Bu_t, \quad y_{t}^{\text{ref}} = Cx_{t}^{\text{ref}},$$

with state $x_t^{\text{ref}} \in \mathbb{R}^n$. Subtracting these two equations, we can express the error signal $e = y - y_{\text{ref}}$ via the system

$$\tilde{x}_{t+1} = A\tilde{x}_t + B(\text{sat}(v_t) - u_t), \quad e_t = C\tilde{x}_t,$$

where $\tilde{x}_t = x_t - x_t^{\text{ref}} \in \mathbb{R}^n$ is the state tracking error.

We now come to the main (and simple) trick: We assume that (or more accurately, our pre-equalizer guarantees that) $|v_t| \leq 1$. In this case $\text{sat}(v_t)$ can be replaced by $v_t$ above,
and we have
\[ \tilde{x}_{t+1} = A\tilde{x}_t + B(v_t - u_t), \quad e_t = C\tilde{x}_t. \]

We can assume that \( \tilde{x}_t \) is available to the equalizer; indeed, by stability of \( A \), the simple estimator
\[ \hat{x}_{t+1} = A\hat{x}_t + B(v_t - u_t) \]
satisfies \( \hat{x}_t \to \tilde{x}_t \) as \( t \to \infty \), so we can use \( \hat{x}_t \) in place of \( \tilde{x}_t \). In addition to \( \tilde{x}_t \), our equalizer uses a look-ahead of \( T \) samples on the input signal, i.e., \( v_t \) is formed with knowledge of \( u_t, \ldots, u_{t+T} \).

We use a standard technique from control, called model predictive control [Mac02], in which at time \( t \) we solve an optimization problem to ‘plan’ our input signal over the next \( T \) steps, and use only the first sample of our plan as the actual equalizer output. At time \( t \) we solve the optimization problem

\[
\begin{align*}
\text{minimize} & \quad \sum_{\tau=t}^{t+T} e_\tau^2 + \tilde{x}_{t+T+1}^T P \tilde{x}_{t+T+1} \\
\text{subject to} & \quad \tilde{x}_{\tau+1} = A\tilde{x}_\tau + B(v_\tau - u_\tau), \quad e_\tau = C\tilde{x}_\tau \\
& \quad |v_\tau| \leq 1, \quad \tau = t, \ldots, t+T,
\end{align*}
\]

(4.1)

with variables \( v_t, \ldots, v_{t+T} \in \mathbb{R} \) and \( \tilde{x}_{t+1}, \ldots, \tilde{x}_{t+T+1} \in \mathbb{R}^n \). The initial (error) state in this planning problem, \( \tilde{x}_t \), is known. The matrix \( P \), which is a parameter, is symmetric and positive semidefinite.

The first term in the objective is the sum of squares of the tracking errors over the time horizon \( t, \ldots, t+T \); the second term is a penalty for the final state error; it serves as a surrogate for the tracking error past our horizon, which we cannot know because we do not know the input beyond the horizon. One reasonable choice for \( P \) is the output Grammian of the linear system,
\[ P = \sum_{i=0}^{\infty} (A^i)^T C^T CA^i, \]
in which case we have
\[ \tilde{x}_{t+T+1}^T P \tilde{x}_{t+T+1} = \sum_{\tau=t+T+1}^{\infty} e_\tau^2, \]
provided \( v_\tau = u_\tau \) for \( \tau \geq t + T + 1 \).

Problem (4.1) is a QP. It can be modified in several ways; for example, we can add a
(regularization) term such as
\[
\rho \sum_{\tau=t+1}^{T+1} (v_{\tau+1} - v_{\tau})^2,
\]
where \( \rho > 0 \) is a parameter, to give a smoother post-equalized signal.

Our pre-equalizer works as follows. At time step \( t \), we solve the QP above. We then use \( v_t \), which is one of the variables from the QP, as our pre-equalizer output. We then update the error state as 
\[
\tilde{x}_{t+1} = A\tilde{x}_t + B(v_t - u_t).
\]

### 4.3.1 Example

We illustrate the linearizing pre-equalization method with an example, in which the linear system is a third-order lowpass system with bandwidth \( 0.1\pi \), with impulse response that lasts for about 35 samples. Our pre-equalizer uses a look-ahead horizon \( T = 15 \) samples, and we choose \( P \) as the output Gramian. We use smoothing regularization with \( \rho = 0.01 \). The input \( u \) a is lowpass filtered random signal, which saturates (i.e., has \(|u| > 1\)) around 20% of the time.

The unequalized and equalized inputs are shown in Figure 4.4. We can see that the pre-equalized input signal is quite similar to the unequalized input when there is no saturation, but differs considerably when there is. The corresponding outputs, including the reference output, are shown in Figure 4.5, along with the associated output tracking errors.

While the earlier paper [MB09] used CVXMOD, an early precursor to CVXGEN, here we report updated performance figures using CVXGEN. The QP (4.1), after transformation, has 111 variables, 63 equality constraints, and 78 inequality constraints. Using an Intel i7-860, it takes approximately 150 \( \mu \)s to solve using CVXGEN-generated code, which compares well with the standard SOCP solvers SDPT3 [TTT99, TTT03] and SeDuMi [Stu99], whose solve times are approximately 430 ms and 160 ms, respectively.

### 4.4 Robust Kalman filtering

Kalman filtering is a well known and widely used method for estimating the state of a linear dynamical system driven by noise. When the process and measurement noises are independent identically distributed (IID) Gaussian, the Kalman filter recursively computes the posterior distribution of the state, given the measurements.

In this section we consider a variation on the Kalman filter, designed to handle an additional measurement noise term that is sparse, i.e., whose components are often zero.
Figure 4.4: Input without pre-equalization (red, $u_t$), and with linearizing pre-equalization (blue, $v_t$).

Figure 4.5: Left: Output $y$ without pre-equalization (red), and with nonlinear pre-equalization (blue). The reference output $y_{ref}$ is shown as the dashed curve (black). Right: Tracking error $e$ with no pre-equalization (red) and with nonlinear pre-equalization (blue).
This term can be used to model (unknown) sensor failures, measurement outliers, or even intentional jamming. Our goal is to design a filter that is robust to such disturbances, i.e., whose performance does not degrade rapidly when disturbances are introduced. (This robustness is to additive measurement noise; see, e.g., [Ger99] for a discussion of filtering that is robust to model parameter variation.) Here we create a robust Kalman filter by replacing the standard measurement update—which can be interpreted as the result of solving a quadratic minimization problem—with the solution of a similar convex minimization problem that includes an $\ell_1$ term to handle the sparse noise. Thus the robust Kalman filter requires the solution of a convex optimization problem in each time step. Compare this to the standard Kalman filter, which requires the solution of an equality-constrained quadratic optimization problem at each step, and has an analytical solution expressible using basic linear algebra operations.

We work with the system

$$x_{t+1} = Ax_t + w_t, \quad y_t = C x_t + v_t + z_t,$$

where $x_t \in \mathbb{R}^n$ is the state (to be estimated) and $y_t \in \mathbb{R}^m$ is the measurement available to us at time step $t$. As in the standard setup for Kalman filtering, the process noise $w_t$ is IID $\mathcal{N}(0, W)$, and the measurement noise term $v_t$ is IID $\mathcal{N}(0, V)$. The term $z_t$ is an additional noise term, which we assume is sparse (meaning, most of its entries are zero) and centered around zero. Without the additional sparse noise term $z_t$, our system is identical to the standard one used in Kalman filtering.

We use the standard notation from the Kalman filter: $\hat{x}_{t|t}$ and $\hat{x}_{t|t-1}$ denote the estimates of the state $x_t$, given the measurements up to $y_t$ or $y_{t-1}$, and $\Sigma$ denotes the steady-state error covariance associated with predicting the next state. In the standard Kalman filter (i.e., without the additional noise term $z_t$), all variables are jointly Gaussian, so the (conditional) mean and covariance specify the conditional distributions of $x_t$, conditioned on the measurements up to $y_t$ and $y_{t-1}$, respectively.

The standard Kalman filter consists of alternating time and measurement updates. The time update,

$$\hat{x}_{t|t-1} = A \hat{x}_{t-1|t-1},$$

(4.2)

propagates forward the state estimate at time $t - 1$, after the measurement $y_{t-1}$, to the state estimate at time $t$, but before the measurement $y_t$ is known. The measurement update,

$$\hat{x}_{t|t} = \hat{x}_{t|t-1} + \Sigma C^T (C \Sigma C^T + V)^{-1} (y_t - C \hat{x}_{t|t-1}),$$

(4.3)
then gives the state estimate at time $t$, given the measurement $y_t$, starting from the state estimate at time $t$, before the measurement is known. In the standard Kalman filter, $\hat{x}_{t|t-1}$ and $\hat{x}_{t|t}$ are the conditional means, and so can be interpreted as the minimum mean-square error estimates of $x_t$, given the measurements up to $y_{t-1}$ and $y_t$, respectively.

To (approximately) handle the additional sparse noise term $z_t$, we modify the Kalman filter measurement update (4.3) and call the new filter robust. To motivate the modification, we first note that $\hat{x}_{t|t}$ can be expressed as the solution of a quadratic optimization problem,

$$\begin{align*}
\min v_t^T V^{-1} v_t + (x - \hat{x}_{t|t-1})^T \Sigma^{-1} (x - \hat{x}_{t|t-1}) \\
\text{subject to } y_t = Cx + v_t,
\end{align*}$$

(4.4)

with variables $x$ and $v_t$. We can interpret $v_t$ as our estimate of the sensor noise; the first term in the objective is a loss term corresponding to the sensor noise, and the second is a loss term associated with our estimate deviating from the prior.

In the robust Kalman filter, we take $\hat{x}_{t|t}$ to be the solution of the convex optimization problem

$$\begin{align*}
\min v_t^T V^{-1} v_t + (x - \hat{x}_{t|t-1})^T \Sigma^{-1} (x - \hat{x}_{t|t-1}) + \lambda \|z_t\|_1 \\
\text{subject to } y_t = Cx + v_t + z_t,
\end{align*}$$

with variables $x$, $v_t$, and $z_t$. A computationally more efficient but equivalent method [MWB10] is to precompute $L = \Sigma C (C \Sigma C^T + V)^{-1}$ and $Q = (I - CL)^TV^{-1}(I - CL) + L^T \Sigma^{-1} L$, and then at each time-step set $e_t = y_t - C \hat{x}_{t|t-1}$ and solve

$$\min (e_t - z_t)^T Q (e_t - z_t) + \lambda \|z_t\|_1,$$

(4.5)

with variable $z_t \in \mathbb{R}^m$. (Standard methods can be used to transform this problem into an equivalent QP.) We may then recover $x = \hat{x}_{t|t-1} + L(e_t - z_t)$.

Here we interpret $v_t$ and $z_t$ as our estimates of the Gaussian and the sparse measurement noises, respectively. The parameter $\lambda \geq 0$ is adjusted so that the sparsity of our estimate coincides with our assumed sparsity of $z_t$. For $\lambda$ large enough, the solution of this optimization problem has $z_t = 0$, and so is exactly the same as the solution of (4.4); in this case, the robust Kalman filter measurement update coincides with the standard Kalman filter measurement update.

In the robust Kalman filter, we use the standard time update (4.2), and the modified measurement update, which requires the explicit solution of the convex optimization problem (4.5). With this time update, the estimation error is not Gaussian, so the estimates $\hat{x}_{t|t}$ and $\hat{x}_{t|t-1}$ are no longer conditional means (and $\Sigma$ is not the steady-state state estimation
CHAPTER 4. SIGNAL PROCESSING

4.4.1 Example

For this example, we randomly generate matrices $A \in \mathbb{R}^{50 \times 50}$ and $C \in \mathbb{R}^{15 \times 50}$. We scale $A$ so its spectral radius is 0.98. We generate a random matrix $B \in \mathbb{R}^{50 \times 5}$ with entries $\sim \mathcal{N}(0,1)$, and use $W = BB^T$ and $V = I$. The sparse noise $z_t$ was generated as follows: with probability 0.05, component $(y_t)_i$ is set to $(v_t)_i$; i.e., the signal component is removed. This means that $z \neq 0$ with probability 0.54, or, roughly, one in two measurement vectors contains at least one bogus element. We compare the performance of a traditional Kalman filter tuned to $W$ and $V$, with the robust Kalman filter described above, and show example traces of the errors in Figure 4.6. In this example, the RMS error of the robust Kalman filter is approximately one quarter that of the Kalman filter.

For this example, the measurement update (4.5) is transformed into a QP with 30 variables, no equality constraints, and 30 inequality constraints. Code generated by CVXGEN solves this problem in approximately 30 $\mu$s, which allows measurement updates at rates better than 30 kHz. Solution with SDPT3 or SeDuMi takes 120 ms or 80 ms, while a standard Kalman filter update takes 6 $\mu$s.

4.5 Conclusion

This chapter shows the potential for convex optimization methods to be much more widely used in signal processing. In particular, automatic code generation makes it easier to create convex optimization solvers that are made much faster by being designed for a
specific problem family. Much work remains to be done in exploring the capabilities and limitations of automatic code generation. As computing power increases, and as automatic code generation improves, we expect convex optimization solvers to be found more and more often in real-time signal processing applications.
Chapter 5

Code Generation for Receding Horizon Control

5.1 Introduction

Receding horizon control (RHC) or model predictive control (MPC) [KH05, Whi82, GSD05, Mac02] is a form of feedback control system that first became popular in the 1980s. With RHC, we solve an optimization problem at each time step to determine a plan of action over a fixed time horizon, and then apply the first input from this plan. At the next time step we repeat this, solving a new optimization problem, with the time horizon shifted one step forward. Estimates of future quantities, based on available measurements and data, enter the optimization; this provides feedback (i.e., the use of real-time measurements or other information in determining the input).

RHC is a nonlinear control policy that naturally handles input constraints, output constraints, and a variety of control objectives. Systems can thus be controlled near their physical limits, obtaining performance superior to linear control. RHC has given excellent results in a wide range of practical settings, including industrial and chemical process control [QB03], supply chain management [CTHK03], stochastic control in economics and finance [Her05], electronics [ONP+09], and revenue management [TR04].

A drawback of RHC is the comparatively long time required to solve the planning problem using conventional numerical optimization techniques, as compared to, say, the time required to compute the control action in a traditional linear controller. Thus, RHC has been mostly limited to slow systems with sample times measured in seconds, minutes,
or hours. Many methods have been proposed to speed up the solution of the optimization problems that arise in RHC. When the problem dimensions (the numbers of states, inputs, and constraints) are small, one approach is explicit MPC [BF04, BMDP02], where symbolic solutions are generated offline and saved for later use. The online algorithm then reduces to a table lookup, followed by a simple linear control law evaluation, which can be made extremely fast. Another method, applicable to a problem of any size, is to code custom online optimization solvers that exploit the particular problem structure arising in RHC applications [WB08, DFS+02, AH04, RWR04]. These custom solvers can yield computation times that are several orders of magnitude faster than generic solvers, but require time-consuming hand coding and significant expertise in optimization algorithms and numerical computation.

In this chapter, we use code generation to produce custom RHC solvers. Code generation makes it easy for a user of RHC to specify and generate fast, reliable custom code. Since the user does not require much optimization expertise, many more people can use RHC, and in new settings—including applications with kilohertz sample rates.

We do not claim that RHC, or any other modern control method, will always outperform traditional control methods. In many cases, a skilled designer can achieve similar performance by carefully tuning a conventional PID (proportional-integral-derivative) or other linear controller, suitably modified to handle the constraints. In our opinion, the main advantage of RHC is the simple design process, which handles constraints directly (indeed, by simply specifying them), and requires far less tweaking and adjustment than is typically required with conventional controller design. Roughly speaking, with RHC the designer simply lists the constraints, whereas in a conventional design process, the designer tweaks controller gains (or design weights in a modern method) to handle the constraints indirectly. We believe RHC via automatic code generation offers an attractive framework for rapid design of sophisticated controllers; especially for problems with challenging constraints, and even for problems with relatively fast sample rates.

In the remainder of the chapter, we give a high-level overview of RHC, briefly explain code generation for RHC using CVXGEN (as introduced in Chapter 3) and illustrate the ideas with four examples. The examples are simple, and chosen to show the variety of problems that can be addressed.

We restrict our attention to systems with linear dynamics and convex objectives and constraints, for several reasons. First, many real systems can be reasonably modeled in this restricted form. Secondly, standard linearization techniques can be used to extend these methods to many nonlinear systems. (Indeed, almost all commercial MPC systems
for process control rely on linearization around an operating point.) And finally, many of the techniques we discuss could be applied to general nonlinear systems. For some work in this area, see [RS07], which describes the software package NEWCON; an example of automatic code generation applied to nonlinear RHC [OK02], or a more recent approach by the same author applied to a two-link robot arm in [Oht04]. Also see the ACADO system [HF08].

5.2 Formulating RHC problems

5.2.1 System dynamics and control

System dynamics. Each of the examples in this chapter applies RHC to a discrete-time linear dynamical system of the form

$$x_{t+1} = A_t x_t + B_t u_t + c_t,$$

where $x_t \in \mathbb{R}^n$ is the system state, $u_t \in \mathbb{R}^m$ is the control action or input, and $c_t \in \mathbb{R}^n$ is an exogenous input. The matrices $A_t \in \mathbb{R}^{n \times n}$ and $B_t \in \mathbb{R}^{n \times m}$ are the dynamics and input matrices, respectively. The subscripts on $A_t$, $b_t$, and $c_t$ indicate that they may change with time, but in many applications some of these data are constant and known.

Constraints and objective. The state and input must satisfy some constraints, expressed abstractly as

$$(x_t, u_t) \in \mathcal{C}_t,$$

where $\mathcal{C}_t \subseteq \mathbb{R}^n \times \mathbb{R}^m$ is the constraint set. The instantaneous cost depends on both the current state and control action, and is denoted $\ell_t(x_t, u_t)$. We judge the quality of control by the average cost,

$$J = \lim_{T \to \infty} \frac{1}{T} \sum_{t=0}^{T-1} \ell_t(x_t, u_t),$$

where we assume the limit exists. If $\ell_t(x_t, u_t)$ is a random variable, we replace $\ell_t(x_t, u_t)$ by $E \ell_t(x_t, u_t)$. Like the dynamics data, we subscript the constraint set and objective function with the time $t$, to handle the case when they vary with time. But in many applications they are constant and known.
Information available for control. The control input $u_t$ is determined using the information available to the controller at time $t$, including estimates of any quantities (or functions) that are not known, based on information that is known. We denote these estimates as

$$
\hat{A}_\tau|t, \quad \hat{B}_\tau|t, \quad \hat{c}_\tau|t, \quad \hat{C}_\tau|t, \quad \hat{\ell}_\tau|t, \quad \hat{x}_t|t,
$$

where the notation $\hat{z}_\tau|t$ means our estimate of $z_\tau$, based on information available to us at time $t$, where $\tau \geq t$. Information available at time $t$ includes conventional data in a control system, such as those available from sensor measurements, or known coefficients. It can also include other relevant information, such as historical usage patterns, weather, and price trends, which are not traditional data in control systems.

These estimates can be obtained in many ways. In the simplest case, we know the quantity being estimated, in which case we simply replace the estimates with the known value. For example, if the system dynamics matrices $A_t$ and $B_t$ have known and constant values $A$ and $B$, we take $\hat{A}_\tau|t = A$ and $\hat{B}_\tau|t = B$. If the controller has access to the (exact) current state $x_t$, we take $\hat{x}_t|t = x_t$.

A traditional method for obtaining the estimates is from a statistical model of the unknown data, in which case the estimates can be conditional expectations or other statistical estimates, based on the data available at time $t$. For example, the additive terms $c_t$ are often modeled as independent zero mean random variables, with natural estimate $\hat{c}_\tau|t = 0$.

The estimates need not be derived from statistical models; for example, future prices (entering the objective through $\ell_t$, say) could be obtained from a futures market, or from analysts who predict trends. Another source of the estimates comes up when the system to be controlled is nonlinear. In this case $\hat{A}_\tau|t$, $\hat{B}_\tau|t$, and $\hat{c}_\tau|t$ can be a linearization of (nonlinear) dynamics, along a trajectory.

Controller design problem. The controller design problem is to find a control policy or control law that chooses the input $u_t$ as a function of the quantities known at time $t$, in such a way that the constraints are always (or almost always) satisfied, and that the average cost $J$ is minimized, or at least made small.

We have not fully specified the uncertainty model, so our description of the control problem is informal, and we cannot really talk about an optimal control policy. But when we give a full mathematical description of the uncertainty, for example as a statistical model, we can talk about the optimal control policy, i.e., the policy that minimizes $J$, among all policies that map the information available into a control action while respecting
the constraints.

5.2.2 Receding horizon control

The basic RHC policy is very simple. At time $t$, we consider an interval extending $T$ steps into the future: $t, t+1, \ldots, t+T$. We then carry out several steps (which we briefly summarize, then describe again in more detail):

1. **Form a predictive model.** Replace all unknown quantities over the time interval with their current estimates, using all data available at time $t$.

2. **Optimize.** Solve the problem of minimizing the (predicted) objective, subject to the (predicted) dynamics and constraints.

3. **Execute.** Choose $u_t$ to be the value obtained in the optimization problem of step 2.

**Steps 1 and 2.** The RHC optimization problem in step 2 takes the form

$$
\begin{align*}
\text{minimize} & \quad \frac{1}{T+1} \sum_{\tau=t}^{t+T} \hat{h}_\tau(\hat{x}_\tau, \hat{u}_\tau) \\
\text{subject to} & \quad \hat{x}_{\tau+1} = \hat{A}_{\tau|t}\hat{x}_\tau + \hat{B}_{\tau|t}\hat{u}_\tau + \hat{c}_{\tau|t}, \\
& \quad (\hat{x}_\tau, \hat{u}_\tau) \in \hat{C}_{\tau|t}, \quad \tau = t, \ldots, t+T \\
& \quad \hat{x}_t = \hat{x}_{t|t},
\end{align*}
$$

with variables $\hat{x}_t, \ldots, \hat{x}_{t+T}$ and $\hat{u}_t, \ldots, \hat{u}_{t+T}$. The data in this RHC optimization problem are the estimates

$$
\hat{A}_{\tau|t}, \quad \hat{B}_{\tau|t}, \quad \hat{c}_{\tau|t}, \quad \hat{h}_\tau|t,
$$

for $\tau = t, \ldots, t+T$, and the current state estimate, $\hat{x}_{t|t}$. (In most applications, we can use known, exact values for many of the parameters.)

We can interpret $\hat{u}_t^*, \ldots, \hat{u}_{t+T}^*$, the optimal values from the RHC optimization problem (5.1), as a *plan of action* for the next $T$ steps.

**Step 3.** We then choose $u_t = \hat{u}_t^*$ to be our RHC action.

At the next time step, the process is repeated, with (possibly) updated estimates of the current state and future quantities. We make a few comments about the RHC policy.

**Terminal constraints or cost terms.** It is common to add a final state constraint, or an extra final state cost, to the RHC problem. In the former case, we add an equality constraint of the form $x_{T+1} = x^{\text{final}}$, or a final constraint set condition $x_{T+1} \in C^{\text{final}}$. In
the latter case, we add $V(x_{T+1})$ to the objective, with $V$ a cost function for the final state. This can allow simpler, shorter-horizon controllers to approximate the behavior of controllers with longer horizons.

**Optimality.** The RHC policy is generally not an optimal control policy, even when we have a formal model of the uncertainty. Instead, RHC is merely a (sophisticated) heuristic that works very well in many applications.

**Convexity.** We assume that $C_t$ and $\ell_t$ are convex, which means that the RHC problem (5.1) is a convex optimization problem. This means that we can solve it efficiently, using standard convex optimization tools [BV04].

**Requirements.** To specify an RHC policy, we must describe the prediction method (i.e., the method for estimating unknown quantities from current data), the horizon $T$, and any terminal cost or constraint.

### 5.2.3 Computer modeling of RHC

The two substantial design tasks required for RHC are system modeling, and creating the method to solve the optimization problem (5.1). The former task involves choosing a system model and appropriate cost functions and constraints, and has been tackled extensively in the literature [Joh93, LL87, Nel02]. A wide range of computing tools are available to accelerate development of that phase. That leaves the onerous task of solving (5.1) at an acceptable speed. This is needed during development and testing, and, especially, for implementation in a real-time system. Many convenient software tools [Lö4, GB08a, MB08] are available for solving convex problems, during development and testing. These *parser-solvers* take a high-level specification and perform the necessary transformations for solution by a standard convex optimization solver, *e.g.*, [TTT99, TTT03, Stu99]. This allows quick iteration, with the engineer able to change the optimization problem and immediately see results. However, during development, the demands on the software are not onerous, since an engineer is ‘in the loop’, and must formulate and evaluate each design. Thus, high speed or accuracy is not especially relevant.

Once a design is final and it is time for deployment, solver speed can be of critical importance, particularly when the sampling rate is high. If the solver speed is much faster than that required, we can use a less powerful processor, or a processor performing other tasks. This is why we use automatic code generation, with CVXGEN.
5.3 Examples

5.3.1 Pre-ordering

**Problem statement.** We consider the problem of meeting a fluctuating demand for a perishable commodity by pre-ordering it with different lead times and also purchasing it on the spot market, all at (possibly) different prices. When we place an order, we specify delivery for between 1 and $n$ periods in the future. Faster delivery typically incurs a higher unit cost.

Let $u_t \in \mathbb{R}^n_+$ represent new orders, where $(u_t)_i$ is the amount, ordered in period $t$, to be delivered in period $t + i$. Our state is the order book $x_t \in \mathbb{R}^n_+$, where $(x_t)_i$ is the quantity scheduled to arrive in period $t + i - 1$; in particular, $(x_t)_1$ is the stock at hand.

The system dynamics are $x_{t+1} = Ax_t + Bu_t$, where $A$ is a matrix with ones on the upper diagonal and zeros everywhere else, and $B = I$. The constraint has the form $u_t \geq 0$, which is convex.

Our stage cost has two terms: The cost of placing orders for future delivery (which we recognize immediately), and the cost of making up any unmet demand by purchasing on the spot market. The first term has the form $p_t^T u_t$, where $(p_t)_i \geq 0$ is the price of ordering one unit of the commodity for delivery in period $t + i$. The unmet demand is $(d_t - (x_t)_1)_+$, where $d_t \geq 0$ is the demand in period $t$, and $(\cdot)_+$ denotes the positive part. The cost of meeting the excess demand on the spot market is $p_t^{\text{spot}} (d_t - (x_t)_1)_+$, where $p_t^{\text{spot}} \geq 0$ is the spot market price at time $t$. Thus the overall stage cost is

$$
\ell_t(x_t, u_t) = p_t^T u_t + p_t^{\text{spot}} (d_t - (x_t)_1)_+,
$$

which is a convex function of $x_t$ and $u_t$. Typically the prices satisfy $p_t^{\text{spot}} > (p_t)_1 > \cdots > (p_t)_n$, i.e., there is a discount for longer lead time.

We consider the simple case in which the pre-order and spot market prices are known and do not vary with time (i.e., $p_t = p \in \mathbb{R}^n_+$, $p_t^{\text{spot}} = p^{\text{spot}} \geq 0$), and demand is modeled as a stochastic process. We assume that demand is a stationary log-normal process, i.e., that $\log d_t$ is a stationary Gaussian process with

$$
E \log d_t = \mu, \quad E(\log d_t - \mu)(\log d_{t+\tau} - \mu)) = r_\tau,
$$
so the mean demand is $E d_t = \exp(\mu + r_0/2)$.

In period $t$, the controller has access to the current order book $x_t$, and the current and last $N$ values of demand, $d_t, d_{t-1}, \ldots, d_{t-N}$, in addition to the various constants: the prices $p$ and $p_{\text{spot}}$, the log demand mean $\mu$, and the log demand autocovariances $r_\tau$. The orders made in period $t$ are based on this information.

**Receding horizon policy.** Our RHC policy requires estimates of future stage cost, which depends on the (unknown) future demand. We take

$$\hat{d}_{\tau|t} = \exp \left( a_{\tau-t} (\log d_t, \ldots, \log d_{t-N}) + b \right),$$

i.e., the exponential of the conditional mean of log demand, given the previous $N$ demand values. (Since we know the current demand, we simply take $\hat{d}_{\tau|t} = d_t$.) Since we have assumed the demand is a stationary log-normal process, the conditional expectation of log $d_\tau$ is an affine (linear plus constant) function of $\log d_t, \ldots, \log d_{t-N}$:

$$\hat{d}_{\tau|t} = \exp \left( a_{\tau-t} (\log d_t, \ldots, \log d_{t-N}) + b \right), \ldots,$$

for $\tau = t+1, \ldots, t+T$, where $a_j \in \mathbb{R}^{N+1}$ and $b \in \mathbb{R}$ can be found from the data $\mu$ and $r_0, \ldots, r_{N+T+1}$.

For this example we add a terminal constraint, $1^T \hat{x}_{t+T+1} = nE d_t$, where $E d_t = \exp(\mu + r_0/2)$. This ensures we do not myopically reduce cost by exhausting inventory at the end of the horizon.

The RHC optimization problem (5.1) becomes

\[
\begin{align*}
\text{minimize} & \quad \frac{1}{T+1} \sum_{\tau=t}^{t+T} \left( p^T \hat{u}_\tau + p_{\text{spot}} \left( \hat{d}_{\tau|t} - (\hat{x}_\tau)_1 \right) \right)_+ \\
\text{subject to} & \quad \hat{x}_{\tau+1} = A \hat{x}_\tau + \hat{u}_\tau, \quad \tau = t, \ldots, t+T \\
& \quad \hat{u}_\tau \geq 0, \quad \tau = t, \ldots, t+T \\
& \quad 1^T \hat{x}_{t+T+1} = nE d_t, \quad \hat{x}_t = x_t,
\end{align*}
\]

with variables $\hat{x}_t, \ldots, \hat{x}_{t+T+1}$ and $\hat{u}_t, \ldots, \hat{u}_{t+T}$. This is a convex optimization problem, and can be reduced to a linear program (LP).

**Constant order policy.** We compare the RHC policy with a simple policy: At each time $t$, we let $u_t = (0, \ldots, 0, \bar{u})$, i.e., we order a constant amount with the maximum delivery time. We use $\bar{u} = E d_t = \exp(\mu + r_0/2)$, i.e., we order with maximum lead-time (presumably, at the lowest price) an amount equal to the average demand.
CHAPTER 5. RECEDING HORIZON CONTROL

Related work

Much work has been done on supply chain planning. For an overview of the field, though without the optimization component, see [Mil02]. For the application of RHC to the supply chain, see [SWR06], or [CTHK03], which covers multi-factory supply chains. In [BP00], the authors use extensive simulation of MPC to test the sensitivity of various policies, while [MTA06] explores various levels of decentralization. Finally, for supply chain optimization with mixed-integer constraints see [PLYG03], and for planning under uncertainty see [GM03].

Numerical example

Our example has $n = 5$ order lead times, with prices

$$p_{\text{spot}} = 1, \quad p = (\gamma, \gamma^2, \gamma^3, \gamma^4, \gamma^5),$$

with $\gamma = 0.7$. (Thus, we get a constant 30% discount for each period of lead time.) The demand process data are $\mu = 0$ and $r_T = 0.1(0.95^T)$. Our RHC controller will use horizon $T = 30$, and we estimate future demand using the last $N = 100$ demands.

Results. We simulate both the RHC and constant ordering policies for 1000 steps (with the same demand realization). The constant order policy incurs an average cost $J = 0.37$, while, as expected, the RHC policy performs considerably better, with an average cost $J = 0.28$. Some example trajectories are shown in Figure 5.2. We compare the costs incurred by the RHC policy (blue) and constant policy (red), over 500 time steps. The plots show demand, pre-order cost and spot market costs, and overall stage cost.

In Figure 5.3 we show the log-demand trajectories for a selected time region. The vertical lines show $\exp(\log \hat{d}_{i|220} \pm \sigma_t)$, where $\sigma_t = (E(\log d_t - \log \hat{d}_{i|220}))^{1/2}$. While the predicted trajectory captures the general trend, the prediction error is large, so control is relatively difficult.

The CVXGEN code takes up to 250 $\mu$s to solve at each time step, which is $4000 \times$ faster than with plain CVX. This speed is far faster than would ever be required. However, it means that we could use extensive Monte-Carlo simulation to test different scenarios and ordering strategies. Computation performance details are collected in Table 5.2.
5.3.2 Active suspension control

Problem statement. We consider an active vehicle suspension system, where the control system applies force to a suspension system in real-time, using a preview of the terrain height profile obtained from (say) sensors or maps. For simplicity, we consider a scalar problem, with a single, scalar vehicle height, and one actuator. (A real system would use different heights and actuator forces on each wheel, and more complicated dynamic coupling between them.)

Let $h_t \in \mathbb{R}$ denote the terrain height, $y_t \in \mathbb{R}$ the vehicle height, and $e_t = y_t - h_t$ the suspension extension, at discrete time period $t$. The extension is offset, with the suspension at rest when $e_t = 0$. The suspension has minimum extension $E_{\min} < 0$ and maximum extension $E_{\max} > 0$, so we always have

$$E_{\min} \leq e_t \leq E_{\max},$$

(The vehicle would ‘bottom out’ if the first inequality were violated, and would become
airborne if the second inequality were violated.)

The vehicle dynamics are

\[ x_{t+1} = Ax_t + bu_t + v_t, \]

where \( A \in \mathbb{R}^{n \times n} \) and \( b \in \mathbb{R}^n \) are known dynamic model coefficients, \( x_t \in \mathbb{R}^n \) is the vehicle dynamic state, \( v_t \in \mathbb{R}^n \) is the exogenous force applied to the vehicle by the changing terrain, and \( u_t \in \mathbb{R} \) is the active suspension actuator force, which must satisfy

\[ F_{\text{min}} \leq u_t \leq F_{\text{max}}, \]

where \( F_{\text{min}} < 0 \) and \( F_{\text{max}} > 0 \) are given minimum and maximum active suspension forces.

The vehicle’s height \( y_t \) and vertical acceleration \( a_t \) are given by

\[ y_t = c^T x_t, \quad a_t = d^T x_t + g^T u_t + w_t, \]
where $c$, $d$ and $g \in \mathbb{R}^n$ are known model coefficients, and $w_t \in \mathbb{R}$ is a linear function of the terrain height and gradient at time $t$.

Our goal is to minimize a weighted sum of the squared acceleration, the squared active suspension force, and a penalty term that returns the suspension to equilibrium. This can be thought of as a measure of the ‘ride roughness’, penalized to avoid excessive suspension effort. Our cost function is

$$\ell(x_t, u_t) = a_t^2 + \rho u_t^2 + \mu e_t^2,$$

where the parameters $\rho > 0$ and $\mu > 0$ control the effort and extension penalties.

**Receding horizon policy.** At time $t$, the controller has access to an estimate of the current vehicle state $\hat{x}_{t|t}$, and a preview of upcoming terrain, i.e., $\hat{h}_{\tau|t}$ for $\tau = t, \ldots, t + L$, where $L$ is the look-ahead time. It also has access to exogenous input estimates $\hat{v}_{\tau|t}$ and $\hat{w}_{\tau|t}$, which are formed using the terrain height estimates, along with estimates of the terrain’s gradient. The actuator force to apply at the current time step, $u(t)$, is determined
from these data. In our RHC formulation, we add the terminal constraint $x_{t+T+1} = 0$, which means our suspension returns to a neutral position at the end of the look-ahead interval. The RHC optimization problem (5.1) becomes

$$\begin{align*}
\text{minimize} & \quad \frac{1}{T+1} \sum_{\tau = t}^{t+T} \left( a_\tau^2 + \rho u_\tau^2 + \mu (y_\tau - \hat{h}_\tau|_t)^2 \right) \\
\text{subject to} & \quad x_{\tau+1} = Ax_\tau + bu_\tau + \hat{v}_\tau|_t, \quad y_\tau = c^T x_\tau, \quad a_\tau = d^T x_\tau + gu_\tau + \hat{w}_\tau|_t, \\
& \quad E_{\text{min}} \leq y_\tau - \hat{h}_\tau|_t \leq E_{\text{max}}, \quad F_{\text{min}} \leq u_\tau \leq F_{\text{max}}, \quad \tau = t, \ldots, t+T \\
& \quad x_{t+T+1} = 0, \quad x_t = \hat{x}_t|_t,
\end{align*}$$

with variables $x_\tau, \ldots, x_{\tau+T} \in \mathbb{R}^n$, $y_\tau, \ldots, y_{\tau+T} \in \mathbb{R}$, $a_\tau, \ldots, a_{\tau+T} \in \mathbb{R}$, and $u_\tau, \ldots, u_{\tau+T} \in \mathbb{R}$. This is a convex optimization problem, and can be solved as a QP.

**Uncontrolled policy.** We compare the RHC policy to the performance of the uncontrolled mass-spring-damper system.

**Related work**

Active suspension control is used in a range of production vehicles, although with less sophisticated control algorithms [Wil97]. In [MAH+97], the authors use a model similar to ours but also incorporating unsprung mass, and solve the problem as a QP. A number of authors consider semi-active suspension to reduce cost, including [GBTH05] and [CMN06], which addresses MPC to control passive damping, and [DSL05], which uses $H_\infty$ control.

**Numerical example**

Our example has a sampling rate of 20 Hz, and a horizon $T = 20$ to give a one-second planning horizon. To determine $A$, $b$, $c$, $d$ and $g$, we use a quarter-vehicle model, with mass 2000 kg, damping coefficient 5 kNs/m, and spring constant 30 kN/m. This gives a natural frequency of 0.6 Hz and a light damping coefficient of 0.3. We assume forces are held constant during each time interval, and use an exact discretization via the matrix exponential. We set the negative and positive extension limits to $E_{\text{min}} = -0.1$ and $E_{\text{max}} = 0.1$ m, and restrict our maximum actuator force to 3 kN. We set $\rho = 10^{-4}$ and $\mu = 5$. Figure 5.4 shows CVXGEN code for this example.
CHAPTER 5. RECEIVING HORIZON CONTROL

\[ \begin{align*}
\text{dimensions} \\
n &= 2 \\
T &= 20 \\
\end{align*} \]

\[ \begin{align*}
\text{parameters} \\
A \ (n, n); & \quad B \ (n, 1) \\
C \ (1, n); & \quad D \\
\end{align*} \]

\[ \begin{align*}
h[t], \ t &= 0..T+1 \\
v[t] \ (n), \ t &= 0..T \\
w[t], \ t &= 0..T \\
\rho \ \text{positive}; & \quad \mu \ \text{positive} \\
F_{\min}; & \quad F_{\max}; \\
E_{\min}; & \quad E_{\max} \\
x[0] \ (n) \\
\end{align*} \]

\[ \begin{align*}
\text{variables} \\
u[t], \ t &= 0..T \\
x[t] \ (2), \ t &= 1..T+1 \\
a[t], \ t &= 0..T \\
\end{align*} \]

\[ \begin{align*}
\text{minimize} \\
\sum_{t=0}^{T}(\text{square}(a[t]) + \rho \star \text{square}(u[t]) + \mu \star \text{square}(x[t][1] - h[t])) \\
\text{subject to} \\
x[t+1] &= A \star x[t] + B \star u[t] + v[t], \ t &= 0..T \\
a[t] &= C \star x[t] + D \star u[t] + w[t], \ t &= 0..T \\
F_{\min} &= u[t] \leq F_{\max}, \ t &= 0..T \\
E_{\min} &= x[t][1] - h[t] \leq E_{\max}, \ t &= 1..T \\
\end{align*} \]

Figure 5.4: CVXGEN code segment for the suspension example.

**Results.** Figure 5.5 shows vehicle and suspension behavior for a 5-second journey over terrain with a single, large bump. Our control system reduces the vehicle’s root-mean-square acceleration value by approximately 70%. Note that the uncontrolled system exceeds the extension limits twice, which is avoided by our controller.

The CVXGEN code always takes less than 1 ms to solve at each time step, even on a low-power (2 W) processor. This is comfortably faster than required for the 20 Hz sample rate. This means that suspension control could be one of multiple jobs handled on a single computer with a suitable real-time operating system. Further performance details are in Table 5.2.
Figure 5.5: (From top) Vertical position (m), suspension extension (m), suspension force (kN) and acceleration (ms$^{-2}$). RHC (blue), open loop (red). Dashed lines indicate the terrain profile (top) and constraints.
5.3.3 Processor speed control

Problem statement. In this example, a single processor handles jobs from a set of \( n \) queues. At each time step the processor adjusts the work rate for each queue. The total work rate determines the processor (clock) speed, which in turn determines the power dissipated by the processor. The goal is to adjust the rates to balance average processor power dissipation and queue length.

We use a discrete-time formulation, with state \( x_t \in \mathbb{R}^n_+ \) and input \( u_t \in \mathbb{R}^n_+ \), where \( (x_t)_i \) is the amount of work to be done in queue \( i \), and \( (u_t)_i \) is the work rate (expressed as effective speed) for queue \( i \), at time \( t \). The dynamics are \( x_{t+1} = x_t - u_t + a_t \), where \( a_t \in \mathbb{R}^n_+ \) denotes the new work arriving in each queue between time periods \( t \) and \( t+1 \). At each time we cannot process more than the available work in each queue, so we must have \( u_t \leq x_t \). The total work rate of the processor, over all queues, is \( 1^T u_t \).

The processor speed at time \( t \) is a function of the work rate vector \( u_t \):

\[
s_t = \max\{S^{\text{min}}, 1^T u_t\},
\]

where \( S^{\text{min}} \) is the minimum allowed processor speed. The processor has a maximum allowed processor speed, \( s_t \leq S^{\text{max}} \), which translates to the constraint \( 1^T u_t \leq S^{\text{max}} \). The processor power dissipation is modeled as \( \alpha s_t^2 \), where \( \alpha > 0 \).

With each queue we associate a quadratic-plus-linear cost \( c_i(x_t)_i + d_i(x_t)_i^2 \), where \( c_i \) and \( d_i \) are positive weights. We can interpret \( c_i \) as relative queue priorities, when the queues are small, and \( c_i/d_i \) as the queue length at which the cost is twice the linear cost alone. When the queue lengths have some probability distribution, the expected value of the queue cost is \( c_i \) times the mean queue length, plus \( d_i \) times the mean square queue length.

The overall stage cost is

\[
\ell(x_t, u_t) = \alpha \max\{S^{\text{min}}, 1^T u_t\}^2 + c^T x_t + d^T x_t^2,
\]

where \( x_t^2 \) is interpreted elementwise. This is a convex function.

For this example the dynamics matrices, constraints, and stage costs are all known. The only uncertainty is the arrivals \( a_t \), which we assume have the form

\[
(a_t)_i = \exp(\lambda_i \sin(2\pi t/M - \theta_i) + (w_t)_i), \quad i = 1, \ldots, n,
\]

where \( M \) is the period, \( \lambda_i, \theta_i \) are known constants, and \( w_t \) is IID gaussian with mean \( \mu \).
and covariance $\Sigma$.

In period $t$, the controller chooses the work rates $u_t$ based on knowledge of the current state $x_t$, as well as the data

$$S_{\text{min}}, S_{\text{max}}, \alpha, a, b, \lambda, \theta, \mu, \Sigma, M.$$  

**Receding horizon policy.** In the RHC policy, our estimates of the arrivals are

$$(\hat{a}_{\tau|t})_i = E(a_t)_i = \exp(\lambda_i \sin(2\pi t/M - \theta_i) + \mu_i + 0.5\Sigma_{ii}), \quad i = 1, \ldots, n, \quad \tau = t, \ldots, t + T.$$  

The RHC optimization problem becomes

$$\begin{aligned}
\text{minimize} & \quad \frac{1}{T+1} \sum_{\tau=t}^{t+T} \alpha \max\{ S_{\text{min}}, 1^T \hat{u}_\tau \}^2 + c^T \hat{x}_\tau + d^T \hat{x}_\tau^2 \\
\text{subject to} & \quad \hat{x}_{\tau+1} = \hat{x}_\tau - \hat{u}_\tau + \hat{a}_{\tau|t}, \quad \tau = t, \ldots, t + T \\
& \quad 0 \leq \hat{u}_\tau \leq \hat{x}_\tau, \quad 1^T \hat{u}_\tau \leq S_{\text{max}}, \quad \tau = t, \ldots, t + T \\
& \quad \hat{x}_t = x_t,
\end{aligned}$$  

(5.2)

where the variables are $\hat{x}_t, \ldots, \hat{x}_{t+T+1}, \hat{u}_t, \ldots, \hat{u}_{t+T}$. This is a convex optimization problem, which can be transformed into a QP.

**Proportional policy.** A simple policy is to set the work rates to be proportional to the amount of work left in each queue. Specifically, we take

$$(u_t)_i = \min\{ (x_t)_i, (x_t)_i/1^T x_t S_{\text{max}} \}.$$  

Here we take the minimum to ensure $u_t \leq x_t$ is satisfied. (The constraint $s_t \leq S_{\text{max}}$ is automatically satisfied by this policy.)

**Related work**

For an overview of power-aware design, see one of the survey papers [IP05, SSH⁺03, DM06]. Closely related work appears in [WAT09], which uses a dynamic speed scaling scheme, motivated by queueing theory, to balance energy consumption and mean response time in a multi-processor system. The problem is formulated as a stochastic dynamic program, with an upper bound used instead of an exact solution. In [MBM⁺09] the authors consider a related problem, where the goal is to maximize processing speed while respecting certain system temperature limits, and solve the problem with a custom convex optimization
Numerical instance

We consider a simple numerical example with \( n = 3 \) queues, and problem data

\[
S^{\text{min}} = 1, \quad S^{\text{max}} = 5, \quad \alpha = 2, \quad c = (1, 1, 1), \quad d = (0.5, 0.5, 0.5),
\]

\[
\lambda = (3, 3.5, 3.2), \quad \theta = (0, 1, 2), \quad \mu = (-2, -2, -2), \quad \Sigma = \text{diag}((0.04, 0.04, 0.04)).
\]

Typical arrivals trajectories are shown in Figure 5.6. For our RHC policy we use horizon \( T = 30 \).

Results. We simulate both policies for 1000 time steps (with the same arrival realization). The RHC policy incurs an average cost of \( J = 71.3 \), while the proportional policy achieves \( J = 95.3 \), which is around 34\% worse. Figure 5.8 shows some sample trajectories. We compare the RHC policy (blue) with the simple proportional policy (red). The plots are (from top to bottom): \((x_t)_1, (x_t)_2, (x_t)_3\), and stage cost \( \ell(x_t, u_t) \).

The CVXGEN code takes at most 0.9 ms to solve at each time step, which is 5000\times faster than with CVX. This means that a dedicated processor could adjust work rates at
CHAPTER 5. RECEDING HORIZON CONTROL

dimensions
n = 3; T = 30
end

parameters
ahat[t] (n), t=0..T
alpha positive
lambda positive
c (n)
D (n,n) psd diagonal
x[0] (n)
Smin positive
Smax positive
end

variables
x[t] (n), t=1..T+1
u[t] (n), t=0..T
end

minimize
alpha*sum[t=0..T](sum(square( pos(max(Smin, sum(u[t]))) ))
+ sum[t=0..T+1](c'*x[t] + sum[t=0..T+1](quad(x[t], D))
+ lambda*sum[t=0..T](sum(square(u[t])))
subject to
x[t+1] = x[t] - u[t] + ahat[t], t=0..T
u[t] >= 0, t=0..T
u[t] <= x[t], t=0..T
sum(u[t]) <= Smax, t=0..T
end

Figure 5.7: CVXGEN code segment for the processor speed control example.

1 kHz, or, alternatively, a processor could use 1% of its processing power and adjust rates at 10 Hz. Further computation performance details are collected in Table 5.2.

5.3.4 Energy storage

Problem statement. We consider an energy storage system that can be charged or discharged from a source with varying energy price. A simple example is a battery connected to a power grid. The goal is to alternate between charging and discharging in order to maximize the average revenue.

Let \( q_t \geq 0 \) denote the charge in the energy store at time period \( t \). The energy store has capacity \( C \), so we must have \( q_t \leq C \). We let \( u^c_t \geq 0 \) denote the amount of energy
Figure 5.8: Sample trajectories for processor speed control example. From top to bottom: \((x_t)_1, (x_t)_2, (x_t)_3,\) and stage cost \(\ell(x_t, u_t),\) for RHC policy (blue) and proportional policy (red).

taken from the source in period \(t\) to charge the energy store, and we let \(u_t^d \geq 0\) denote the amount of energy discharged into the source from our energy store. (We will see that in each time period, at most one of these will be positive; that is, we will never charge and discharge the store simultaneously.) The charging and discharging rates must satisfy

\[
u_t^c \leq C_{\text{max}}, \quad u_t^d \leq D_{\text{max}},
\]

where \(C_{\text{max}}\) and \(D_{\text{max}}\) are the maximum charge/discharge rates.

Charging increases the energy in our store by \(\kappa^c u_t,\) where \(\kappa^c \in (0, 1)\) is the charge efficiency; discharging decreases the energy in our store by \(u_t/\kappa^d,\) where \(\kappa^d \in (0, 1)\) is the discharge efficiency. In each time period the energy store leaks, losing energy proportional to its charge, with leakage coefficient \(\eta \in (0, 1)\). With all these effects incorporated, the system dynamics are

\[
q_{t+1} = \eta q_t + \kappa^c u_t^c - u_t^d/\kappa^d.
\]

In the context of our general framework, the dynamics matrices are \(A = \eta\) and \(B = (\kappa^c, 1/\kappa^d)^T,\) with \(u_t = (u_t^c, u_t^d).\)
The revenue in period $t$ is given by $p_t(u_t^d - u_t^c)$, where $p_t$ is the energy price at time $t$. To discourage excessive charging and discharging, we add a penalty of the form $\gamma(u_t^c + u_t^d)$, where $\gamma \geq 0$ is a parameter. (An alternative interpretation of this term is a transaction cost, with bid-ask spread $\gamma$: We buy energy at price $p_t + \gamma$, and sell energy back at price $p_t - \gamma$.) Our stage cost (i.e., negative revenue, to be minimized) is thus

$$\ell_t(q_t, u_t) = p_t(u_t^c - u_t^d) + \gamma(u_t^c + u_t^d) = (p_t + \gamma)u_t^c - (p_t - \gamma)u_t^d,$$

which can be interpreted as the profit, at time $t$.

We model the energy price as a stationary log-normal process with

$$\mathbb{E} \log p_t = \mu, \quad \mathbb{E}(\log p_t - \mu)(\log p_{t+1} - \mu) = \tau.$$

At time period $t$ the controller has access to the current charge level $q_t$, the data

$$C, \quad C^{\text{max}}, \quad D^{\text{max}}, \quad \kappa^c, \quad \kappa^d, \quad \eta, \quad \gamma,$$

the current and last $N$ prices $p_t, p_{t-1}, \ldots, p_{t-N}$, as well as the mean and autocovariance, $\mu$ and $\tau$. The future prices are not known.

**Receding horizon policy.** To implement the receding horizon policy, we take our estimates of the future prices to be

$$\hat{p}_{t|t} = \exp \mathbb{E}(\log p_{t+1}, \ldots, p_{t+N}), \quad t + 1, \ldots, t + T,$$

which are an affine function of $\log p_t, \ldots, \log p_{t-N}$. (Note that this is not the same as $\mathbb{E}(p_{t+1}, \ldots, p_{t+N})$, which can also be computed and used as estimates of future prices.) Our estimates of the stage costs are

$$\hat{\ell}_t(q_t, \hat{u}_t) = (\hat{p}_{t|t} + \gamma)\hat{u}_t^c - (\hat{p}_{t|t} - \gamma)\hat{u}_t^d.$$
Thus, the RHC optimization problem becomes

\[
\begin{align*}
\text{minimize} & \quad \sum_{\tau=t}^{t+T} \hat{\ell}_t(\hat{q}_\tau, \hat{u}_\tau) \\
\text{subject to} & \quad \hat{q}_{\tau+1} = \eta \hat{q}_\tau + \kappa^c \hat{u}^c_\tau - \hat{u}^d_\tau / \kappa^d, \\
& \quad 0 \leq \hat{u}^c_\tau \leq C^{\text{max}}, \quad 0 \leq \hat{u}^d_\tau \leq D^{\text{max}}, \\
& \quad \tau = t, \ldots, t + T \\
& \quad 0 \leq \hat{q}_\tau \leq C, \quad \tau = t, \ldots, t + T + 1 \\
& \quad \hat{q}_t = q_t,
\end{align*}
\]

with variables \( \hat{q}_t, \ldots, \hat{q}_{t+T+1}, \hat{u}^c_t, \ldots, \hat{u}^c_{t+T}, \hat{u}^d_t, \ldots, \hat{u}^d_{t+T} \). This is an LP.

**Thresholding policy.** We compare the receding horizon policy with a simple thresholding policy, which works as follows:

\[
\begin{align*}
\hat{u}^c_t &= \begin{cases} \\
\min(C^{\text{max}}, C - q) & pt \leq p_{\text{thc}} \\
0 & \text{otherwise} \end{cases}, \\
\hat{u}^d_t &= \begin{cases} \\
\min(D^{\text{max}}, q) & pt \geq p_{\text{thd}} \\
0 & \text{otherwise} \end{cases}.
\end{align*}
\]

In other words, we charge at the maximum rate if the price is below a threshold \( p_{\text{thc}} \), and we discharge at the maximum rate if the price is above a threshold \( p_{\text{thd}} \). If the price is in between we do nothing. We take the minimum to ensure we do not charge above the capacity or discharge below zero.

**Related work**

There is a particularly diverse set of work on optimization in energy storage and production. In [HNH+07], the authors consider a distributed energy system where individual grid-connected households use an MPC-based controller to control ‘micro combined heat and power’ plants. For more on distributed generation and variable pricing, see, respectively, [HNNS06] and [Bra05]. On the generation side, [KS09] considers using MPC and batteries to smooth the power produced by wind turbines. The paper includes a case study with real data.

A different but related application is for hybrid vehicles. Here multiple power sources are available. See [KY06] or [PBK+07], or for a vehicle with multiple different energy storage units see [WBS03].
CHAPTER 5. RECEDING HORIZON CONTROL

Figure 5.9: CVXGEN code segment for the storage example.

Numerical example

We look at a particular numerical instance with $\eta = 0.98$, $\kappa^c = 0.98$, $\kappa^d = 0.98$, $C^{\text{max}} = 10$, $D^{\text{max}} = 10$, $C = 50$, $\gamma = 0.02$, $q_0 = 0$, $\mu = 0$, $r_T = 0.1(0.99^{\tau}\cos(0.1\tau))$. For the receding horizon policy we used a time horizon of $T = 50$ steps, and $N = 100$ previous prices to estimate future prices.

Results. The simulations were carried out for 1000 time steps. Figure 5.10 shows the cumulative profit,

$$r_t = \sum_{\tau=0}^t p_\tau (u_{\tau}^d - u_{\tau}^c) - \gamma (u_{\tau}^d + u_{\tau}^c),$$

for the RHC policy (blue) and the simple thresholding policy (red), over 500 time steps. For the thresholding policy, we adjusted the charge/discharge thresholds via trial and error to achieve good performance. The final thresholds we used are $p_{\text{thc}} = 0.8$, $p_{\text{thd}} = 1.3$. Clearly, the RHC policy outperforms the thresholding policy. The average profit achieved for the
RHC policy is 0.23 per-period, whereas thresholding achieves a profit of 0.029 per-period (averaged over 1000 time steps).

Figure 5.11 shows the actual (black) and predicted (blue) log-price trajectories starting at $t = 150$. The vertical lines show $\exp(\log \hat{p}_t|_{150} \pm \sigma_t)$, where $\sigma_t = (\mathbb{E} (\log p_t - \log \hat{p}_{t|150})^2)^{1/2}$.

The CVXGEN code for this example is shown in Figure 5.9. The generated solver takes up to 360 $\mu$s to solve at each time step, which is 3500× faster than with CVX. Again, these speeds are much faster than is required in practice, since prices would not usually vary on the timescale of microseconds. However, these computation speeds are useful for Monte Carlo simulations and scenario testing. Further computation performance details are collected in Table 5.2.
5.4 CVXGEN performance

To give a rough guide to CVXGEN’s performance, we tested CVXGEN code for each example on three different computers. The given timings should not be taken too seriously, as there are many things that could easily improve performance, often reducing speed by an order of magnitude or more. First, single-precision floats could be used in place of double-precision, because the scale of data is known ahead of time. This would improve performance on a variety of processors. Secondly, the time-horizon selected for the given examples is relatively long. With a suitable choice of final state cost as in [WB09b], this could be reduced further, giving a linearly proportional performance improvement. Finally, we solve the problems to high accuracy (so that control performance does not suffer from suboptimality), which required up to 16 steps. With a small amount of tuning, adequate control performance could easily be achieved using a fixed step limit of (say) 5 steps [WB08]. Thus, all of the numerical results should be taken as preliminary upper bounds on performance, and they will change over time.
Each computer’s properties are summarized in Table 5.1. We used gcc-4.4 on each processor, with the compiler optimization flag -0s. We have not yet conducted tests with a real-time operating system.

In each case, we ensured the computer was idle, then solved optimization problem instances continuously for at least one second. We calculated the maximum time taken for solving any instance, ensuring that each problem was solved to sufficient optimality that further steps would not change control performance. For a rough guide to the speed of a traditional parser-solver (a somewhat unfair comparison), we also tested the performance of CVX on the fastest computer, Computer 3, using Matlab 7.9 and CVX 1.2. For preorder and storage we used Sedumi 1.2; for suspension and proc_speed we used SDPT3 4.0. All results are shown in Table 5.2.

### 5.5 Conclusion

This chapter has shown four examples of code generation in practice. In all cases we implemented an RHC policy, formulating it as a convex optimization problem. We then used
CVXGEN to generate high-speed solvers specific to those problems, and demonstrated typical results. In situations like these, automatic code generation and RHC combine to make a control system designer’s job easy and efficient. Significant performance improvements are possible as well. These examples demonstrate the power of automatic code generation for RHC.
Bibliography


BIBLIOGRAPHY


