Custom Optimization Algorithms for Efficient Hardware Implementation

Juan Luis Jerez

May 2013

Supervised by George A. Constantinides and Eric C. Kerrigan

Submitted in part fulfilment of the requirements for the degree of
Doctor of Philosophy in Electrical and Electronic Engineering of Imperial College London
and the Diploma of Imperial College London
Abstract

The focus is on real-time optimal decision making with application in advanced control systems. These computationally intensive schemes, which involve the repeated solution of (convex) optimization problems within a sampling interval, require more efficient computational methods than currently available for extending their application to highly dynamical systems and setups with resource-constrained embedded computing platforms.

A range of techniques are proposed to exploit synergies between digital hardware, numerical analysis and algorithm design. These techniques build on top of parameterisable hardware code generation tools that generate VHDL code describing custom computing architectures for interior-point methods and a range of first-order constrained optimization methods. Since memory limitations are often important in embedded implementations we develop a custom storage scheme for KKT matrices arising in interior-point methods for control, which reduces memory requirements significantly and prevents I/O bandwidth limitations from affecting the performance in our implementations. To take advantage of the trend towards parallel computing architectures and to exploit the special characteristics of our custom architectures we propose several high-level parallel optimal control schemes that can reduce computation time. A novel optimization formulation was devised for reducing the computational effort in solving certain problems independent of the computing platform used. In order to be able to solve optimization problems in fixed-point arithmetic, which is significantly more resource-efficient than floating-point, tailored linear algebra algorithms were developed for solving the linear systems that form the computational bottleneck in many optimization methods. These methods come with guarantees for reliable operation. We also provide finite-precision error analysis for fixed-point implementations of first-order methods that can be used to minimize the use of resources while meeting accuracy specifications. The suggested techniques are demonstrated on several practical examples, including a hardware-in-the-loop setup for optimization-based control of a large airliner.
Acknowledgements

I feel indebted to both my supervisors for giving a very rewarding PhD experience. To Prof. George A. Constantinides for his clear and progressive thinking, for giving me total freedom to choose my research direction and for allowing me to travel around the world several times. To Dr Eric C. Kerrigan for being a continuous source of interesting ideas, for teaching me to write technically, and for introducing me to many valuable contacts during a good bunch of conference trips we had together.

There are several people outside of Imperial that have had an important impact on this thesis. I would like to thank Prof. Ling Keck-Voon for hosting me at the Control Group at the Nanyang Technical University in Singapore during the wonderful summer of 2010. Prof. Jan M. Maciejowski for hosting me many times at Cambridge University during the last three years, and Dr Edward Hartley for the many valuable discussions and fruitful collaborative work at Cambridge and Imperial. To Dr Paul J. Goulart for hosting me at the Automatic Control Lab at ETH Zürich during the productive spring of 2012, and to Dr Stefan Richter and Mr Alexander Domahidi for sharing my excitement and enthusiasm for this technology.

Within Imperial I would especially like to thank Dr Andrea Suardi, Dr Stefano Longo, Dr Amir Shahzad, Dr David Boland, Dr Ammar Hasan, Mr Theo Drane, and Mr Dinesh Krishnaamoorthy. I am also grateful for the support of the EPSRC (Grants EP/G031576/1 and EP/I012036/1) and the EU FP7 Project EMBOCON, as well as industrial support from Xilinx, the Mathworks, National Instruments and the European Space Agency.

Last but not least, I would like to thank my mother and sisters for always supporting my decisions.
To my grandmother
## Contents

1 Introduction ................................................................. 17  
  1.1 Objectives .............................................................. 17  
  1.2 Overview of thesis .................................................. 18  
  1.3 Statement of originality ............................................ 19  
  1.4 List of publications ................................................ 20  
    1.4.1 Journal papers ................................................ 21  
    1.4.2 Conference papers ............................................. 21  
    1.4.3 Other conference talks ..................................... 22  

2 Real-time Optimization .................................................. 23  
  2.1 Application examples ............................................... 25  
    2.1.1 Model predictive control ..................................... 25  
    2.1.2 Other applications ............................................ 29  
  2.2 Convex optimization algorithms .................................. 30  
    2.2.1 Interior-point methods ....................................... 31  
    2.2.2 Active-set methods .......................................... 34  
    2.2.3 First-order methods ......................................... 35  
  2.3 The need for efficient computing ................................ 39  

3 Computing Technology Spectrum ....................................... 42  
  3.1 Technology trends .................................................. 42  
    3.1.1 The general-purpose microprocessor ....................... 42  
    3.1.2 CMOS technology limitations ................................. 47  
    3.1.3 Sequential and parallel computing ......................... 48  
    3.1.4 General-purpose and custom computing ..................... 49  
  3.2 Alternative platforms .............................................. 51  
    3.2.1 Embedded microcontrollers .................................. 52  
    3.2.2 Digital signal processors ................................... 53  
    3.2.3 Graphics processing units ................................... 54  
    3.2.4 Field-programmable gate arrays ............................ 56  
  3.3 Embedded computing platforms for real-time optimal decision making .................. 58  

4 Optimization Formulations for Control ............................ 59  
  4.1 Model predictive control setup .................................. 60
6.2 Fixed-point aspects of first-order solution methods 115
   6.2.1 The performance gap between fixed-point and floating-point arithmetic 115
   6.2.2 Error sources in fixed-point arithmetic 116
   6.2.3 Notation and assumptions 117
   6.2.4 Overflow errors 118
   6.2.5 Arithmetic round-off errors 119
6.3 Embedded hardware architectures for first-order solution methods 124
   6.3.1 Hardware architecture for the primal fast gradient method 125
   6.3.2 Hardware architecture for ADMM 126
6.4 Case studies 128
   6.4.1 Optimal control of an atomic force microscope 128
   6.4.2 Spring-mass-damper system 131
6.5 Summary and open questions 136

7 Predictive Control Algorithms for Parallel Pipelined Hardware 138
   7.1 The concept of pipelining 139
      7.1.1 Low- and high-level pipelining 139
      7.1.2 Consequences of long pipelines 140
   7.2 Methods for filling the pipeline 141
      7.2.1 Oversampling control 141
      7.2.2 Moving horizon estimation 143
      7.2.3 Distributed optimization via first-order methods 144
      7.2.4 Minimum time model predictive control 144
      7.2.5 Parallel move blocking model predictive control 145
      7.2.6 Parallel multiplexed model predictive control 147
   7.3 Summary and open questions 152

8 Algorithm Modifications for Efficient Linear Algebra Implementations 153
   8.1 The Lanczos algorithm 156
   8.2 Fixed-point analysis 157
      8.2.1 Results with existing tools 157
      8.2.2 A scaling procedure for bounding variables 158
      8.2.3 Validity of the bounds under inexact computations 163
   8.3 Numerical results 165
   8.4 Evaluation in FPGAs 169
      8.4.1 Parameterizable architecture 169
      8.4.2 Design automation tool 171
      8.4.3 Performance evaluation 173
   8.5 Further extensions 177
      8.5.1 Other linear algebra kernels 177
List of Tables

4.1 Comparison of the computational complexity imposed by the different quadratic programming (QP) formulations. ............................ 70
4.2 Comparison of the memory requirements imposed by the different QP formulations. ....................................................... 70

5.1 Performance comparison for several examples. The values shown represent computational time per interior-point iteration. The throughput values assume that there are many independent problems available to be processed simultaneously. ...................................................... 80
5.2 Characteristics of existing FPGA-based QP solver implementations ...... 81
5.3 Total number of floating point units in the circuit in terms of the parameters of the control problem. This is independent of the horizon length $N$. $i$ is the number of parallel instances of Stage 1, which is 1 for most problems. 87
5.4 Cost function .................................................................................................................. 96
5.5 Input constraints .............................................................................................................. 96
5.6 Effects of offline preconditioning .................................................................................. 100
5.7 Values for $c$ in (5.2) for different implementations. ..................................................... 100
5.8 FPGA resource usage. ................................................................................................. 103
5.9 Comparison of FPGA-based MPC regulator performance (with baseline floating point target calculation in software) ...................... 104
5.10 Table of symbols ....................................................................................................... 107

6.1 Resource usage and input-output delay of different fixed-point and floating-point adders in Xilinx FPGAs running at approximately the same clock frequency. 53 and 24 fixed-point bits can potentially give the same accuracy as double and single precision floating-point, respectively. .............. 116
6.2 Resources required for the fast gradient and ADMM computing architectures .............................................................................. 127
6.3 Relative percentage difference between the tracking error for a double precision floating-point controller using $I_{\text{max}} = 400$ and different fixed-point controllers. .................................................................................. 130
6.4 Resource usage and potential performance at 400MHz (Virtex6) and 230MHz (Spartan6) with $I_{\text{max}} = 20$. .................................................. 130
6.5 Percentage difference in average closed-loop cost with respect to a standard double precision implementation. In each table, \( b \) is the number of fraction bits employed and \( I_{\text{max}} \) is the (fixed) number of algorithm iterations. In certain cases, the error increases with the number of iterations due to increasing accumulation of round-off errors. ........................................ 135

6.6 Resource usage and potential performance at 400MHz (Virtex6) and 230MHz (Spartan6) with 15 and 40 solver iterations for FGM and ADMM, respectively. The suggested chips in the bottom two rows of each table are the smallest with enough embedded multipliers to support the resource requirements of each implementation. ........................................ 136

7.1 Computational delay for each implementation when \( I_{IP} = 14 \) and \( I_{\text{MINRES}} = Z \). The gray region represents cases where the computational delay is larger than the sampling interval, hence the implementation is not possible. The smallest sampling interval that the FPGA can handle is 0.281 seconds (3.56Hz) when computing parallel MMPC and 0.344 seconds (2.91Hz) when computing conventional model predictive control (MPC). The relationship \( Ts = \frac{T_h}{N} \) holds. ........................................ 151

7.2 Size of QP problems solved by each implementation. Parallel MMPC solves six of these problems simultaneously. ........................................ 151

8.1 Bounds on \( r_2 \) computed by state-of-the-art bounding tools [23, 149] given \( r_1 \in [-1,1] \) and \( A_{ij} \in [-1,1] \). The tool described in [44] can also use the fact that \( \sum_{j=1}^{N} |A_{ij}| = 1 \). Note that \( r_1 \) has unit norm, hence \( \|r_1\|_{\infty} \leq 1 \), and \( A \) can be trivially scaled such that all coefficients are in the given range. ‘-‘ indicates that the tool failed to prove any competitive bound. Our analysis will show that when all the eigenvalues of \( A \) have magnitude smaller than one, \( \|r_i\|_{\infty} \leq 1 \) holds independent of \( N \) for all iterations \( i \). ........................................ 158

8.2 Delays for arithmetic cores. The delay of the fixed-point divider varies nonlinearly between 21 and 36 cycles from \( k = 18 \) to \( k = 54 \). ........................................ 171

8.3 Resource usage ........................................ 173
List of Figures

2.1 Real-time optimal decision making. .......................................................... 24
2.2 Block diagram describing the general structure of a control system. ........... 26
2.3 The operation of a model predictive controller at two contiguous sampling
time instants. The solid lines represent the output trajectory and optimal
control commands predicted by the controller at a particular time instant.
The shaded lines represent the outdated trajectories and the solid green
lines represent the actual trajectory exhibited by the system and the applied
control commands. The input trajectory assumes a zero-order hold between
sampling instants. ................................................................................ 27
2.4 Convergence behaviour of the gradient (dotted) and fast gradient (solid)
methods when solving two toy problems. ...................................................... 36
2.5 System theory framework for first-order methods. ..................................... 37
2.6 Dual and augmented dual functions for a toy problem. ............................... 38
3.1 Ideal instruction pipeline execution with five instructions (A to E). Time
progresses from left to right and each vertical block represents one clock cy-
cle. F, D, E, M and W stand for instruction fetching, instruction decoding,
execution, memory storage and register writeback, respectively. ................. 44
3.2 Memory hierarchy in a microprocessor system showing on- and off-chip
memories. ................................................................................................. 45
3.3 Intel Pentium processor floorplan with highlighted floating-point unit (FPU).
Diagram taken from [65]. ........................................................................... 46
3.4 Floating-point data format. Single precision has an 8-bit exponent and
a 23-bit mantissa. Double precision has an 11-bit exponent and a 52-bit
mantissa. ................................................................................................. 50
3.5 Components of a floating-point adder. FLO stands for finding leading one.
Mantissa addition occurs only in the 2’s complement adder block. Figure
taken from [137]. ..................................................................................... 51
3.6 Fixed-point data format. An imaginary binary point, which has to be taken
into account by the programmer, lies between the integer and fraction fields. ...
3.7 CUDA-based Tesla architecture in a GPGPU system. The memory ele-
ments are shaded. SP and SM stand for streaming processor and streaming
multiprocessor, respectively. ................................................................. 55
4.1 Accurate count of the number of floating point operations per interior-point iteration for the different QP formulations discussed in this chapter. The size of the control problem is \( n_u = 2, n_x = 6, l = 6 \) and \( r = 3 \). 71

4.2 Oscillating masses example. 72

4.3 Trade-off between closed-loop control cost and computational cost for all different QP formulations. 73

5.1 Hardware architecture for computing dot-products. It consists of an array of \( 2M - 1 \) parallel multipliers followed by an adder reduction tree of depth \( \lceil \log_2(2M - 1) \rceil \). The rest of the operations in a minimum residual (MINRES) iteration use dedicated components. Independent memories are used to hold columns of the stored matrix \( A_k \) (refer to Section 5.4.3 for more details). \( z^{-M} \) denotes a delay of \( M \) cycles. 84

5.2 Proposed two-stage hardware architecture. Solid lines represent data flow and dashed lines represent control signals. Stage 1 performs all computations apart from solving the linear system. The input is the current state measurement \( x \) and the output is the next optimal control move \( u^*_0(x) \). 85

5.3 Floating point unit efficiency of the different blocks in the design and overall circuit efficiency with \( n_u = 3, N = 20 \), and 20 line search iterations. For one and two states, three and two parallel instances of Stage 1 are required to keep the linear solver active, respectively. The linear solver is assumed to run for \( Z \) iterations. 86

5.4 Structure of original and CDS matrices showing variables (black), constants (dark grey), zeros (white) and ones (light grey) for \( n_u = 2, n_x = 4 \), and \( N = 8 \). 89

5.5 Memory requirements for storing the coefficient matrices under different schemes. Problem parameters are \( n_u = 3 \) and \( N = 20 \). 1 does not affect the memory requirements of \( A_k \). The horizontal line represents the memory available in a memory-dense Virtex 6 device [229]. 91

5.6 Online preconditioning architecture. Each memory unit stores one diagonal of the matrix. 91

5.7 Resource utilization on a Virtex 6 SX 475T (\( n_u = 3, N = 20, P \) given by (5.3)). 93

5.8 Performance comparison showing measured performance of the CPU, normalised CPU performance with respect to clock frequency, and FPGA performance when solving one problem and \( 2P \) problems given by (5.3). Problem parameters are \( n_u = 3, N = 20 \), and \( f_c = 250\text{MHz} \). 94

5.9 Energy per interior-point iteration for the CPU, and FPGA implementations when solving one problem and \( 2P \) problems, where \( P \) is given by (5.3). Problem parameters are \( n_u = 3, N = 20 \) and \( f_c = 250\text{MHz} \). 95
5.10 Numerical performance for a closed-loop simulation with \( N = 12 \), using PC-based MINRES-PDIP implementation with no preconditioning (top left), offline preconditioning only (top right), online preconditioning only (bottom left), and both (bottom right). Missing markers for the mean error indicate that at least one control evaluation failed due to numerical errors.

5.11 Hardware-in-the-loop experimental setup. The computed control action by the QP solver is encapsulated into a UDP packet and sent through an Ethernet link to a desktop PC, which decodes the data packet, applies the control action to the plant and returns new state, disturbance and trajectory estimates. \textit{lwip} stands for light-weight TCP/IP stack.

5.12 Closed loop roll, pitch, yaw, altitude and airspeed trajectories (top) and input trajectory with constraints (bottom) from FPGA-in-the-loop testbench.

6.1 Fast gradient compute architecture. Boxes denote storage elements and dotted lines represent \( N n_u \) parallel vector links. The dot-product block \( \hat{v}^T \hat{w} \) and the projection block \( \pi_{\mathbb{R}} \) are depicted in Figures 6.2 and 6.4 in detail. FIFO stands for first-in first-out memory and is used to hold the values of the current iterate for use in the next iteration. In the initial iteration, the multiplexers allow \( \hat{x} \) and \( \hat{\Phi}_n \) through and the result \( \hat{\Phi}_n \hat{x} \) is stored in memory. In the subsequent iterations, the multiplexers allow \( \hat{y}_i \) and \( I - \hat{H}_n \) through and \( \hat{\Phi}_n \hat{x} \) is read from memory.

6.2 Hardware architecture for dot-product block with parallel tree architecture (left), and hardware support for warm-starting (right). Support for warm-starting adds one cycle delay. The last entries of the vector are padded with \( w_N \), which can be constant or depend on previous values. 

6.3 ADMM compute architecture. Boxes denote storage elements and dotted lines represent \( n_A \) parallel vector links. The dot-product block \( \hat{v}^T \hat{w} \) and the projection block \( \pi_{\mathbb{R}} \) are depicted in Figures 6.2 and 6.5 in detail. FIFO stands for first-in first-out memory and is used to hold the values of the current iterate for use in the next iteration. In the initial iteration, the multiplexers allow \( \hat{x} \) and \( M_{12} b(x) \) through and the result \( M_{12} b(x) \) is stored in memory.

6.4 Box projection block. The total delay from \( \hat{t}_i \) to \( \hat{z}_{i+1} \) is \( l_A + 1 \). A delay of \( l_A \) cycles is denoted by \( z^{-l_A} \).

6.5 Truncated cone projection block. The total delay for each component is \( 2l_A + 1 \). \( x \) and \( \delta \) are assumed to arrive and leave in sequence.

6.6 Schematic diagram of the atomic force microscope (AFM) experiment. The signal \( u \) is the vertical displacement of the piezoelectric actuator, \( d \) is the sample height, \( r \) is the desired sample clearance, and \( y \) is the measured cantilever displacement.
6.7 Bode diagram for the AFM model (dashed, blue), and the frequency response data from which it was identified (solid, green).

6.8 Typical cantilever tip deflection (nm, top), control input signal (Volts, middle) and sample height variation (nm, bottom) profiles for the AFM example.

6.9 Convergence of the fast gradient method under different number representations.

6.10 Closed-loop trajectories showing actuator limits, desirable output limits and a time-varying reference. On the top plot 21 samples hit the input constraints. On the bottom plot 11, 28 and 14 samples hit the input, rate and output constraints, respectively. The plots show how MPC allows for optimal operation on the constraints.

6.11 Theoretical error bounds given by (6.15) and practical convergence behavior of the fast gradient method (left) and ADMM (right) under different number representations.

7.1 Different pipelining schemes.

7.2 Different sampling schemes with $T_c$ and $T_s$ denoting the computation times and sampling times, respectively. Figure adapted from [26].

7.3 Predictions for a move blocking scheme where the original horizon length of 9 samples is divided into three hold intervals with $m_0 = 2$, $m_1 = 3$ and $m_2 = 4$. The new effective horizon length is three steps. Figure adapted from [134].

7.4 Standard MPC (top) and multiplexed MPC (bottom) schemes for a two-input system. The angular lines represent when the input command is allowed to change.

7.5 Parallel multiplexed MPC scheme for a two-input system. Two different multiplexed MPC schemes are solved simultaneously. The angular lines represent when the input command is allowed to change.

7.6 Computational time reduction when employing multiplexed MPC on different plants. Results are normalised with respect to the case when $n_u = 1$. The number of parallel channels is given by (5.3), which is: a) 6 for all values of $n_u$; b) 14 for $n_u = 1$, 12 for $n_u \in (2, 5)$, 10 for $n_u \in (6, 13]$ and 8 for $n_u \in (14, 25]$. For parallel multiplexed MPC the time required to implement the switching decision process was ignored, however, this would be negligible compared to the time taken to solve the QP problem.

7.7 Comparison of the closed-loop performance of the controller using conventional MPC (solid) and parallel MMPC (dotted). The horizontal lines represent the physical constraints of the system. The closed-loop continuous-time cost represents $\int_0^\infty x(s)^TQ_s x(s) + u(s)^TR_c u(s)ds$. The horizontal axis represents time in seconds.
8.1 Evolution of the range of values that $\alpha$ takes for different Lanczos problems arising during the solution of an optimization problem from the benchmark set of problems described in Section 8.3. The solid and shaded curves represent the scaled and unscaled algorithms, respectively.  

8.2 Convergence results when solving a linear system using MINRES for benchmark problem sherman1 from [42] with $N = 1000$ and condition number $2.2 \times 10^4$. The solid line represents the single precision floating-point implementation (32 bits including 23 mantissa bits), whereas the dotted lines represent, from top to bottom, fixed-point implementations with $k = 23, 32, 41$ and 50 bits for the fractional part of signals, respectively.  

8.3 Histogram showing the final log relative error $\log_2(\frac{\|Ax-b\|_2}{\|b\|_2})$ at termination for different linear solver implementations. From top to bottom, preconditioned 32-bit fixed-point, double precision floating-point and single precision floating-point implementations, and unpreconditioned single precision floating-point implementation.  

8.4 Accumulated closed-loop cost for different mixed precision interior-point controller implementations. The dotted line represents the unpreconditioned 32-bit fixed-point controller, whereas the crossed and solid lines represent the preconditioned 32-bit fixed-point and double precision floating-point controllers, respectively.  

8.5 Lanczos compute architecture. Dotted lines denote links carrying vectors whereas solid lines denote links carrying scalars. The two thick dotted lines going into the $x^T y$ block denote $N$ parallel vector links. The input to the circuit is $q_1$ going into the multiplexer and the matrix $A$ being written into on-chip RAM. The output is $\alpha_i$ and $\beta_i$.  

8.6 Reduction circuit. Uses $P + l_A - 1$ adders and a serial-to-parallel shift register of length $l_A$.  

8.7 Latency of one Lanczos iteration for several levels of parallelism.  

8.8 Latency tradeoff against FF utilization (from model) on a Virtex 7 XT 1140 [234] for $N = 229$. Double precision ($\eta = 4.05 \times 10^{-14}$) and single precision ($\eta = 3.41 \times 10^{-7}$) are represented by solid lines with crosses and circles, respectively. Fixed-point implementations with $k = 53$ and 29 are represented by the dotted lines with crosses and circles, respectively. These Lanczos implementations, when embedded inside a MINRES solver, match the accuracy requirements of the floating-point implementations.  

8.9 Latency against accuracy requirements tradeoff on a Virtex 7 XT 1140 [234] for $N = 229$. The dotted line, the cross and the circle represent fixed-point and double and single precision floating-point implementations, respectively.
8.10 Sustained computing performance for fixed-point implementations on a Vir-
tex 7 XT 1140 [234] for different accuracy requirements. The solid line
represents the peak performance of a 1 TFLOP/s general-purpose graphics
processing unit (GPGPU). $P$ and $k$ are the degree of parallelisation and
number of fraction bits, respectively.
1 Introduction

This introductory chapter summarises the objectives of this thesis and its main contributions.

1.1 Objectives

Optimal decision making has many practical advantages such as allowing for a systematic design of the decision maker or improving the quality of the decisions taken in the presence of constraints. However, the need to solve an optimization problem at every decision instant, typically via numerical iterative algorithms, imposes a very large computational demand on the device implementing the decision maker. Consequently, so far, optimization-based decision making has only been widely adopted in situations that require making decisions only once during the design phase of a system, or in systems that, while requiring repeated decisions, can afford long computing times or powerful machines.

Implementation of repeated optimal decisions on systems with resource constraints remains challenging. Resource constraints can refer to:

i) time – the time allowed for computing the solution of the optimization problem is strictly limited,

ii) the computational platform – the power consumption, cost, size, memory available, or the computational power are restricted,

or both. In all cases, the key to enabling the power of real-time optimal decision making in increasingly resource-constrained embedded systems is to improve the computational efficiency of the decision maker, i.e. increasing the number of decisions of acceptable quality per unit of time and computational resource.

There are several ways to achieve the desired improvements in computational efficiency. Independently of the method or platform used, one can aim to formulate specific decision making problems as optimization problems such that the number of computations required to solve the resulting optimization problem are minimized. A reduction in the number of computations needed can also be attained by exploring the use of suboptimal decisions and their impact on the behaviour of a system over time. One can also improve the computational efficiency through tailored implementation of optimization algorithms by exploring different computing platforms and exploiting their characteristics. Deriving new optimization algorithms tailored for a specific class of problems or computing platforms is also a promising avenue.
Throughout this thesis we will consider all these methods with a special focus on decision making problems arising in real-time optimal control. We will apply a multidisciplinary approach where the design of the computing hardware and the optimization algorithm is considered jointly. The bulk of research on optimization algorithm acceleration focuses on a reduction of the computation count ignoring details of the embedded platforms on which these algorithm will be deployed. Similarly, in the field of hardware acceleration, much of the application work is concerned with accelerating a given software implementation and replicating its behaviour. Neither of these approaches results in an optimal use of scarce embedded resources. In this thesis, control tools will be used to make hardware decisions and hardware concepts will be used to design new control algorithms. This approach can offer substantial computational efficiency improvements, as we will see in the remainder of this thesis.

1.2 Overview of thesis

Since this thesis lies at the boundary between optimization algorithms and computer architecture design, the first two chapters give the necessary background on each of these topics. Chapter 2 presents the benefits of real-time optimal decision making and discusses several current and future applications. Background on the main optimization algorithms used for control applications is also included. Chapter 3 discusses past and current trends in computing technology, from general-purpose platforms to parallelism and custom computing. The goal is to build an understanding of the hardware features that can lead to computational efficiency or inefficiency for performing certain tasks.

The same optimal control problem can be formulated in various different ways as an optimization problem. Chapter 4 studies the effect of the optimization formulation on the resulting computing effort and memory requirements that can be expected for a solver for such a problem. The chapter starts by reviewing the standard formulations used in the literature and follows by proposing a novel formulation, which, for specific problems, provides a reduction in the number of operations and the memory needed to solve the optimization problem using standard methods.

Tailored implementations of optimization solvers can provide improvements in computational efficiency. The following two chapters explore the tailoring of the computing architecture to different kinds of optimization methods. Chapter 5 proposes a custom single precision floating-point hardware architecture for interior-point solvers for control, designed for high throughput to maximise the computational efficiency. The structure in the optimization problem is used in the design of the datapath and the memory subsystem with a custom storage technique that minimises memory requirements. The numerical behaviour of the reduced floating-point implementations is also studied and a heuristic scaling procedure is proposed to improve the reliability of the solver for a wide range of problems. The proposed designs and techniques are evaluated on a detailed case study for a large airliner, where the performance is verified on a hardware-in-the-loop setup where
the entire control system is implemented on a single chip.

Chapter 6 proposes custom fixed-point hardware architectures for several first-order methods, each of them suitable for a different type of optimal control problem. Numerical investigations play a very important role for improving the computational efficiency of the resulting implementations. A fixed-point round-off error analysis using systems theory predicts the stable accumulation of errors, while the same analysis can be used for choosing the number of bits and resources needed to achieve a certain accuracy at the solution. A scaling procedure is also suggested for improving the convergence speed of the algorithms. The proposed designs are evaluated on several case studies, including the optimal control of an atomic force microscope at megaHertz sampling rates.

The high throughput design emphasis in the interior-point architectures described in Chapter 5 resulted in several interesting characteristics of the architectures, the main one being the capability to solve several independent optimization problems in the same time and using the same amount of resources as when solving a single problem. Chapter 7 is concerned with exploiting this observation to improve the computational efficiency. We discuss how several non-conventional control schemes in the recent literature can be applied to make use of the slack computational power in the custom architectures.

The main computational bottleneck in interior-point methods, and the task that consumes most computational resources in the architectures described in Chapter 5, is the repeated solution of systems of linear equations. Chapter 8 proposes a scaling procedure to modify a set of linear equations such that they can be solved using more efficient fixed-point arithmetic while provably avoiding overflow errors. The proofs presented in this chapter are beyond the capabilities of current state-of-the-art arithmetic variable bounding tools and are shown to also hold under inexact computations. Numerical studies suggest that substantial improvements in computational efficiency can be expected by including the proposed procedure in the interior-point hardware architectures.

Chapter 9 summarises the main results in this thesis.

1.3 Statement of originality

We now give a summary of the main contribution in each of the chapters in this thesis. A more detailed discussion of contributions is given in the introductory section of each chapter. The main contributions are:

- a novel way to formulate optimization problems coming from a linear time-invariant predictive control problem. The approach uses a specific input transformation such that a compact and sparse optimization problem is obtained when eliminating the equality constraints. The resulting problem can be solved with a cost per interior-point iteration which is linear in the horizon length, when this is bigger than the controllability index of the plant. The computational complexity of existing condensed approaches grow cubically with the horizon length, whereas existing non-condensed
and sparse approaches also grow linearly, but with a greater proportionality constant than with the method derived in Chapter 4.

- a novel parameterisable hardware architecture for interior-point solvers customised for predictive control problems featuring parallelisation and pipelining techniques. It is shown that by considering that the quadratic programs (QPs) come from a control formulation, it is possible to make heavy use of the sparsity in the problem to save computations and reduce memory requirements by 75%. The design is demonstrated with an FPGA-in-the-loop testbench controlling a nonlinear simulation of a large airliner. This study considers a much larger plant than any previous FPGA-based predictive control implementation to date, yet the implementation comfortably fits into a mid-range FPGA, and the controller compares favourably in terms of solution quality and latency to state-of-the-art QP solvers running on a conventional desktop processor.

- the first hardware architectures for first-order solvers for predictive control problems, parameterisable in the size of the problem, the number representation, the type of constraints, and the degree of parallelisation. We provide analysis ensuring the reliable operation of the resulting controller under reduced precision fixed-point arithmetic. The results are demonstrated on a model of an industrial atomic force microscope where we show that, on a low-end FPGA, satisfactory control performance at a sample rate beyond 1 MHz is achievable.

- a novel parallel predictive control algorithm that makes use of the special characteristics of pipelined interior-point hardware architectures, which can reduce the resource usage and improve the closed-loop performance further despite implementing sub-optimal solutions.

- a novel procedure for scaling linear equations to prevent overflow errors when solving the modified problem using iterative methods in fixed-point arithmetic. For this class of nonlinear recursive algorithms the bounding problem for avoiding overflow errors cannot be automated by current tools. It is shown that the numerical behaviour of fixed-point implementations of the modified problem can be chosen to be at least as good as a double precision floating-point implementation, if necessary. The approach is evaluated on FPGA platforms, highlighting orders of magnitude potential performance and efficiency improvements by moving form floating-point to fixed-point computation.

1.4 List of publications

Most of the material discussed in Chapters 4, 5, 6, 7 and 8 originates from the following publications:
1.4.1 Journal papers


1.4.2 Conference papers


1.4.3 Other conference talks

J. L. Jerez, “Embedded Optimization in Fixed-Point Arithmetic”, in Int. Conf. on Continuous Optimization, Lisbon, Portugal, Jul 2013.


2 Real-time Optimization

A general continuous optimization problem has the form

\[
\begin{align*}
\text{minimize} & \quad f(z) \\
\text{subject to} & \quad c_i(z) = 0, \quad i \in \mathcal{E}, \\
& \quad c_i(z) \leq 0, \quad i \in \mathcal{I}.
\end{align*}
\]

(2.1a) (2.1b) (2.1c)

Here, \( z := (z_1, z_2, \cdots, z_n) \in \mathbb{R}^n \) are the decision variables. \( \mathcal{E} \) and \( \mathcal{I} \) are finite sets containing the indices of the equality and inequality constraints, satisfying

\[ \mathcal{E} \cap \mathcal{I} = \emptyset, \]

with the number of equality and inequality constraints denoted by the cardinality of the sets \(|\mathcal{E}|\) and \(|\mathcal{I}|\), respectively. Functions \( c_i : \mathbb{R}^n \to \mathbb{R} \) define the feasible region and \( f : \mathbb{R}^n \to \mathbb{R} \) defines the performance criterion to be optimized, which often involves a weighted combination (trade-off) of several conflicting objectives, e.g.

\[ f(z) := f_0(z_1, z_2) + 0.5 f_1(z_2, z_4) + 0.75 f_2(z_1, z_3). \]

A vector \( z^* \) is a global optimal decision vector if for all vectors \( z \) satisfying (2.1b)-(2.1c), we have \( f(z^*) \leq f(z) \).

The search for optimal decisions is ubiquitous in all areas of engineering, science, business and economics. For instance, every engineering design problem can be expressed as an optimization problem like (2.1), as it requires the choice of design parameters under economical or physical constraints that optimize some selection criterion. For example, in the design of base stations for cellular networks one can choose the number of antenna elements and their topology to minimize the cost of the installation while guaranteeing coverage across the entire cell and adhering to radiation regulations [126]. Conceptually similar, least-squares fitting in statistical data analysis selects model parameters to minimize the error with respect to some observations while satisfying constraints on the model such as previously obtained information. In portfolio management, a common problem is to find the best way to invest a fixed amount of capital in different financial assets to trade off expected return and risk. In this case, a trivial constraint is a requirement on the investments to be nonnegative. In all of these examples, the ability to find and apply optimal decisions has great value.

Later on in this thesis we will use ideas from digital circuit design to devise more efficient
methods for solving computationally intensive problems like (2.1). Interestingly, optimal decision making has also had a large impact on integrated circuit design as an application. For example, optimization can be used to design the number of bits used to represent different signals in a signal processing system in order to minimize the resources required while satisfying signal-to-noise constraints at the system’s output [37]. At a lower level, individual transistor and wire sizes can be chosen to minimize the power consumption or total silicon area of a chip while meeting signal delay and timing requirements and adhering to the limits of the target manufacturing process [206,217]. Optimization-based techniques have also been used to build accurate performance and power consumption models for digital designs from a reduced number of observations in situations when obtaining data points is very expensive or time consuming [163].

What all the mentioned applications have in common is that they are only solved once or a few times with essentially no constraints on the computational time or resources and the results are in most cases implemented by humans. For this kind of application belonging to the field of classical operations research, there exist mature software packages such as Gurobi [84], IBM’s CPLEX [98], MOSEK [155], or IPOPT [221] that are designed to efficiently solve large-scale optimization problems mostly on x86-based machines with a large amount of memory and using double-precision floating-point arithmetic, e.g. on powerful desktop PCs or servers. In this domain, the main challenge is to formulate the decision making problems in such a way that they can be solved by existing powerful solvers.

Real-time optimal decision making

There exist other applications, in which optimization is used to make automatic decisions with no human interaction in a setup such as the one illustrated in Figure 2.1. Every time new information is available from some sensors (physical or virtual), an optimization problem is solved online and the decision is sent to be applied by some actuators (again, physical or virtual) to optimize the behaviour of a process. Because in this setting there is typically no human feedback, the methods used to solve these problems have to be extremely reliable and predictable, especially for safety-critical applications. Fortunately, since the sequence of problems being solved only varies slightly from instance to instance and there exists the possibility for a detailed analysis prior to deployment, one can devise highly customised methods for solving these optimization problems that can efficiently

![Figure 2.1: Real-time optimal decision making.](image-url)
exploit problem-specific characteristics such as size, structure and problem type. Many of
the techniques described in this thesis exploit this observation.

A further common characteristic of these problems is that they are, in general, signifi-
cantly smaller than those in operations research but they have to be solved under resource
limitations such as computing time, memory storage, cost, or power consumption, typically
on non-desktop or embedded platforms (see Chapter 3 for a discussion on the different
available embedded technologies). In this domain, the main challenge is still to devise
efficient methods for solving problems that, if they were only solved once – offline – might
appear trivial. This is the focus of this thesis.

2.1 Application examples

In this section we discuss several applications in the increasingly important domain of
embedded optimal decision making. The main application on which this thesis focuses,
advanced optimization-based control systems, is described first in detail. We then briefly
discuss several other applications on which the findings in this thesis could have a similar
impact.

2.1.1 Model predictive control

A computer control system gives commands to some actuators to control the behaviour
and maintain the stable operation of a physical or virtual system, known as the plant,
over time. Because the plant operates in an uncertain environment, the control system
has to respond to uncertainty with control actions computed online at regular intervals,
denoted by the sampling time $T_s$. Because the control actions depend on measurements or
estimates of the uncertainty, this process is known as feedback control. Figure 2.2 describes
the structure of a control system and shows the possible sources of uncertainty: actuator
and sensor noise, plant-model mismatch, external disturbances acting on the plant and
estimation errors. Note that not all control systems will necessarily have all the blocks
shown in Figure 2.2.

In model predictive control the input commands given by the controller are computed
by solving a problem like (2.1). The equality constraints (2.1b) describe the model of the
plant, which is used to predict into the future. As a result, the success of a model predictive
control strategy, like any model-based control strategy, largely relies on the availability of
good models for control. These models can be obtained through first principles or through
system identification. A very important factor that has a large effect on the difficulty of
solving (2.1) is whether the model is linear or nonlinear, which results in convex or non-
convex constraints, respectively.

The inequality constraints (2.1c) describe the physical constraints on the plant. For
example, the amount of fluid that can flow through a valve providing an input for a
chemical process is limited by some quantity determined by the physical construction of
the valve and cannot be exceeded. In some other cases, the constraints describe virtual
limitations imposed by the plant operator or designer that should not be exceeded for a safe operation of the plant. The presence of inequality constraints prevents one from computing analytical solutions to (2.1) and forces one to use numerical methods such as the ones described in Section 2.2.

The cost function (2.1a) typically penalizes deviations of the predicted trajectory from the setpoint, as well as the amount of input action required to achieve a given tracking performance. Deviations from the setpoints are generally penalized with quadratic terms whereas penalties on the input commands can vary from quadratic terms to 1- and $\infty$-norm terms. Note that in all these cases, the problem (2.1b) can be formulated as a quadratic program. The cost function establishes a trade-off between conflicting objectives. As an example, a model predictive controller on an aeroplane could have the objective of steering the aircraft along a given trajectory while minimizing fuel consumption and stress on the wings. A formal mathematical description of the functions involved in (2.1) will be given in Chapter 4.

The operation of a model predictive controller is illustrated in Figure 2.3. At time $t$ a measurement of the system’s output is taken and, if necessary, the state and disturbances are estimated and the setpoint is recalculated. The optimization problem (2.1) is then solved to compute open-loop optimal output and input trajectories for the future, denoted by the solid black lines in Figure 2.3. Since there is a computational delay associated with solving the optimization problem, the first input command is applied at the next sampling instant $t + T_s$. At that time, another measurement is taken, which, due to various uncertainties might differ from what was predicted at the previous sampling time, hence the whole process has to be repeated at every sampling instant to provide closed-loop stability and robustness through feedback.

Optimization-based model predictive control offers several key advantages over conventional control strategies. Firstly, it allows for systematic handling of constraints. Com-
Figure 2.3: The operation of a model predictive controller at two contiguous sampling time instants. The solid lines represent the output trajectory and optimal control commands predicted by the controller at a particular time instant. The shaded lines represent the outdated trajectories and the solid green lines represent the actual trajectory exhibited by the system and the applied control commands. The input trajectory assumes a zero-order hold between sampling instants.

Compared to control techniques that employ application-specific heuristics, which involve a lot of hand tuning, to make sure the system’s limits are not exceeded, MPC’s systematic handling of constraints can significantly reduce the development time for new applications [122]. As a consequence, the validation of the controller’s behaviour can be substantially simpler. A further advantage is the possibility of specifying meaningful control objectives directly when those objectives can be formulated in a mathematically favourable way. Furthermore, the controller formulation allows for simple adaptability of the controller to changes in the plant or controller objectives. In contrast to conventional controllers, which would need to be redesigned if the control problem changes, an MPC controller would only require changing the functions in (2.1).

The second key advantage is the potential improvement in performance from an optimal handling of constraints. It is well known that if the optimal solution to an unconstrained convex optimization problem is infeasible with respect to the constraints, then the solution to the corresponding constrained problem will lie on at least one of the constraints. Unlike conventional control methods, which avoid the system limits by operating away from the constraints, model predictive control allows for optimal operation at the system limits, potentially delivering extra performance gains. The performance improvement has different consequences depending on the particular application, as we will see in the example sections that follow.

Figure 2.3 also highlights the main limitation for implementing model predictive controllers - the sampling frequency can only be set as fast as the time taken to compute the solution to the optimization problem (2.1). Since solving these problems requires several orders of magnitude more computations than with conventional control techniques, MPC
has so far only enjoyed widespread adoption in systems with both very slow dynamics (with sampling intervals in the order of seconds, minutes, or longer) and the possibility of employing powerful computing hardware. Examples of such systems arise in the chemical process industries [139, 181]. In these industries, the use of optimization-based control has changed industrial control practice over the last three decades and accounts for multi-million dollar yearly savings.

**Next generation MPC applications**

Intuitively, the state of a plant with fast dynamics will respond faster to a disturbance, hence a prompter reaction is needed in order to control the system effectively. The challenge now is to extend the applicability of MPC to applications with fast dynamics that can benefit from operating at the system limits, such as those encountered in the aerospace [111, 158, 188], robotics [219], ship [69], electrical power [192], or automotive [62, 154] industries. Equally challenging is the task of extending the use of MPC to applications that, even if the sampling requirements are not in the milli- to microsecond range, currently implement simple PID control loops due to the limitations of the available computing hardware.

We now list several important applications areas where real-time optimization-based control has been recently shown, in research labs, to have the potential to make a significant difference compared to existing industrial solutions if the associated optimization problems could be solved fast enough with the available computing resources.

- Optimal control of an industrial electric drive for medium-voltage AC motors could reduce harmonic distortions in phase currents by 20% [73] leading to enhanced energy efficiency and reduced grid distortion, while enlarging the application scope of existing drives.

- Optimal idle speed control of a diesel combustion engine could lead to a 5.5% improvement in fuel economy [48], lower emissions and enhanced drivability, while avoiding engine stalls.

- Real-time optimization-based constrained trajectory generation for advanced driver assistance systems could improve the smoothness of the trajectory of the vehicle on average (maximum) by 10% (30%) [40].

- Optimal platform motion control for professional driving simulators could generate more realistic driving feelings than with currently available techniques [143].

- Optimal control of aeroplanes with many more degrees of freedom, such as the number of flaps, ailerons or the use of smart airfoils [59], could minimize fuel consumption and improve passenger comfort.

- Optimal trajectory control of airborne power generating kites [83, 100] could minimize energy losses under changing wind conditions.
Optimal control for spacecraft rendezvous maneuvers could minimize fuel consumption while avoiding obstacles and debris in the spacecraft’s path and handling other constraints [47, 87]. Note that computing hardware in spacecraft applications has extreme power consumption limitations.

2.1.2 Other applications

Besides feedback control, there are many emerging real-time optimal decision making applications in various other fields. In this section we briefly discuss several of these applications.

In signal processing, an optimization-based technique known as compressed sensing [50] has had a major impact in recent years. In summary, the technique consists of adding an $l_1$ regularizing term to objective (2.1a) in the form

$$f(z) + w\|z\|_1,$$

which has the effect of promoting sparsity in the solution vector since $\|z\|_1$ can be interpreted as a convex relaxation of the cardinality function. The sparsity in the solution can be tuned through weight vector $w$. Since the problem is convex there exist efficient algorithms [112] based on the ones discussed in the following Section 2.2 to solve this problem. In practical terms, these techniques allow one to reconstruct many coefficients from a small number of observations, a situation in which classical least squares fails to give useful information. Example applications include real-time magnetic resonance imaging (MRI) where compressed sensing can enhance brain and dynamic heart imaging at reduced scanning rates of only 20 ms while maintaining good spatial resolution [213], or for simple inexpensive single-pixel cameras where real-time optimization could allow fast reconstruction of low memory images and videos [55].

Real-time optimization techniques have also been proposed for audio signal processing where optimal perception-based clipping of audio signals could improve the perceptual audio quality by 30% compared to existing heuristic clipping techniques [45]. In the communications domain several optimization-based techniques have been proposed for wireless communication networks. For example, for real-time resource allocation in cognitive radio networks that have to accommodate different groups of users, the use of optimization-based techniques can increase overall network throughput by 20% while guaranteeing the quality of service for premium users [243]. Multi-antenna optimization-based beamforming could also be used to improve the transmit and receive data rates in future generation wireless networks [71].

Beyond signal processing applications, real-time optimization could have an impact in future applications such as the smart recharging of electric vehicles, where the vehicle could decide at which intensity to charge its battery to minimize energy costs while ensuring the required final state of charge using a regularly updated forecast of energy costs, or in next generation low cost DNA sequencing devices with optimization-based genome
2.2 Convex optimization algorithms

In this section we briefly describe different numerical methods for solving problems like (2.1) that will be further discussed throughout the rest of this thesis.

In this thesis, we focus on convex optimization problems. This class of problems have convex objective and constraint functions and have the important property that any local solution is also a global solution [25]. We will focus on a subclass of convex optimization problems known as convex quadratic programs in the form

\[
\begin{align*}
\min_z & \quad \frac{1}{2} z^T H z + h^T z \\
\text{subject to} & \quad F z = f, \\
& \quad G z \leq g,
\end{align*}
\]  

(2.2a)

(2.2b)

(2.2c)

where matrix \( H \) is positive semidefinite. Note that linear programming is a special case with \( H = 0 \).

The Lagrangian associated with problem (2.1) and its dual function are defined as

\[
L(z, \lambda, \nu) := f(z) + \sum_{i \in \mathcal{E}} \nu_i c_i(z) + \sum_{i \in \mathcal{I}} \lambda_i c_i(z) \quad \text{and} \quad \quad g(\lambda, \nu) = \inf_z L(z, \lambda, \nu).
\]  

(2.3)

(2.4)

where \( \nu_i \) and \( \lambda_i \) are Lagrange multipliers giving a weight to their associated constraints. The dual problem is defined as

\[
\begin{align*}
\max_{\lambda, \nu} & \quad g(\lambda, \nu) \\
\text{subject to} & \quad \lambda \geq 0,
\end{align*}
\]  

(2.5a)

(2.5b)

and for problem (2.2) it is given by

\[
\begin{align*}
\max_{\lambda, \nu} & \quad \frac{1}{2} z^T H z + h^T z + \nu^T (F z - f) + \lambda^T (G z - g) \\
\text{subject to} & \quad H z + h + F^T \nu + G^T \lambda = 0, \\
& \quad \lambda \geq 0,
\end{align*}
\]  

(2.6a)

(2.6b)

(2.6c)

where one can eliminate the primal variables \( z \) using (2.6b). Since problem (2.2) is convex, Slater’s constraint qualification condition holds [25] and we have \( f(z^*) = g(\lambda^*, \nu^*) \). Assuming that the objective and constraint functions are differentiable, which is the case in problem (2.2), the optimal primal \( (z^*) \) and dual \( (\lambda^*, \nu^*) \) variables have to satisfy the
which are known as the first-order optimality conditions or Karush-Kuhn-Tucker (KKT) conditions. For convex problems these conditions are necessary and sufficient. Note that (2.7b) and (2.7c) correspond to the feasibility conditions for the primal problem (2.2) and (2.7a) and (2.7d) correspond to the feasibility conditions with respect to the dual problem (2.6). Condition (2.7e) is known as complementary slackness and states that the Lagrange multipliers $\lambda^*_i$ are zero unless the associated constraints are active at the solution.

We now discuss several convex optimization algorithms that can be interpreted as methods that iteratively compute solutions to (2.7).

### 2.2.1 Interior-point methods

Interior-point methods generate iterates that lie strictly inside the region described by the inequality constraints. Feasible interior-point methods start with a primal-dual feasible initial point and maintain feasibility throughout, whereas infeasible interior-point methods are only guaranteed to be feasible at the solution. We discuss two types, primal-dual [228] and logarithmic-barrier [25], which are conceptually different but very similar in practical terms.

**Primal-dual methods**

We can introduce slack variables $s$ to turn the inequality constraint (2.2c) into an equality constraint and rewrite the KKT optimality conditions as

\[
F(z, \nu, \lambda, s) := \begin{bmatrix}
Hz + h + F^T \nu + G^T \lambda \\
Fz - f \\
Gz - g + s \\
\Lambda S \mathbf{1}
\end{bmatrix} = 0,
\]

\[
\lambda, s \geq 0.
\]

where $\Lambda$ and $S$ are diagonal matrices containing the elements of $\lambda$ and $s$, respectively, and $\mathbf{1}$ is an appropriately sized vector whose components are all one. Primal-dual interior-point methods use Newton-like methods to solve the nonlinear equations (2.8) and use a line
search to adjust the step length such that (2.9) remains satisfied. At each iteration $k$ the search direction is computed by solving a linear system of the form

$$
\begin{bmatrix}
H & F^T & G^T & 0 \\
F & 0 & 0 & 0 \\
G & 0 & 0 & I \\
0 & 0 & S_k & \Lambda_k
\end{bmatrix}
\begin{bmatrix}
\Delta z_k \\
\Delta \nu_k \\
\Delta \lambda_k \\
\Delta s_k
\end{bmatrix} =
\begin{bmatrix}
H z_k + h + F^T \nu_k + G^T \lambda_k \\
F z_k - f \\
G z_k - g + s_k \\
\Lambda_k S_k 1 - \tau_k 1
\end{bmatrix} :=
\begin{bmatrix}
r_z \\
r_\nu \\
r_\lambda \\
r_s
\end{bmatrix},
$$

(2.10)

where $\tau_k$ is the barrier parameter, which governs the progress of the interior-point method and converges to zero. The barrier parameter is typically set to $\sigma_k \mu_k$ where

$$
\mu_k := \frac{\lambda^T s_k}{|Z|}
$$

(2.11)

is a measure of suboptimality known as the duality gap.

Note that solving (2.10) does not give a pure Newton search direction due to the presence of $\tau_k$. The parameter $\sigma_k$, known as the centrality parameter, is a number between zero and one that modifies the last equation to push the iterates towards the centre of the feasible region and prevent small steps being taken when the iterates are close to the boundaries of the feasible region. The weight of the centrality parameter decreases as the iterates approach the solution (as the duality gap decreases). Several choices for updating $\sigma_k$ give rise to different primal-dual interior-point methods. A popular variant known as Mehrotra’s predictor-corrector method [148] is used in most interior-point quadratic programming software packages [49, 72, 146]. For more information on the role of the centrality parameter see [228].

The main computational task in interior-point methods is solving the linear systems (2.10). An important point to note is that only the bottom block row of the matrix is a function of the current iterate, a fact which can be exploited when solving the linear system. The so called unreduced system of (2.10) has a non-symmetric indefinite KKT matrix, which we denote with $K_4$. However, the matrix can be easily symmetrized using the following diagonal similarity transformation [66]

$$
D = \begin{bmatrix}
I & 0 & 0 & 0 \\
0 & I & 0 & 0 \\
0 & 0 & I & 0 \\
0 & 0 & 0 & S^\frac{1}{2}_k
\end{bmatrix}, \quad \hat{K}_4 := D^{-1} K_3 D = \begin{bmatrix}
H & F^T & G^T & 0 \\
F & 0 & 0 & 0 \\
G & 0 & 0 & S^\frac{1}{2}_k \\
0 & 0 & S^\frac{1}{2}_k & \Lambda_k
\end{bmatrix}.
$$

(2.12)

One can also eliminate $\Delta s$ from (2.10) to obtain the, also symmetric, augmented system
given by
\[
\begin{bmatrix}
H & F^T & G^T \\
F & 0 & 0 \\
G & 0 & -W_k
\end{bmatrix}
\begin{bmatrix}
\Delta z_k \\
\Delta \nu_k \\
\Delta \lambda_k
\end{bmatrix}
= -
\begin{bmatrix}
r_k^z \\
r_k^\nu \\
r_k^\lambda - \Lambda^{-1}r_k^z
\end{bmatrix},
\] (2.13)

where \( W := \Lambda^{-1}S \) and
\[
\Delta s_k = -\Lambda^{-1}r_k^s - W_k \Delta \lambda_k.
\] (2.14)

Since the matrix in (2.13) is still indefinite and the block structure lends itself well to further reduction, it is common practice to eliminate \( \Delta \lambda \) to obtain the saddle-point system given by
\[
\begin{bmatrix}
H + G^T W_k^{-1}G & F^T \\
F & 0
\end{bmatrix}
\begin{bmatrix}
\Delta z_k \\
\Delta \nu_k
\end{bmatrix}
= -
\begin{bmatrix}
r_k^z + G^T \left( -S^{-1}r_k^s + W_k^{-1}r_k^\lambda \right) \\
F z_k - f
\end{bmatrix},
\] (2.15)

where
\[
\Delta \lambda_k = -S^{-1}r_k^s + W_k^{-1}r_k^\lambda + W_k^{-1}G \Delta z_k.
\] (2.16)

This formulation is used in many software packages [29, 72, 146]. Other solvers [49] perform an extra reduction step to obtain a positive semidefinite system known as the normal equations
\[
F \left( H + G^T W_k^{-1}G \right)^{-1} F^T = F \left( H + G^T W_k^{-1}G \right)^{-1} \left( -r_k^z + G^T \left( -S^{-1}r_k^s + W_k^{-1}r_k^\lambda \right) \right) + r_k^\nu
\]
with
\[
\Delta z_k = \left( H + G^T W_k^{-1}G \right)^{-1} \left( -r_k^z + G^T \left( -S^{-1}r_k^s + W_k^{-1}r_k^\lambda \right) - F^T \Delta \nu_k \right).
\] (2.17)

Employing this formulation allows one to use more robust linear system solvers, however, it requires computing \( (H + G^T W_k^{-1}G)^{-1} \) in order to form the linear system, which is potentially problematic when \( (H + G^T W_k^{-1}G) \) is ill-conditioned.

**Barrier methods**

The main idea in a logarithmic barrier interior-point method is to remove the inequality constraints by adding penalty functions in the cost function that are only defined in the interior of the feasible region. For instance, instead of solving problem (2.2) we solve
\[
\min_z \frac{1}{2} z^T H z + h^T z - \tau 1^T \ln(Gz - g)
\] (2.18a)

subject to \( F z = f \) .
\] (2.18b)
where $\tau$ is again the barrier parameter and $\ln()$ is the natural logarithm applied component-wise. Of course, the solution to problem (2.18) is only optimal with respect to (2.2) when $\tau$ goes to zero. However, problem (2.18) is harder to solve for smaller values of $\tau$, so the algorithm solves a sequence of problems like (2.18) with decreasing $\tau$, each initialised with the previous solution.

In this case, after eliminating $\Delta \lambda$ the Newton search direction is given by

$$
\begin{bmatrix}
H - \tau G^T Q^{-2} G & F^T \\
F & 0
\end{bmatrix}
\begin{bmatrix}
\Delta z_k \\
\Delta \nu_k
\end{bmatrix}
= -
\begin{bmatrix}
H z_k + h + F^T \nu_k - \tau G^T Q_k^{-1} 1 \\
F z_k - f
\end{bmatrix},
$$

where $Q := \text{diag}(Gz - g)$. Observe that (2.19) has the same structure as (2.15). If we use slack variables in the formulation (2.18), the KKT conditions become

$$
F(z, \nu, \lambda, s) := 
\begin{bmatrix}
Hz + h + F^T \nu + G^T \lambda \\
Fz - f \\
Gz - g + s \\
\Lambda S1 - 1\tau
\end{bmatrix}
= 0,
$$

$$
\lambda, s \geq 0,
$$

which is the same as the modified KKT conditions used in primal-dual methods, highlighting the similarity in the role of the barrier parameter and centrality parameters in the two types of interior-point methods.

### 2.2.2 Active-set methods

Active-set methods [166] will not be discussed in the remainder of this thesis, however, we include a brief discussion here for completeness.

These methods find the solution to the KKT conditions by solving several equality constrained problems using Newton’s method. The equality constrained problems are generated by estimating the active set

$$
\mathcal{A}(z^*) := \{ i \in \mathcal{I} : c_i(z^*) = 0 \},
$$

i.e. the constraints that are active at the solution, enforcing them as equalities, and ignoring the inactive ones. Once the active set is known, the solution can be obtained by solving a single Newton problem, so the major difficulty is in determining the active-set.

The running estimate of the active set, known as the working set, is updated when:

- the full Newton step cannot be taken because some constraints become violated, then the first constraints to be violated are added to the working set,
- the current iterate minimizes the cost function over the working set but some Lagrange multipliers are negative, then the associated constraints are removed from the working set.
The method terminates when the current iterate minimizes the cost function over the working set and all Lagrange multipliers associated with constraints in the working set are non-negative.

Active-set methods tend to be the method of choice for offline solution of small to medium scale quadratic programs since they often require a small number of iterations, especially if a good estimate of the active-set is available to start with. However, their theoretical properties are not ideal since, in the worst case, active-set methods have a computational complexity that grows exponentially in the number of constraints. This makes their use problematic in applications that need high reliability and predictability. For software packages based on active-set methods, refer to [61].

2.2.3 First-order methods

In this section we discuss several methods that, unlike interior-point or active-set methods, only use first-order gradient information to solve constrained optimization problems. While interior-point methods typically require few expensive iterations that involve solving linear equations, first order methods require many more iterations that involve, in certain important cases, only simple operations. Although these methods only exhibit linear convergence, compared to quadratic convergence for Newton-based methods, it is possible to derive practical bounds for determining the number of iterations required to achieve a certain suboptimality gap, which is important for certifying the behaviour of the solver. However, unlike with Newton-based methods, the convergence is greatly affected by the conditioning of the problem, which restricts their use in practice.

A further limitation is the requirement on the convex set defined by the inequality constraints, denoted here by $\mathcal{K}$, to be simple. By simple we mean that the Euclidean projection defined as

$$
\pi_{\mathcal{K}}(z_k) := \arg \min_{z \in \mathcal{K}} \|z - z_k\|_2
$$

is easy to compute. Examples of such sets include the 1- and $\infty$-norm boxes, cones and 2-norm balls. For general polyhedral constraints solving (2.23) is as complex as solving a quadratic program. Since this operation is required at every iteration, it is only practical to use these methods for problems with simple sets.

Primal accelerated gradient methods

We first discuss primal first-order methods for solving inequality constrained problems of the type

$$
\min_{z \in \mathcal{K}} f(z),
$$

(2.24)
where $f(z)$ is strongly convex on set $\mathbb{K}$, i.e. there exist a constant $\mu > 0$ such that

$$f(z) \geq f(y) + \nabla f(y)^T (z - y) + \frac{\mu}{2} \|z - y\|^2,$$

and its gradient is Lipschitz continuous with Lipschitz constant $L$. The simplest method is a variation of gradient descent for constrained optimization known as the projected gradient method [15] where the solution is updated according to

$$z_{k+1} := \pi_{\mathbb{K}} \left( z_k - \frac{1}{L} \nabla f(z_k) \right),$$

(2.25)

As with gradient descent, the projected gradient method often converges very slowly when the problem is not well-conditioned. There is a variation due to Nesterov, known as the fast or accelerated gradient method [164], which loses the monotonicity property, i.e. $f(z_{k+1}) \leq f(z_k)$ does not hold for all $k$, but significantly reduces the dependence on the conditioning of the problem, as illustrated in Figure 2.4. The iterates are updated according to

$$z_{k+1} := \pi_{\mathbb{K}} \left( y_k - \frac{1}{L} \nabla f(y_k) \right),$$

(2.26)

$$y_{k+1} := z_k + \beta_k (z_{k+1} - z_k),$$

(2.27)

where different choices of $\beta_k$ lead to different variants of the method.

Both methods can be interpreted as two connected dynamical systems, as shown in Figure 2.5, where the solution to the optimization problem is a steady-state value of the overall system. The nonlinear system is memoryless and implements the projection
operation. For a quadratic cost function like \( (2.2a) \), the output of the linear dynamical system, say \( t_k \), is a simple gain for the projected gradient method

\[
t_k = (I - \frac{1}{L} H) z_k - \frac{1}{L} h,
\]

(2.28)

and a 2-tap low-pass finite impulse response (FIR) filter for the fast gradient method

\[
t_k = (I - \frac{1}{L} H) \beta_k z_k + (I - \frac{1}{L} H)(1 - \beta_k) z_{k-1} - \frac{1}{L} h.
\]

(2.29)

Even though it has been proven that it is not possible to derive a method that uses only first-order information and has better theoretical convergence bounds than the fast gradient method [165], in certain cases one can obtain faster practical convergence by using different filters in place of the linear dynamical system in Figure 2.5 [54].

**Augmented Lagrangians**

In the presence of equality constraints, in order to be able to apply first-order methods one has to solve the dual problem via Lagrange relaxation of the equality constraints

\[
\sup_{\nu} g(\nu) := \min_{z \in \mathcal{K}} f(z) + \sum_{i \in \mathcal{E}} \nu_i c_i(z).
\]

(2.30)

For both projected gradient and fast gradient methods one has to compute the gradient of the dual function, which is itself an optimization problem

\[
\nabla g(\nu) = c(z^*(\nu))
\]

(2.31)

where

\[
z^*(\nu) := \arg \min_{z \in \mathcal{K}} f(z) + \sum_{i \in \mathcal{E}} \nu_i c_i(z).
\]

(2.32)

When the objective function is separable, i.e. \( f(z) := f_1(z_1) + f_2(z_2) + f_3(z_3) + \ldots \), the inner problem (2.32) is also separable since \( c_i(z) \) is an affine function, hence one can solve several independent smaller optimization problems to compute the gradient (2.31). This procedure, which will be discussed again in Chapter 7, is sometimes referred to as dual
Figure 2.6: Dual and augmented dual functions for a toy problem

with $H = \begin{bmatrix} 0.85 & 0.76 & 0.68 \\ 0.76 & 1.46 & 1.14 \\ 0.68 & 1.14 & 0.94 \end{bmatrix}$, $h = [0.19 \ 0.08 \ 0.74]$, $F = [0.44 \ 0.16 \ 0.88]$, 
$f = 0.27$, with all variables constrained in the interval $(-0.29, 0.29)$ and 
optimal Lagrange multiplier $\nu^* = -0.94$. For the augmented dual function 
$\rho = 2$. 

decomposition in the distributed optimization literature [41].

However, gradient-based methods for solving problem (2.30) typically exhibit very slow 
convergence because $g(\nu)$ is not necessarily strongly concave, as shown in Figure 2.6. In 
order to overcome this problem one can add a quadratic regularizing term to form the 
so-called augmented Lagrangian [94,180] and, instead, solve the following problem 

$$
\sup_{\nu} g(\nu) := \min_{z \in K} f(z) + \sum_{i \in E} \nu_i c_i(z) + \frac{\rho}{2} \sum_{i \in E} c_i^2(z).
$$

However, this renders the inner minimization problem for the computation of the gradient 
of the dual function non-separable, even if the objective function is separable, since $c_i^2(z)$ 
couples variables together. As a consequence, it can no longer be solved in a parallel or 
distributed manner.

**Alternating directions method of multipliers (ADMM)**

The motivation behind ADMM [68,75] is to keep the good robustness properties of the 
augmented Lagrangian method but still be able to solve the subproblems in a distributed 
fashion. The method works by splitting the variables into several groups and solving the problem 

$$
\sup_{\nu} g(\nu) := \min_{z_1 \in K_1, z_2 \in K_2} f_1(z_1) + f_2(z_2) + \sum_{i \in E} \nu_i c_i(z_1, z_2) + \frac{\rho}{2} \sum_{i \in E} c_i^2(z_1, z_2).
$$

Note that while we have split the variables into two groups for clarity of presentation, it is possible to split the original variables into an arbitrary number of groups. Of course,
depending on the specific problem, there will be many different possible splittings that will result in different methods. The ADMM steps for computing the gradient of the dual function and taking a step in the direction of the gradient are given by

\[ z_{1,k+1} := \arg \min_{z \in K_1} f_1(z) + f_2(z_{2,k}) + \sum_{i \in E} \nu_{i,k} c_i(z, z_{2,k}) + \frac{\rho}{2} \sum_{i \in E} c_i^2(z, z_{2,k}), \quad (2.33) \]

\[ z_{2,k+1} := \arg \min_{z \in K_2} f_1(z_{1,k+1}) + f_2(z) + \sum_{i \in E} \nu_{i,k} c_i(z_{1,k+1}, z) + \frac{\rho}{2} \sum_{i \in E} c_i^2(z_{1,k+1}, z), \quad (2.34) \]

\[ \nu_{k+1} := \nu_k + \rho c(z_{1,k+1}, z_{2,k+1}) \sim \nu_k + \rho \nabla g(\nu). \quad (2.35) \]

Note that \( z_2 \) and \( z_1 \) are constant in steps (2.33) and (2.34), respectively, so there is no coupling between \( z_1 \) and \( z_2 \) despite the augmenting regularizing term.

### 2.3 The need for efficient computing

Model predictive control requires solving a problem like (2.1) to determine the control actions to be applied to the plant at every sampling instant. For certain problem formulations it is possible to precompute the solution map offline explicitly in the form of a piecewise affine function [13], i.e. if \( p \) is the parameter that changes between problem instances, the optimal solution is given by

\[ z^* = \begin{cases} 
  g_1(p), & \forall p \in P_1, \\
  g_2(p), & \forall p \in P_2, \\
  \vdots \\
  g_N(p), & \forall p \in P_N, 
\end{cases} \quad (2.36) \]

where all functions \( g \) are affine, \( N \) is the number of pieces in the solution map, and \( P_1 \cup P_2 \cup \ldots \cup P_N = P \), the parameter space. This approach is referred to as explicit MPC. Online implementation is reduced to determining the set to which \( p \) belongs to and evaluating an affine function. There have been several efforts to integrate the design of explicit control algorithms and embedded circuits that have further increased the efficiency in performing this operation [35, 179]. However, this approach is only practical for parameters of low dimensionality, say smaller than four. For larger problems, the number of pieces necessary to describe the control law (2.36) increases very quickly making the approach impractical, mainly due to prohibitive memory requirements, forcing a return to solving (2.1) via online numerical optimization methods like the ones described in Section 2.2.

The goal of extending the use of optimal decision making, and model predictive control in particular, to systems requiring faster decision updates and systems that are restricted to employing low capability computing hardware relies on improving the computational efficiency of online solutions for problem (2.1). By improving the computational efficiency we mean extracting more useful computing performance out of a fixed amount of logic or
computing resource. The latest trends in CMOS technology (see Chapter 3) suggest that improvements in computing efficiency purely from integrated circuit fabrication technology will not be enough for some potential optimal decision making applications.

There are many complementary approaches for improving the computational efficiency. Most techniques in the embedded optimization literature approach the problem by reducing the operation count by means of more efficient algorithms and algorithm modifications. Some of these techniques include warm-starting, where the solver is initialized with the solution to the previous instance with the goal of improving the convergence based on the observation that contiguous optimization problems will be very similar (especially if the sampling rate is high and the process dynamics are slow). Other techniques, like the truncated Newton method [160], can achieve a reduction in complexity by reducing the accuracy required in the first interior-point iterations when the iterate is far from the optimal solution. In a similar spirit, the Levenberg-Marquart method [144] starts with a low complexity first-order method and switches to a more complex second-order method once inside the quadratic convergence region, a property that can be easily checked.

This work is independent of the computing technology. In line with this work, in this thesis we have approached the problem of deciding how to formulate a model predictive control problem as a mathematical optimization problem to reduce the computational count for solving it regardless of the optimization method and computational platform used. This is the subject of Chapter 4.

However, the remaining topics in this thesis are motivated by the potential synergies that can be obtained by designing the optimization algorithms and computing hardware simultaneously. A potential approach to improving the computing efficiency is to work with suboptimal solutions and study the behaviour of the process when driven by suboptimal decisions. This approach has been mentioned in the literature in the context of reducing the computation count through early termination of the algorithms [222], however, a more general question that exposes more degrees of freedom and can deliver greater efficiency gains is whether the knowledge about the behaviour of a process under suboptimal decisions can be used to decide the form that the computing hardware should take given an error tolerance at the solution. This point is briefly addressed in Chapter 6 with an empirical study that indicates the potential efficiency improvements that could arise from further study in this area.

Given the current trends in computing hardware towards parallel architectures, a clear way for improving the computing efficiency is to study the parallelization opportunities in existing optimization methods and develop new algorithms that can better exploit this relatively new computing paradigm. In Chapters 5 and 6 we study custom computing architectures for maximizing the computing efficiency when implementing interior-point methods and several first-order methods in a parallel fashion. In order to best exploit the characteristics of these parallel architectures and improve the computing efficiency further, several tailored high-level model predictive control algorithms are proposed in Chapter 7.

Another approach to achieve efficiency gains is through the use of more efficient arith-
metric. In Chapter 8 we propose algorithm modifications that allow one to solve problems that have traditionally been considered floating-point problems using significantly more efficient fixed-point arithmetic. These problems lie at the heart of popular optimization algorithms such as interior-point and active-set methods, but there is still room for further efficiency gains by extending this approach to the entire algorithm.
3 Computing Technology Spectrum

Since the birth of modern computing there has been a continuous sequence of fabrication and architectural innovations that have led to an exponential improvement in absolute computing performance, defined as

\[
\frac{\text{operations}}{\text{second}} := \frac{\text{cycles}}{\text{second}} \times \frac{\text{operations}}{\text{cycle}}.
\]

(3.1)

For potential embedded resource-constrained optimal decision making applications, (3.1) is not the only important figure of merit. The cost, or how well the resources available are being used to achieve a certain level of performance, and the timing predictability are also key factors. Since these two factors depend to a great extent on a good match between algorithm and computing platform, in this chapter our goal is to develop an understanding of the architectural characteristics that make a computing machine efficient for performing certain tasks. This will help to co-design machines and algorithms for improving the computational efficiency when solving optimal decision making problems. We will also describe the hardware and software features that affect our capability for accurately predicting timing.

3.1 Technology trends

In this section we focus on the most common computing platform, the general-purpose microprocessor. Its advantages and disadvantages are discussed to explain the existence of alternative computing technologies that deviate from the mainstream. We examine the reasons for the current state of technology to anticipate the direction that computing technology is likely to follow and help shape the development of optimization algorithms that are better suited for future computing platforms.

3.1.1 The general-purpose microprocessor

Since Robert Noyce, co-founder of Intel, invented the silicon integrated circuit in 1959 [167] the rate of progress in the capabilities of computing machinery has been incredible. Gordon Moore, the other co-founder of Intel, famously predicted that the amount of transistors in a given amount of silicon would double approximately every 18 months [152]. This rate of growth in transistor density, which still holds today, has been sustained over the years with continuous innovations in integrated circuit manufacturing that have led to increasingly faster and smaller transistors.
Until very recently the trend has been to use the faster switching transistors to boost clock frequencies and use the excess in transistors to devise hardware architectures to make a sequential set of instructions execute faster and faster. The simplicity of the sequential programming model allowed for an efficient high-level programming abstraction that increased the productivity of a large base of software developers. This led to great progress in software for sequential machines that in turn spurred further investment into sequential hardware that could support further progress in software capabilities. For the great majority of applications, there was little incentive to think outside of the sequential programming model since the vigorous progress in sequential hardware meant that if the computing performance required for the next generation of applications was not already available, it would certainly be available in the near future.

The dominant microarchitecture for general-purpose computing has been the x86 instruction set architecture first realized in the 16-bit Intel 8086 in 1978. A slightly modified version was used in the first IBM personal computer and since then there have been 32-bit versions first introduced in the Intel 80386 in 1985 and 64-bit versions appearing recently. The main suppliers of x86-based processors are currently Intel and AMD and their devices power most desktop workstations, personal portable computers and servers.

In the remainder of this section we give an overview of the main techniques that have been used over the years in x86-based machines to improve the computing performance, as described by (3.1), with the goal of explaining the relatively low computational efficiency of these machines and their limitations for providing predictable execution times.

Instruction pipeline

A sequential machine has to perform several tasks to complete one instruction. In general, these tasks involve fetching an instruction from the instruction memory, decoding it, executing it, storing the result in memory and updating the local registers. Each of these tasks is handled by a separate hardware block. In digital designs the clock frequency is determined by the longest delay between two latches, where a latch is a device that holds its output value constant within one clock cycle. A common technique to increase the clock frequency in microprocessor design is to insert several latches between the different hardware blocks required to execute one instruction. This increases the overall signal delay (or latency) for executing each instruction but it allows each hardware block to operate on a different instruction at each clock cycle, as illustrated in Figure 3.1, potentially achieving a throughput of one instruction per cycle.

Figure 3.1 represents a very simple scenario. Modern x86-based machines have significantly more complex instructions and are pipelined more aggressively to minimize inter-latch signal delays and maximize the clock frequency. For instance, the Intel Xeon processor found in high-end desktop machines and servers has a 31-stage pipeline.

In order to achieve a throughput of one instruction per cycle it is necessary for the sequence of instructions to be independent. The deeper the pipeline, the more independent instructions that are needed to sustain maximum throughput. In x86-based machines, the
Figure 3.1: Ideal instruction pipeline execution with five instructions (A to E). Time progresses from left to right and each vertical block represents one clock cycle. F, D, E, M and W stand for instruction fetching, instruction decoding, execution, memory storage and register writeback, respectively.

An approach to exploit the so-called instruction level parallelism (ILP) in sequential software has been to use more transistors to (modestly) increase the number of useful instruction per cycle. For instance, with respect to Figure 3.1, if one of the operands of instruction B depends on the result to instruction A, one can either insert no operation (idle) instructions to avoid operating on the wrong data, or have a hardware block that re-schedules operations on-the-fly to allow for out-of-order execution and reduce the number of idle operations that have to be introduced to preserve correctness [92]. While this strategy can increase sequential performance at the cost of extra transistors, aggressive online instruction scheduling severely hinders our capability for accurately predicting timing [121].

A further problem arises when the sequential code has conditional statements. In this case, when a branching instruction is taken, all of the instructions in the pipeline have to be discarded. The longer the pipeline the greater the overhead associated with this operation. Again, the approach in x86-based machines has been to use more transistors to compute execution statistics on-the-fly to speculate [103] on the chances of a particular branch being taken and schedule instructions accordingly [79]. This strategy adds further timing uncertainty. It should be noted that timing uncertainty is acceptable in general-purpose computing since timing only matters in an aggregate sense.

Hyperthreading [145] (as it is referred to by Intel) is another strategy with a smaller transistor footprint that has been used to exploit ILP in sequential software. In this case, several independent threads or programs are executed on the same instruction pipeline in a time multiplexed fashion. While this can increase the chances of having more independent instructions available to avoid stalling the pipeline, it can also cause contention on the scarce local memory resources, which can lead to slower overall execution. Superscalar microprocessor architectures [110] have several arithmetic units for the execution stage and have an extra hardware block that analyses the incoming instruction sequence on-the-fly and dispatches instructions for parallel execution whenever possible. Both superscalar and hyperthreading affect the timing predictability.

Another approach to increase the number of instructions per cycle has come through x86 instruction extensions [56] that allow the same operation to be applied to multiple
data simultaneously when the pieces of data are smaller than the register word-length. This is known as single instruction multiple data parallelism [64]. These extensions, first introduced as MMX in the 1996 Intel Pentium and called SSE in subsequent revisions, were devised for improving the performance of emerging Internet multimedia applications that involved a lot of similar operations on small data.

**Memory hierarchy**

Programs need memory to store intermediate results. The trade-off between memory access times and cost has driven the way memory is organized in a computing system. The memory subsystem consists of a hierarchy of memories, as illustrated by Figure 3.2, where expensive fast memory, used to store data that is being used often, is placed closer to the arithmetic units to minimize the time wasted transferring data. Since the cost of implementing and operating memories increases with memory speed, the memory hierarchy is designed to have memories of increasing size with the distance from the processing unit. Magnetic disk can store a large amount of data inexpensively but access times are in the order of hundreds of clock cycles, while access times for DRAM (dynamic RAM) are typically between 50 and 100 cycles. More expensive SRAM (static RAM) can be used to buffer data on-chip and reduce access times to less than 10 cycles. Registers, typically implemented using flip-flops, are next to the arithmetic unit and their access time is one cycle, hence compilers optimize programs in order to perform operations using registers as often as possible. Unfortunately, the amount of registers is limited by the number of bits needed to address them, which account for most of the available bits in an instruction.

On-chip memories can take different forms depending on the computing platform. Specific locations in scratchpad memories [10] can be explicitly addressed in software giving greater control and predictability but adding complexity to the programming task. On the contrary, cache memories [205] store a duplicate of a section of main memory and are generally hardware controlled. x86-based machines use cache memories because they allow to abstract away the memory hierarchy and present it to the programmer as a linear address space. This simplifies the programming task significantly but introduces timing uncertainty since cache misses (when the data to be addressed is not present in the
cache) have a major impact on performance and cannot be easily predicted. In fact, the performance has become so dependent on optimal cache utilization that the use of highly optimized libraries such as LAPACK [5] (linear algebra package) are essential for achieving high performance for scientific computations that operate on data that cannot fit inside the processor cache. The trend in x86-base machines has been to use more transistors to continuously increase cache sizes to minimise the chance of cache misses, reaching a point where it is not uncommon for cache to account for more than 50% of the transistors in a modern general-purpose microprocessor.

**x86 is not well suited for our needs**

The philosophy that has driven architectural decisions in x86-based machines has been to simplify the task for the software programmer as much as possible and use shrinking transistor sizes to employ a larger amount of transistors to increase the utilization of the execution pipeline. This has led to the introduction of caches that transfer more data than necessary and burn more power, and the use of extra hardware blocks to perform speculation for exploiting ILP. While these techniques undoubtedly improve the performance of a single thread, the large resource overheads lead to significantly lower performance per Watt [177], which is already a key factor for many resource constrained embedded applications and will be a key factor for future progress in all computing machines (see next section). In fact, the proportion of logic dedicated to computation in modern general-purpose microprocessors has dipped below 15%, as shown in Figure 3.3. Furthermore, all the mentioned techniques add execution uncertainty, which is problematic for designing systems with real-time deadlines.
3.1.2 CMOS technology limitations

Operating transistors consumes electrical energy, which is converted into heat energy. The generated heat has to be dissipated fast enough to avoid several problems. Firstly, transistors exposed to high temperatures degrade faster and have greater resistance, both factors affecting the achievable clock speeds. In the worst case, the degradation can lead to premature chip failure. Secondly, when the heat energy to be dissipated goes above a certain threshold, additional cooling mechanisms, such as fans, are required to keep the temperature within a safe operating interval. These cooling mechanisms take up valuable space in embedded applications and the energy cost of operating them can be a significant fraction of the cost of operating large computing installations such as data centers. In addition, the functionality of embedded devices that run on batteries is directly dependent on the amount of electrical power used by the chip.

In the early days of integrated circuit manufacturing, CMOS (complementary MOS-FET) technology won over bipolar and NMOS technology precisely due to its favourable power characteristics. In theory, CMOS devices only consume power when switching. The average power density in CMOS integrated circuits is given by

$$\text{power density} = d_t e_o f_c v_s,$$

where $d_t$ is the transistor density, $e_o$ is the switching energy, $f_c$ is the clock frequency and $v_s$ is the supply voltage. In the past, continuously decreasing transistor feature sizes, $\kappa$, led to:

- increasing transistor density by a factor of $\kappa^2$,
- reduction in the energy per switching operation by a factor of $\frac{1}{\kappa}$ due to a reduction in parasitic capacitance,
- increasing clock frequency by a factor of $\kappa$ due to decreasing signal delays,
- decreasing supply voltage by a factor of $\frac{1}{\kappa^2}$ due to a reduction in MOSFET threshold voltage.

While these trends were maintained, decreasing feature sizes meant faster chips with more resources while the power envelope was kept constant. However, the limitations of CMOS technology have recently affected the last trend in our list. CMOS transistors are not perfect switches, hence they leak current even when they are turned off. This phenomenon worsens as the threshold voltage decreases, limiting the possibility of reducing the supply voltage indefinitely. For feature sizes below 90nm, leakage currents have had a non-negligible effect on the total chip power. In fact, transistors are so leaky in a current microprocessor that some chips can consume 50 watts of power while standing still [33]. In the latest microprocessor generations, it has only been possible to decrease the supply voltage by a factor of $\frac{1}{\kappa}$ or less, hence the clock frequency has had to be kept constant or even reduced to keep the power density in the chip within a safe operating interval.
As a result, power consumption is the main factor limiting the performance of the sequential general-purpose microprocessor [156]. While, even if still distant, new technologies such as carbon nanotubes could help to overcome the limits of CMOS technology, these new technologies will also have their own power limitations, hence power efficiency will be a key factor for future progress in computing machinery [67], both for embedded and high-performance computing.

### 3.1.3 Sequential and parallel computing

Even though processor clock frequencies have stopped scaling as a result of the so-called power wall, Moore’s trend is still applying and the transistor density of new integrated circuits continues to increase at almost the same rate as before. Consequently, the focus for acceleration and performance improvement has shifted towards parallelism [8]. In the general-purpose domain, the general approach has been to design multicore chips consisting of two or more microprocessors in the same die attached to a shared memory bus. Shared caches in multicore processors can make timing even more unpredictable since programs being executed on one core can trigger cache misses on another program. A further problem for multicore general-purpose computing is that old software, even if compatible, cannot be easily parallelized to run efficiently on these new parallel machines, hence a new programming model is necessary for efficient use. Since the rise of the x86 architecture was mainly based on a simple programming abstraction and the fact that the same code kept running faster and faster on new machines, this paradigm shift poses a considerable threat to the conventional microprocessor business model going forward.

Besides multicore, parallel computing presents other additional fundamental challenges. A non-obvious requirement for an application to have acceleration potential through parallel computing is that it should be compute bound, i.e. the ratio between arithmetic operations and I/O operations should be greater than one. If an application is I/O bound, it means that the performance will be limited by how fast data can be transferred in and out of a chip regardless of the amount of parallelisation employed. An example I/O bound application is the matrix-vector multiplication $Ax$, where $A \in \mathbb{R}^{n \times n}$ and $x \in \mathbb{R}^n$. In this case, there are $n^2 + 2n$ I/O operations and $n(2n + 1)$ compute operations for each matrix-vector multiplication, hence, even if the operation offers many parallelisation opportunities, if all the data has to be loaded every time from memory, the performance will be limited by the memory bandwidth of the machine. In general, memory bandwidth is growing significantly slower than computational capabilities. In fact, it is fundamentally limited by the number of pins one can have in an integrated circuit. Fortunately, in model predictive control, if all the problem data can be stored using on-chip memories, the amount of I/O is significantly smaller than the arithmetic requirements. Hence, being able to store all problem data using on-chip memories is of crucial importance for accelerating predictive control applications in parallel hardware [106].

Needless to say, an application has to have enough inherent parallelism for it to be
accelerated on a parallel machine. In order to determine the potential speed-up the key characteristic is the proportion of the application that has sequential dependencies, and the proportion of operations that are independent, $P$. Amdahl’s law [4] states that the potential acceleration with $R$ parallel resources, assuming enough memory bandwidth is available and instantaneous parallel execution, is given by

$$
\frac{1}{(1 - P) + \frac{P}{R}}.
$$

This means that if an application has 25% of sequential code, even with an infinite amount of parallel resources, the speed-up will never be larger than 4x. Under the outlook of cheap transistors in the future, it is not clear that the absolute number of operations required to implement an algorithm will be important anymore. Rather, algorithms with smaller data dependencies will execute faster even if they are slower on a traditional sequential machine. This point should be considered for the development of new algorithms for optimal decision making too.

Amdahl’s law presents a theoretical upper bound for potential acceleration, but in multicore general-purpose computing the situation is far from this ideal bound. Even if an application has very few sequential dependencies, the performance with an increasing number of cores does not follow Amdahl’s law, mainly due to contention on the memory resources between parallel threads, and the need for synchronizing different threads with unpredictable timing before returning control to the sequential parts of the application [204]. This often leads to poor computational efficiency.

3.1.4 General-purpose and custom computing

A way to overcome the low efficiency problem is through specialisation of the computing hardware to a specific task. The architecture variety for general-purpose computing is very limited. The computing hardware found in your desktop machine has to be able to handle tasks such as running an operating system, word processing, video encoding, sending emails or solving systems of linear equations. General-purpose hardware trades resource efficiency for the ability to carry out many different tasks with very different computing patterns. Throughout this chapter we have gone through the main sources of inefficiencies.

For some classes of applications, such as digital filtering or graphics rendering, there exists domain-specific hardware that can handle the computing patterns found in that particular class more efficiently than a general-purpose machine (see Section 3.2). These computing architectures are also software-programmable to provide some generality. For example, there are many algorithms for processing audio signals but the computing patterns are similar in most of them.

The highest level of customisability is achieved by hardwiring a particular algorithm directly in silicon. This approach offers limited or no software-programmability and its
main benefit is predictability and the ability to control execution. By designing a custom computing datapath and a custom memory subsystem to match that datapath, it is possible to provide just the necessary memory bandwidth to keep the arithmetic units busy all the time, avoid or minimize contention on the compute and memory resources, perfectly synchronise independent computations, and achieve high computational efficiency. In addition, since the circuit is designed for executing only one algorithm one can completely avoid having redundant speculative circuits burning power unnecessarily.

**Number representation**

When designing a custom computing architecture for a particular application, the designer is free to decide how to represent data. This choice has a large impact on the amount of resources needed to store and perform arithmetic operations on that data. As a result, the extractable parallelism for a fixed silicon budget is dependent on the format and precision used to represent numbers, meaning that in order to maximise the computational efficiency it is important to use a minimal representation for the specific algorithm to behave in a numerically reliable way or for the output to meet the accuracy specifications of the application.

In general-purpose hardware the use of power-hungry 64-bit double precision floating-point units is ubiquitous due to the need to serve many different applications. A floating-point number is represented as

$$s \times 2^\text{bias} \cdot e \times 1.m$$

where $s$ is the sign bit, $e$ is the exponent and $m$ is the mantissa, which lies in the interval $[0, 1)$. The different fields are concatenated in a bit string as shown in Figure 3.4. This format allows one to represent a very wide range of numbers with a small number of bits, which is necessary for a general-purpose computer that has to handle different applications with unknown data ranges. Also in this case, generality leads to inefficient resource use. For example, a floating-point addition requires mantissa alignment according to the difference in the exponents of both operands and denormalisation before and after the core mantissa addition. As a result, a floating-point adder, such as the one shown in Figure 3.5, consists mostly of hardware blocks that are not performing mantissa addition. These use additional resources and increase computational delays.

If there exists available information about the range of the data that the computer will be operating on, one can either use a custom floating-point data format in order to minimise the hardware overhead, or use a fixed-point data format that only consists of
Floating-point multiplications require a number of stages besides the multiplication of the mantissas and the addition of exponents. These stages include the normalization, rounding, and re-normalization of the mantissa, if necessary [3].

In the case of floating-point additions two initial stages are required, one to detect which exponent is the highest, and another to align the mantissa of the lowest number to the same magnitude of the larger number. These stages are illustrated in Fig. 2.

![Floating-point multiplier and adder diagrams showing the alignment stage in the adder and the normalization, rounding and re-normalization stages on both operations. FLO represents "finding leading one."](image)

Floating-point arithmetic defined by the IEEE 754 standard [3] applies to atomic scalar operations, not to composite computations such as dot-product. As such, because of the non-associativity of floating-point addition, re-ordering of operands changes roundoff error. Thus we should see floating-point realizations of dot-products as producing a "family" of possible arithmetic accuracies rather than one single accuracy. Our scheme aims to be indistinguishable from this family under an appropriate measure of accuracy, while out-performing a straightforward floating-point core based implementation.

In the fully parallelized and deeply pipeline dot-product circuit depicted in Fig. 1, where each floating-point operation output is connected to an adder input, there is a recurrent connection between a normalization, rounding and re-normalization circuit and a mantissa alignment circuitry. This recurrent logic

![Fixed-point data format. An imaginary binary point, which has to be taken into account by the programmer, lies between the integer and fraction fields.](image)

the core arithmetic operation and has no hardware overhead. The fixed-point data format is illustrated in Figure 3.6. In this case, arithmetic units are the same as for integer arithmetic. This makes the circuitry much simpler and efficient, however, it introduces new design challenges that will be addressed throughout this thesis in the context of optimization solvers.

For applications dominated by multiplications and divisions, a logarithmic number system that converts these complex operations into hardware-simple additions and subtractions can be an appropriate choice, whereas for applications that have to be especially careful about rounding, e.g. some financial applications, a decimal instead of binary representation is mandatory.

### 3.2 Alternative platforms

In the previous section we have described the technical and economic reasons for the rise of the x86 instruction set architecture for general-purposed computing and we have analyzed the causes for its low computational efficiency and the difficulties it introduces for accurately predicting timing. We also saw how the limitations in CMOS technology have affected microprocessor design in recent years and how these changes challenge the conventional x86 value proposition.
The computing market keeps growing at a healthy rate, both in the high-performance and embedded domains. Since under the current technology situation one of the few ways to extract extra performance is to specialise the computing platform to be efficient at certain kinds of computation, the architecture variety is likely to increase significantly in the near future. In this section, we will describe several computing alternatives to the x86 architecture and analyze their features in the context of real-time optimization solvers. All of the discussed platforms could be suitable for optimal decision making applications with different specifications.

3.2.1 Embedded microcontrollers

An embedded microcontroller includes a processor, memory and programmable I/O peripherals in a single chip. It is typically connected to sensors and actuators and performs only one or a few functions. It runs either no operating system or a thin real-time operating system to guarantee that function executions meet real-time deadlines.

Intel’s x86-based Atom processor with complex instructions has been Intel’s attempt to enter the embedded market. However, the embedded microcontroller market is overwhelmingly dominated by reduced instruction set computers (RISC) based on the ARM, MIPS or PowerPC instruction set architectures. The RISC concept [177] advocates for hardware simplicity through simple instructions. Examples of simple instructions include adding the contents of registers A and B, or storing the contents of register C in a certain memory location. These instructions typically execute in one cycle, which leads to a shorter execution pipeline and a lesser need for speculative ILP-exploiting strategies. As a result of the simplicity of the instructions, the instruction sequence has a more regular structure and strategies for exploiting ILP are significantly simpler and can be mostly left for the compiler, instead of having dedicated hardware blocks. In summary, RISC processors can use fewer transistors than an x86 processor to execute the same code, often leading to significantly lower power consumption and higher computational efficiency.

Embedded microcontrollers range from 8-bit machines running at kiloHertz or single digit megaHertz clock frequencies for extremely low power applications to 32-bit machines running at several gigaHertz for performance-critical applications. The ARM architecture is the leader in this domain and chips based on it are sold by a variety of vendors such as NXP, Texas Instruments, Atmel, Freescale Semiconductors or STMicroelectronics. They can be found in automobiles, washing machines, Apple’s iPad, medical devices, and in more than 90% of all mobile phones.

For implementing optimal decision makers, embedded microcontrollers offer better power and computational efficiency, as well as more predictable timing, making them more suitable than x86 processors for the next generation of resource-constrained real-time optimization applications. Furthermore, they follow the same programming paradigm as general-purpose processors. However, there exist other limitations. The absolute performance is lower than for performance-oriented Intel processors and the simple instructions lead to
larger code sizes, which is an important consideration for embedded systems. There is also limited support for data sizes bigger than 32 bits. Perhaps more importantly, floating-point hardware support is not common, hence, if floating-point computations are required they often have to be emulated in software, which slows down execution significantly.

**Programmable logic controllers**

Programmable logic controllers (PLCs) integrate a microcontroller core with modular inputs and outputs in a single rugged package. Recently, there has been several investigations into their use for small-scale model predictive controllers [97, 185, 215]. Even though PLCs are significantly more expensive than the microcontrollers at their core, they are still widely used in industrial environments for their ruggedness and reliability and because they can be programmed in a higher level abstraction than C with very simple constructs known as ladder logic. In addition, they provide real-time execution monitoring capabilities and have support for simplifying field updates.

### 3.2.2 Digital signal processors

In digital signal processing a very common operation is filtering a stream of data where, in the simplest single-input single-output (SISO) situation, the filtered output at time $n$ is given by

$$y[n] := \sum_{i=0}^{N-1} c_i x[n - i],$$

(3.2)

where $x[n]$ is the input stream, $c_i$ are coefficients and $N$ is the number of taps. In computational terms, this operation requires multiplying two values, adding it to the running total, and repeating the same operation again on adjacent data.

Digital signal processors (DSPs) have an execution pipeline that is specialised for computing operations like (3.2) very efficiently. They include a multiply accumulate unit to perform a multiplication and an addition in one cycle allowing for extended precision in the intermediate result. They are Harvard based, i.e. they have different instruction and data buses, and they have support for simultaneous fetching of several data items from memory. Furthermore, they include hardware support for common addressing modes like auto increment (to support operations such as (3.2)), circular and bit-reversed, which reduce or eliminate addressing overheads. DSPs are complex instruction set (CISC) machines. An example of a complex instruction could be: fetch two pieces of data and put them in registers A and B, multiply them together and add them to the contents of register C, store the result in the same register and increment the pointers to fetch the next data. For more details about the history of DSP processor architectures, see [119].

The market for DSPs is dominated by Texas Instruments. These processors were first used for speech synthesis and data modems but they can now be found in more demanding applications like professional video processing, medical imaging or machine vision. Tra-
ditionally, they have supported fixed-point arithmetic only and they have been mainly used in embedded applications due to their low power consumption, high computational efficiency when handling certain kinds of computation, and lack of hardware features that introduce timing uncertainty. However, there are other operations beyond filtering that follow similar computing patterns to (3.2). For example, dense matrix operations also involve multiply-accumulate operations on adjacent data, although not in a streaming fashion. Since the introduction of multicore DSPs with floating-point support, these devices have been proposed for efficient high-performance computing [99] and there exist efficient libraries [216], such as those available for general-purpose processors, for linear algebra computations.

Optimization solvers are rich in linear algebra operations, so they could benefit from these recent developments. In general, DSPs are harder to program than general-purpose processors, and because they support more complex instructions it is more difficult for the compiler to optimize execution. As a result, hand-coded processor-specific assembly libraries are often needed for efficiency, and the fixed specialised execution pipeline may prove not suitable for certain parts of an optimization solver. So far, investigations into the use of DSPs for model predictive control have been limited [192]. Furthermore, the cheapest and most computationally efficient DSPs only support fixed-point arithmetic, adding further challenges.

Other exotic CISC processors

Very long instruction word (VLIW) computers execute multiple arithmetic operations per instruction. Unlike with superscalar general-purpose processors, parallel execution is determined at compile time, benefiting timing predictability.

3.2.3 Graphics processing units

Graphics processing units (GPUs) were once very specialised devices tailored to perform graphics rendering for video games very efficiently. In the last decade NVIDIA introduced the Compute Unified Device Architecture (CUDA) [168] to expose the computational power of GPUs to non-graphics computing applications. Nowadays, general-purpose GPUs (GPGPUs) have fixed architectures with up to several hundreds of processing units that can perform operations in parallel. The peak theoretical floating-point performance of these devices is extremely high (in the order of TFLOPs). Furthermore, the gaming mass market allows the main vendors, NVIDIA and AMD, to keep very competitive prices.

A CUDA-based architecture is shown in Figure 3.7. The individual cores are simple compared to a general-purpose processor. They have a 4-stage pipeline and are grouped into groups of eight into a streaming multiprocessor (SM). Each core executes the same instruction on different data in a SIMD fashion. The memory subsystem consists of several local registers for each core, 16KB of shared memory for each SM, level 1 cache shared between groups of SMs and a global graphics cache shared between all groups. Note
that the cache and the main memory are distributed, providing much higher memory bandwidth than in a general-purpose processor system.

The programming model follows the so-called single instruction multiple thread (SIMT) paradigm. Independent threads are grouped into blocks and each block is assigned to one SM, which requires 32 independent threads to hide the latency of the execution pipeline. As a consequence, in order to achieve high overall GPGPU utilization there needs to be some hundreds or thousands of independent threads available at all times. The GPGPU approach is known as “throughput computing”, since instead of working to speed up the program’s operation on a single dataset, the system works to increase the rate at which a collection of datasets can be processed by the program. The architecture is also specialised for streaming computations where data is not expected to be reused many times, hence the amount of on-chip cache available is limited.

In contrast to general-purpose multicore, the approach in the GPGPU domain has been to devote a greater proportion of the transistors to computation and a lesser proportion for execution control and speculation. However, this means that the computations and memory accesses have to be extremely regular to achieve close to peak performance. Any
slight deviation from this rule leads to large performance penalties. There have been claims that GPGPUs can provide from 100x to 1000x performance improvements over a general-purpose processor for some applications [169, 203, 211]. The reality is that for most real applications the performance gap is not as large [123].

In principle, GPGPUs seem an attractive option for exploiting the many parallelisation opportunities in optimization solvers. In reality, one would only want to use a GPGPU to accelerate the very regular operations in the solver, say dense linear algebra operations (if they exist), and the performance benefit will only be significant if the problem is large enough to provide enough independent threads. Besides, GPGPUs have several additional properties that make them problematic for embedded applications. Firstly, the order of execution is scheduled by hardware on-the-fly, which given the performance sensitivity of the architecture leads to very unpredictable timing. Secondly, a GPGPU cannot be a standalone component. It requires an additional general-purpose host to transfer data and start execution, hence the cost and, more importantly, the power requirements of the system are very high, typically well above 100 Watts. Lastly, if high accuracy is needed, double precision floating-point computations incur a performance penalty between two and four times [28].

Even though there have been studies for implementing model predictive controllers using GPGPUs [199], we believe that these processors are not suitable for achieving our goal of extending optimal decision making to real-time resource-constrained applications, hence they will not be directly considered in the remainder of this thesis. However, the concept of throughput computing will play an important role in Chapter 7.

**Heterogeneous architectures**

Recently, the main processor designers, Intel (Sandy Bridge), AMD (Fusion accelerated processing units) and ARM (Mali), have released solutions that integrate a GPGPU and a microprocessor sharing an address space on the same chip. In these heterogeneous architectures the code that will run on the CPU is likely to be very different from the one it executes now [7]. It will have limited ILP, hard to predict branching, smaller use of SIMD, and hard to predict memory access patterns, which may help to shape future microprocessor designs.

3.2.4 Field-programmable gate arrays

This section has so far only described fixed architectures. Custom architectures are interesting from the efficiency point of view and also because the flexibility allows one to research novel architectures without being restricted by what is available on the market. However, the non-recurring costs for fabricating application-specific integrated circuits (ASICs) have reached a level that can only be supported by mass markets such as mobile telephony and computer gaming.

Field programmable gate-arrays (FPGAs) are hardware-reconfigurable devices that have
a finely tunable general-purpose fabric that can be programmed to implement specialised circuits. Because FPGAs can be reconfigured after fabrication, the non-recurring engineering costs are amortized over a large number of customer designs, leading to significantly lower prices for the consumer than building a custom ASIC. The approach in FPGAs is similar to GPUs in the sense that most of the hardware resources are devoted to computation. However, the basic computational units are much larger in number (many thousands vs several hundreds for GPUs) and much simpler – look-up tables (LUTs) that can be programmed to implement logical functions with few input bits [90]. These simple LUTs can be combined through a flexible reconfigurable routing network to implement arbitrarily complicated higher level operations such as integer addition or even floating-point division. In addition, there exist dedicated hardware multipliers and RAMs embedded in the reconfigurable fabric.

The leading FPGA suppliers are Xilinx, Altera, and Microsemi to a lesser extent. Initially, FPGAs were conceived for prototyping ASIC designs before being sent to production. Nowadays, Moore’s trend has promoted FPGAs to a level where it is possible to implement full complex high-performing systems on a single chip. FPGAs form the backbone of global communication networks, which have a relatively low number of performance-critical nodes but have extremely high throughput requirements [194]. They are also used for demanding signal processing applications like computer vision and radar, and have also become common for implementing simple control loops with very tight real-time requirements [2]. Beyond streaming applications, FPGAs have also been recently proposed for efficient floating-point implementations of basic linear algebra operations [212, 245].

FPGAs are traditionally programmed using hardware description languages such as VHDL or Verilog [210]. Hardware design flows rely on slow error-prone tools that require low-level hardware expertise. This is often a big limitation for application domain experts. For this reason, there have been considerable efforts for application-independent automatic conversion of high-level code into hardware descriptions. For instance, Xilinx’s AutoESL [36, 238] accepts an annotated C description as an input. There have also been several attempts to convert high-level visual descriptions of programs, which can capture parallel dataflow computations more naturally, into hardware descriptions. Notable examples include Xilinx’s System Generator [237], MathWorks’ HDL coder [209] and National Instruments’ Labview FPGA [161]. In this case, inefficiencies arise when the control structures in the algorithms are more complex than the simple DSP-type algorithms for which these tools were conceived.

For optimization solvers, FPGAs can achieve maximal computational efficiency due the possibility of tailoring the computing architecture to the particular algorithm, promising to extend the use of optimal decision making in resource-constrained applications. Besides, hardware implementations have cycle-accurate predictable timing, which is a significant advantage for guaranteeing tight real-time deadlines. However, FPGAs remain at a higher price level compared to other embedded alternatives such as microcontrollers and DSPs. In addition, floating-point computation, while possible, carries a large overhead due to
lack of explicit hardware support in the FPGA fabric for the alignment operations needed in floating-point arithmetic. Presently, the efficiency gap between fixed-point and floating-point computation in FPGAs is up to two orders of magnitude [107].

This thesis focuses on FPGAs for implementing efficient optimal decision makers, although some of the developed techniques will be equally applicable to other embedded platforms such as microcontrollers and fixed-point DSPs.

Heterogeneous architectures

In a similar spirit to heterogeneous GPU-CPU architectures, there have also been recent releases by the main FPGA manufacturers, Xilinx (Zynq [239]) and Altera (Arria V [3]), which include an ARM dual core microcontroller with clock frequency in the gigahertz region and a large amount of reconfigurable FPGA resources in a single chip.

3.3 Embedded computing platforms for real-time optimal decision making

This chapter has introduced several computer architecture concepts that will be useful for the remainder of this thesis. We have analyzed the microarchitectural features introduced in general-purpose processors to increase the utilization of the execution pipeline, which, together with the memory hierarchy, has helped to explain why modern general-purpose machines can be rather computationally inefficient and have unpredictable timing when carrying out specific tasks, like solving optimization problems, repeatedly.

The need to increase the utilization of the execution pipeline can also arise in the context of custom circuit designs and this is one of the topics in this thesis. However, in our case, the problem will be approached from the derivation of new algorithms that can make better use of this hardware feature rather than by adding redundant hardware blocks to perform speculation.

This chapter has also examined computing technology trends to help to explain the reasons for the recent paradigm shift towards parallelism across the computing spectrum. We have analyzed other alternative fixed architectures in the computing market and described their suitability for embedded optimal decision making applications. One can also anticipate the form that future fixed architectures will take by projecting these technology trends into the future, predicting that the important metric for comparing new optimization algorithms in the near future could become the proportion of parallelisable work rather than the absolute number of operations.

The topic of number representation, which will be a central topic in the following chapters, has been introduced in the context of custom architectures. The rest of this thesis will consider the joint design of computing machines and optimization algorithms for improving the computational efficiency of embedded solutions and hence increase the range of applications that can benefit from real-time optimal decision making.
Chapter 2 described how the very high computational demands of solving optimization problems stand as a barrier that has prevented the use of optimal decision making functionality in applications with resource constraints. In model predictive control, the computational burden depends to a large extent on the way the optimal control problem is formulated as an optimization problem. In this chapter we explore several new and existing optimization formulations for control.

The method employed when formulating a constrained optimal control problem as a quadratic program (QP) has a big impact on the problem size and structure, the resulting computational and memory requirements, as well as on the numerical conditioning. The standard approach makes use of the plant dynamics to eliminate the plant states from the decision variables by expressing them as an explicit function of the current state measurement and future control inputs [139]. This condensed formulation leads to compact and dense quadratic programs. In this case, the complexity of solving the QP scales cubically in the horizon length (how far we predict into the future) when using an interior-point method. For model predictive control problems that require long horizon lengths, the non-condensed formulation, which keeps the plant states as decision variables and considers the system dynamics implicitly by enforcing equality constraints [184, 226, 227], can result in significant speed-ups. With this approach the problem becomes larger but its sparsity structure can be exploited to find a solution in time linear in the horizon length.

The non-condensed formulation is often also referred to as the sparse method due to the abundant structure in the resulting optimization problems. In this chapter, it will be shown that this label does not provide the complete picture and that it is indeed possible to have a sparse condensed formulation that can also be solved in time linear in the horizon length. In addition, it will be shown that this method is at least as fast as the standard condensed formulation and it is faster than the non-condensed formulation for a wide variety of common control problems. Our approach is based on the use of a specific linear feedback policy to simulate a change of variables that results in a quadratic program with banded matrices in cases where the horizon length is larger than the controllability index of the plant. The use of feedback policies for pre-stabilization has been previously studied as an aid for proving stability [195] and as a way of improving the problem conditioning for guaranteed stability MPC algorithms [196]. However, it is surprising that it has not yet been applied to introduce structure into the optimization problem, as we show in this chapter, considering the important practical implications.
Outline

This chapter will start by formally introducing the model predictive control setup in Section 4.1. This setup will be used throughout this thesis. The existing condensed and non-condensed formulations are reviewed in Section 4.2 and their computational complexity and memory requirements are analyzed in the context of several optimization methods. Section 4.3 presents our sparse condensed approach and compares its advantages and limitations with the existing QP formulations. A numerical study is included in Section 4.4 to verify the feasibility of the proposed approach. The chapter concludes with a brief overview of other recent alternative formulations in Section 4.5 and a discussion on open questions in this area in Section 4.6.

4.1 Model predictive control setup

Throughout, we address control of a discrete-time linear time-invariant (LTI) system where the system state at the next sampling instant, assuming a zero-order hold (ZOH), is given by

$$x^+ = Ax + Bu,$$  \hspace{1cm} (4.1)

where \( A \in \mathbb{R}^{n_x \times n_x} \), \( B \in \mathbb{R}^{n_x \times n_u} \), \( x \in \mathbb{R}^{n_x} \) is the current system state and \( u \in \mathbb{R}^{n_u} \) is the system input held constant between sampling instants. As an example, consider the classical problem of stabilizing an inverted pendulum on a moving cart. In this case, the system dynamics are linearized around the upright position to obtain a representation such as (4.1), where the states are the pendulum’s angle displacement and velocity, and the cart’s displacement and velocity. The single input is a horizontal force acting on the cart. The overall design goal is to construct a time-invariant (possibly nonlinear) static state feedback controller \( \mu : \mathbb{R}^{n_x} \rightarrow \mathbb{R}^{n_u} \) such that \( u = \mu(x) \) stabilizes the system (4.1) while simultaneously satisfying a collection of state and input constraints in the time domain.

In the inverted pendulum case, the control objective could be to maintain the pendulum angle close to zero.

In standard design methods for constructing linear controllers for systems in the form (4.1), the bulk of the computational effort is spent offline in identifying a suitable controller, whose online implementation has minimal computing requirements. The inclusion of state and input constraints renders most such design methods unsuitable. A now standard alternative is to use MPC \([139, 187]\), which moves the bulk of the required computationally effort online and which addresses directly the system constraints. At every sampling instant, given an estimate or measurement of the current state of the plant \( x \), an MPC controller solves a constrained \( N \)-stage optimal control problem in the form
\[ J^*(x) := \min_{u_0, x_0, \delta_0, \ldots, x_{N-1}, x_N, \delta_N} \frac{1}{2} (x_N - x_{ss})^T Q N (x_N - x_{ss}) \]
\[ + \frac{1}{2} \sum_{k=0}^{N-1} (x_k - x_{ss})^T Q (x_k - x_{ss}) + (u_k - u_{ss})^T R (u_k - u_{ss}) \]
\[ + \sum_{k=0}^{N} (x_k - x_{ss})^T S (u_k - u_{ss}) + \sum_{k=0}^{N} (\sigma_1 \cdot 1^T \delta_k + \sigma_2 \cdot ||\delta_k||^2) \]
subject to
\[ x_0 = x, \quad (4.3a) \]
\[ x_{k+1} = A_d x_k + B_d u_k + B_w \hat{w}, \quad k = 0, 1, \ldots, N - 1, \quad (4.3b) \]
\[ u_k = K x_k + v_k, \quad k = 0, 1, \ldots, N - 1, \quad (4.3c) \]
\[ u_k \in U, \quad k = 0, 1, \ldots, N - 1, \quad (4.3d) \]
\[ (x_k, \delta_k) \in X_\Delta, \quad k = 0, 1, \ldots, N. \quad (4.3e) \]

where \( x_{ss} \) and \( u_{ss} \) are steady-state references for the states and inputs given by the target calculator (refer to figure 2.2), and \( \hat{w} \) is a disturbance estimate which is zero when estimates are not available. For clarity, the term \( B_w \hat{w} \) is omitted in the analysis in this chapter.

If a feasible optimal input sequence \( \{ u^*_i(x) \}_{i=0}^{N-1} \) and state trajectory \( \{ x^*_i(x) \}_{i=0}^{N-1} \) exists for this problem given the initial state \( x \) (and disturbance estimate \( \hat{w} \)), then an MPC controller can be implemented by applying the control input \( u = u^*_0(x) \).

The system states can have both free (index set \( F \)), hard-constrained (index set \( B \)) and soft-constrained (index set \( S \)) components, i.e. the set \( X_\Delta \) in (4.2) is defined as

\[ X_\Delta = \left\{ (x, \delta) \in \mathbb{R}^{nx} \times \mathbb{R}^{[S]} \mid x_F \text{ free}, x_{\min} \leq x_B \leq x_{\max}, |x_i - x_{i,ss}| \leq r_i + \delta_i, i \in S \right\}, \]

with \( x_{c,i} \in \mathbb{R} \) being the center of the interval constraint of radius \( r_i > 0 \) for a soft-constrained state component. The index sets \( F, B, S \) are assumed to be pairwise disjoint and to satisfy \( F \cup B \cup S = \{1, 2, \ldots, n_x \} \).

It is assumed throughout that the pair \( (A_d, B_d) \) is controllable, \( (Q^\frac{1}{2}, A_d) \) is detectable, the penalty matrices \( (Q, Q_N) \in \mathbb{R}^{nx \times nx} \) are positive semidefinite, \( R \in \mathbb{R}^{nu \times nu} \) is strictly positive definite, and \( S \in \mathbb{R}^{nu \times nu} \) is chosen such that the objective function in (4.2) is jointly convex in the states and inputs. There is by now a considerable body of literature [147, 187] describing conditions on the penalty matrices and/or horizon length \( N \) sufficient to ensure that the resulting MPC controller is stabilizing (even when no terminal state constraints are imposed), and we do not address this point further. For stability conditions for soft-constrained problems, the reader is referred to [242] and [202] and the references therein.

Note that (4.3c) is effectively only a change of variables and it does not modify the
optimal control problem, hence the computed optimal input is independent of the transformation used. Moreover, any procedure to guarantee stability and feasibility can still be used.

If the soft-constrained index set $\mathcal{S}$ is nonempty, then a linear-quadratic penalty on the slack variables $\delta_k \in \mathbb{R}^{\mid \mathcal{S} \mid}$, weighted by positive scalars $(\sigma_1, \sigma_2)$, can be added to the objective. In practice, soft constraints are a common measure to avoid infeasibility of the MPC problem (4.2) in the presence of disturbances. However, there also exist hard state constraints that can always be enforced and cannot lead to infeasibility, such as state constraints arising from remodeling of input-rate constraints (see below). For the sake of generality we address both types of state constraints in the problem setup. The presence of soft state constraints will have a large impact on the methods described in Chapter 6 and a lesser impact on the rest.

If $\sigma_1$ is chosen large enough, then the optimization problem (4.2) corresponds to an exact penalty reformulation of the associated hard-constrained problem (i.e. one in which the optimal solution of (4.2) maintains $\delta_k = 0$ if it is possible to do so). An exact penalty formulation preserves the optimal behavior of the MPC controller when all constraints can be enforced. We first characterize conditions under which a soft constraint penalty function for a convex optimization problem is exact.

**Theorem 1** (Exact Penalty Function for Convex Programming [16, Prop. 5.4.5]). Consider the convex problem

$$f^* := \min_{z \in \mathcal{Q}} f(z)$$

subject to

$$g_j(z) \leq 0, \quad j = 1, 2, \ldots, r,$$

where $f : \mathbb{R}^n \to \mathbb{R}$ and $g_j : \mathbb{R}^n \to \mathbb{R}$, $j = 1, \ldots, r$, are convex, real-valued functions and $\mathcal{Q}$ is a closed convex subset of $\mathbb{R}^n$. Assume that an optimal solution $z^*$ exists with $f(z^*) = f^*$, strong duality holds and an optimal Lagrange multiplier vector $\mu^* \in \mathbb{R}^r_+$ for the inequality constraints exists.

i. If $\sigma_1 \geq \|\mu^*\|_\infty$ and $\sigma_2 \geq 0$, then

$$f^* = \min_{z \in \mathcal{Q}} f(z) + \sum_{j=1}^r \left( \sigma_1 \cdot \delta_j + \sigma_2 \cdot \delta_j^2 \right)$$

subject to

$$g_j(z) \leq \delta_j, \quad \delta_j \geq 0, \quad j = 1, 2, \ldots, r.$$
ii. If \( \sigma_1 > \|\mu^*\|_\infty \) and \( \sigma_2 \geq 0 \), the set of minimizers of the penalty reformulation in (4.5) coincides with the set of minimizers of the original problem in (4.4).

In the context of the MPC problem (4.2), the penalty reformulation is exact if the penalty parameter \( \sigma_1 \) is chosen to be greater than the largest Lagrange multiplier for any constraint \( |x_i - x_{c,i}| \leq r_i, i \in S \), over all feasible initial states \( x \). In general, this bound is unknown a priori and is treated as a tuning parameter in the control design. The quadratic penalty parameter \( \sigma_2 \) need not be nonzero for such a penalty formulation to be exact, but the inclusion of a nonzero quadratic term can improve the conditioning of the problem and is necessary for the numerical stability results that will be presented in Chapter 6.

**Input-rate constraints**

In addition to constraints on the control inputs and plant states it is not uncommon to have constraints on the actuator slew rate, i.e. \( \Delta u_{\text{min}} \leq u - u^- \leq \Delta u_{\text{max}} \), due to physical limitations of the actuators. There are several approaches for enforcing these constraints. One alternative is to augment the state such that

\[
x \leftarrow \begin{bmatrix} x \\ u^- \end{bmatrix}, \quad u \leftarrow u - u^-,
A_d \leftarrow \begin{bmatrix} A_d & B_d \\ 0 & I \end{bmatrix}, \quad B_d \leftarrow \begin{bmatrix} B_d \\ I \end{bmatrix}, \quad Q \leftarrow \begin{bmatrix} Q & S \\ S^T & R \end{bmatrix}, \quad S \leftarrow 0 \quad \text{and} \quad R \text{ is overwritten by a penalty on } \Delta u.
\]

In this case, the state dimension becomes \( n_x + n_u \) and the free variables are the state vector and the input-rates.

One can avoid increasing the size of the optimization problem by writing the input-rate constraints and the constraints (4.3d)-(4.3e) in the form

\[
J \begin{bmatrix} x_{\mathcal{B},0} \\ x_{\mathcal{S},0} \\ \delta_0 \end{bmatrix} + E_0 u_0 \leq d,
\]

\[
J \begin{bmatrix} x_{\mathcal{B},k} \\ x_{\mathcal{S},k} \\ \delta_k \end{bmatrix} + \begin{bmatrix} E & E^- \end{bmatrix} \begin{bmatrix} u_k \\ u_{k-1} \end{bmatrix} \leq d, \quad k = 1, \ldots, N - 1,
\]

\[
J_N \begin{bmatrix} x_{\mathcal{B},N} \\ x_{\mathcal{S},N} \\ \delta_N \end{bmatrix} \leq d_N.
\]

For the case where the input constraint set \( \mathcal{U} \) is defined as a set of interval constraints
\[ U := \{ u \mid u_{\text{min}} \leq u \leq u_{\text{max}} \} \], we have

\[ J := \begin{bmatrix}
I & 0 & 0 \\
-I & 0 & 0 \\
0 & 0 & -I \\
0 & I & -I \\
0 & -I & -I \\
0 & 0 & 0 \\
0 & 0 & 0 \\
0 & 0 & 0 \\
0 & 0 & 0 \\
0 & 0 & 0 \\
\end{bmatrix}, \quad J_N := \begin{bmatrix}
I & 0 & 0 \\
-I & 0 & 0 \\
0 & 0 & -I \\
0 & I & -I \\
0 & -I & -I \\
\end{bmatrix}, \quad d_N := \begin{bmatrix}
x_{\text{max}} \\
x_{\text{min}} \\
0 \\
0 \\
r + x_c \\
r - x_c \\
u_{\text{max}} \\
-\Delta u_{\text{max}} \\
-\Delta u_{\text{min}} \\
\end{bmatrix} \]

Note that this approach does not increase the size of the optimization problem but will affect the structure of the matrices under certain formulations.

### 4.2 Existing formulations

Consider the problem of formulating the optimal control problem (4.2) as a convex quadratic program of the form:

\[
\begin{align*}
\min_z & \quad \frac{1}{2} z^T H z + h^T z \\
\text{subject to} & \quad Fz = f, \\
& \quad Gz \leq g.
\end{align*}
\]

Primal-dual interior-point methods can be used to solve for optimal \( z \). If the augmented formulation is used (refer to Section 2.2.1), the main operation at each interior-point iteration is solving the system of linear equations (2.13). Instead, if one uses the saddle-point formulation, computing the matrix triple product \( G^T W_k G \) and solving the system of linear equations (2.15) account for most of the computation. In both cases, the choice of formulation has a similar impact, hence we will only consider the saddle-point approach and we will express the overall complexity considering the cost of the main operations only.

The linear systems solved at each iteration of an active-set method can be derived from those in interior-point methods so the impact of the optimization formulation is alike.

In most first-order methods, the main cost at each iteration is matrix-vector multiplication involving the Hessian \( H \). However, the freedom for choosing an optimization
formulation is severely restricted by the requirement to keep the feasible set simple to allow for efficient computation of projection operations. Still, parts of the discussion in this chapter will also be applicable to first-order methods.

For the sake of notational simplicity, the results of this chapter are presented with reference to the optimal control problem in regulator form, i.e. with $x_{ss} = 0$ and $u_{ss} = 0$. However, all of the results generalize easily to setpoint tracking problems. We also omit reference to slack variables for clarity.

4.2.1 The classic sparse non-condensed formulation

The future states (and slack variables) can be kept as decision variables and the system dynamics can be incorporated into the problem by enforcing equality constraints [184,226,227]. In this case, for $K = 0$, if we let $z := [x^T v^T]^T$, where

$$x := [x_0^T x_1^T \ldots x_N^T]^T, \quad v := [v_0^T v_1^T \ldots v_{N-1}^T]^T,$$

we have $h := 0$, and the remaining matrices have the following sparse structures that describe the control problem (4.2) exactly:

$$H := \begin{bmatrix} I_N \otimes \begin{bmatrix} Q & S \\ S^T & R \end{bmatrix} & 0 \\ 0 & Q_N \end{bmatrix},$$

$$F := \begin{bmatrix} -I_n \\ A_d & B_d & -I_n \\ \vdots \\ A_d & B_d & -I_n \end{bmatrix}, \quad f := \begin{bmatrix} -x \\ 0 \\ \vdots \\ 0 \end{bmatrix},$$

$$G := \begin{bmatrix} J & E_0 \\ E^- & J & E \\ \vdots \\ E^- & J & E \\ J_N \end{bmatrix}, \quad g := \begin{bmatrix} d \\ \vdots \\ d \\ d_N \end{bmatrix},$$

where $\otimes$ denotes a Kronecker product. If there are no input-rate constraints or the state is augmented to reformulate the problem in terms of input rates, $E^- = 0$.

Observe that this formulation is suitable for time-varying and nonlinear MPC applications, since matrices $H, F, f, G$ and $g$ do not have to be recomputed, just overwritten.

Assuming general constraints, the number of floating-point operations (flops) for computing $G^T W_k G$ is approximately $N l (n_x + n_u)^2$, where $l$ is the dimension of vector $d$. For solving the system of linear equations, the coefficient matrix, say $A_k \in \mathbb{R}^{N(2n_x+n_u) \times N(2n_x+n_u)}$,
is an indefinite symmetric matrix that can be made banded through appropriate row re-ordering (or interleaving of primal variables and Lagrange multipliers). The resulting banded matrix has a half-band of size $2n_x + n_u$. Such a linear system can be solved using a banded \( LDL^T \) factorization in \( N(2n_x + n_u)^3 + 4N(2n_x + n_u)^2 + N(2n_x + n_u) \) flops [25, App. C], or through a block factorization method based on a sequence of Cholesky factorisations in \( \mathcal{O}(N(n_x + n_u)^3) \) operations [184].

It is also worth considering the memory requirements of each formulation since it is an important aspect for embedded implementations [106]. The memory requirements can be approximated by the cost of storing matrices \( H, G, F \) and \( A_k \), which are all sparse and require approximately \( \frac{1}{2}N(n_x + n_u)^2 \), \( N\ell(n_x + n_u) \), \( Nn_x(n_x + n_u) \) and \( N(2n_x + n_u)^2 \) elements, respectively. For time-invariant problems, these matrices mostly consist of repeated blocks.

### 4.2.2 The classic dense condensed formulation

The state variables can be eliminated from the optimization problem by expressing them as an explicit function of the current state and the controlled variables [139]:

\[
x = Ax + Bv, \tag{4.7}
\]

where \( A_K := A_d + B_dK \) and

\[
\begin{bmatrix}
I_n \\
A_K \\
A_K^2 \\
\vdots \\
A_K^{N-1} \\
A_K^N
\end{bmatrix}
\begin{bmatrix}
0 \\
B_d \\
A_K B_d \\
\vdots \\
A_K^{N-2} B_d \\
A_K^{N-1} B_d
\end{bmatrix}
\begin{bmatrix}
B_d \\
\cdot \\
\cdot \\
\cdot \\
\cdot \\
0
\end{bmatrix}.
\tag{4.8}
\]

In this case, if we let \( z := v, F := 0, f := 0 \), then we have an inequality constrained QP with

\[
H := B^T(Q + K^T R K + S K + K^T S^T)B + R + B^T(K^T R + S) + (R K + S^T)B,
\]

\[
h := x^T A^T(QB + S(KB + I) + K^T(R(KB + I) + S^T B)),
\]

\[
G := (J + EK)B + E,
\]

\[
g := d - (J + EK)Ax,
\]

where

\[
Q := \begin{bmatrix} I_N \otimes Q & 0 \\ 0 & Q_N \end{bmatrix}, \quad S := \begin{bmatrix} I_N \otimes S \\ 0 \end{bmatrix}, \quad R := I_N \otimes R,
\]

\[
K := \begin{bmatrix} I_N \otimes K & 0 \end{bmatrix}, \quad J := \begin{bmatrix} I_N \otimes J & 0 \\ 0 & J_N \end{bmatrix}, \quad d := \begin{bmatrix} 1_N \otimes d \\ d_N \end{bmatrix},
\]

66
\[ E := \begin{bmatrix} E_0 & \mathbf{E} & \mathbf{E} & \cdots & \mathbf{E} \\ E^- & E & \cdots & \mathbf{E} \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ \mathbf{E} & \mathbf{E} & \cdots & E^- & \mathbf{E} \end{bmatrix}. \]

Observe that vectors \( h \) and \( g \) have to be recomputed for every state measurement \( x \). For time-varying and nonlinear MPC applications, matrices \( H \) and \( G \) also have to be recomputed periodically, adding a considerable computational overhead.

When \( K = 0 \) (\( u_k = v_k \)) or is an arbitrary stabilizing gain \([196]\), \( G \) is a lower block Toeplitz triangular matrix. The number of flops required for computing \( G^TW_kG \) can be split into \( \frac{1}{2}N^2ln_u \) operations for the row update \( W_kG \) and \( \frac{1}{2}N^3ln_u^2 \) operations for the matrix-matrix multiplication when exploiting the symmetry of the result. In terms of the system of linear equations, in this case \( A_k \in \mathbb{R}^{Nn_u \times Nn_u} \) is a symmetric positive definite dense matrix, hence the problem can be solved using an unstructured Cholesky factorisation in \( \frac{1}{3}N^3n_u^3 + 2N^2n_u^2 \) flops \([25, \text{App. C}]\). The cubic growth in computational requirements with respect to the horizon length, in contrast to the linear growth exhibited by the non-condensed formulation, suggests that the non-condensed approach could be preferable for applications that require long horizons. Furthermore, memory requirements for storing \( H \), \( G \) and \( A_k \) are approximately \( \frac{1}{2}N^2(2n_u^2 + ln_u) \) elements. The quadratic growth with \( N \) arises because matrices are dense and there is no obviously exploitable repetition pattern.

### 4.3 The sparse condensed formulation

This section presents a novel way to formulate the optimal control problem (4.2) as a structured optimization problem.

We will use the following definitions:

**Definition 1** (Controllability index). The smallest number of time steps to drive the system from any \( x \in \mathbb{R}^{n_x} \) to the origin. It is finite if the system \((A_d, B_d)\) is controllable.

**Definition 2** (Nilpotency index). The smallest integer \( r \) such that that \( A^i = 0 \) for all \( i \geq r \) when \( A \) is a nilpotent matrix.

The following proposition summarizes the method to introduce sparsity into the otherwise dense condensed optimization formulation by making use of the variable transformation (4.3c). It is important to clarify that this \( K \) does not have to be the same as the feedback gain being assumed from \( k = N \) to infinity \([202]\), hence stability and feasibility properties are independent of the choice of \( K \). In this context, the effect of the change of variables is not plant pre-stabilization but a mathematical trick to introduce structure into the problem. The gain \( K \) is never implemented in practice.
Proposition 1. If the pair \((A_d, B_d)\) is controllable, we can choose \(K\) such that \(A_K\) is a nilpotent matrix with nilpotency index \(r\) so that when \(N > r + 1\) the prediction matrix \(B\) in (4.8) is block Toeplitz, block banded lower triangular with a halfband of \((r+1)n_x\) elements. The last \((N - r + 1)n_x\) rows of \(A\) are also zero.

Proof. Given a reachable system \((A_d, B_d)\) there exists a feedback law such that the closed-loop dynamics matrix has arbitrary eigenvalues [1]. The problem of obtaining a suitable matrix \(K\) such that \(A_d + B_d K\) has all eigenvalues at zero is analogous to finding a deadbeat gain in the context of static state feedback. A numerically reliable way of computing a deadbeat feedback gain in the multi-input case is not a trivial task, but the problem has been addressed by several authors [51, 58, 207]. These methods start by transforming the original system into the controllability staircase form \([\text{ctrbf} \text{ in Matlab}]\), which unlike the controller canonical form, can be obtained through well-conditioned unitary transformations. The transformed system is given by

\[
\begin{bmatrix}
\tilde{x}_{r+1}^r \\
\tilde{x}_{u+1}^u
\end{bmatrix} = \begin{bmatrix}
A_r & A_ru \\
0 & A_u
\end{bmatrix} \begin{bmatrix}
\tilde{x}_r^r \\
\tilde{x}_u^u
\end{bmatrix} + \begin{bmatrix}
B_r \\
0
\end{bmatrix} u_k,
\]

where the subscripts \(r\) and \(u\) refer to the reachable and unreachable subspaces, respectively, and the matrix \(A_r\) is in staircase form with a number of steps equal to the controllability index of the reachable subsystem \((A_r, B_r)\). These methods yield the minimum nilpotency index for \(A_d + B_d K\), which is equal to the controllability index of \((A_d, B_d)\) given by

\[
r := \left\lceil \frac{n_x}{\text{rank}(B_r)} \right\rceil + r_u,
\]

where \(r_u\) is the nilpotency index of the unreachable subsystem \(A_u\). The structure of \(A\) and \(B\) is clear from direct inspection of (4.8).

Corollary 1. If \(K\) is chosen such that \(A_K\) is nilpotent, then matrices \(H\) and \(G\) are banded, the size of their non-zero bands is independent of \(N\), and each interior-point iteration has a complexity linear with respect to \(N\).

Proof. \(XB\) yields a matrix with the same structure as \(B\) when \(X\) is block-diagonal, and \(B^T XB\) yields a symmetric banded matrix with halfband equal to the halfband of \(B\).

\(H\) is now a block banded symmetric positive definite matrix of size \(N n_u \times N n_u\) with half-band equal to \(r + 1\) blocks of size \(n_u \times n_u\). In the time-invariant case, there are only

68
\( r + 1 + \frac{r(r+1)}{2} \) distinct blocks and its structure is given by

\[
H := \begin{bmatrix}
H_1 & H_2 & \cdots & H_{r+1} & 0 & \cdots & 0 \\
H_2^T & H_1 & \cdots & \cdots & \vdots \\
\vdots & \vdots & \ddots & \ddots & \vdots \\
H_{r+1}^T & \cdots & \cdots & \cdots & \cdots & \cdots & 0 \\
0 & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\
H_1 & H_2 & \cdots & H_{r+1} & H_2^T & \tilde{H}_{1,1} & \cdots & \tilde{H}_{1,r} \\
\vdots & \vdots & \ddots & \ddots & \vdots & \vdots & \ddots & \vdots \\
0 & \cdots & \cdots & 0 & H_{r+1}^T & \tilde{H}_{1,r}^T & \cdots & \tilde{H}_{r,r}
\end{bmatrix}
\]

where

\[
X := Q + K^T R K + S K + K^T S^T,
\]

\[
H_1 := R + \sum_{i=1}^{r} (A_i^{-1} B_d)^T X A_i^{-1} B_d,
\]

\[
H_j := (A_K^{j-2} B_d)^T (K^T R + S) + \sum_{i=j-1}^{r-1} (A_K^{i-1} B_d)^T X A_K^{i-1} B_d,
\]

for \( j = 2, \ldots, r + 1 \),

\[
\tilde{H}_{k,k} := R + \sum_{i=1}^{r-k} (A_K^{i-1} B_d)^T X A_K^{i-1} B_d + (A_K^{r-k} B_d)^T Q_N A_K^{r-k} B_d,
\]

for \( k = 1, \ldots, r \),

\[
\tilde{H}_{k,k+j} := (A_K^{i-1} B_d)^T (K^T R + S) + \sum_{i=j}^{r-k-1} (A_K^{i-1} B_d)^T X A_K^{i-1} B_d + (A_K^{r-k} B_d)^T Q_N A_K^{r-k-j} B_d,
\]

for \( j = 1, \ldots, r - 1 \) and \( k = 1, \ldots, r - j \).

The situation is similar for \( G \) and \( A_k \). \( G \) is a block Toeplitz, block banded lower triangular matrix with a half-band of \( r + 1 \) blocks of size \( l \times n_u \). The number of flops required for computing \( G^T W_k G \) is approximately \( \frac{1}{2} N n_u (r + 1) l \) for the row update plus \( \frac{1}{2} N n_u^2 (r + 1)^2 l \) for the matrix multiplication. The coefficient matrix \( A_k \in \mathbb{R}^{Nn_u \times Nn_u} \) is now a symmetric positive definite banded matrix with the same size and structure as \( H \), hence the linear system can be solved using a banded Cholesky routine with a cost of \( N n_u^2 (r + 1)^2 + 4 N n_u^2 (r + 1) \) flops [25]. Memory requirements grow linearly with \( N \) and can be reduced significantly by exploiting repetition in the time-invariant case, as described above.
Table 4.1: Comparison of the computational complexity imposed by the different QP formulations.

<table>
<thead>
<tr>
<th></th>
<th>Computation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Condensed</td>
<td>$\mathcal{O}(N^3n_u(l + n_u))$</td>
</tr>
<tr>
<td>Non-condensed</td>
<td>$\mathcal{O}(N(n_u + n_x)^2(l + n_u + n_x))$</td>
</tr>
<tr>
<td>Sparse condensed</td>
<td>$\mathcal{O}(Nn_u^2r^2(l + n_u))$</td>
</tr>
</tbody>
</table>

Table 4.2: Comparison of the memory requirements imposed by the different QP formulations.

<table>
<thead>
<tr>
<th></th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>Condensed</td>
<td>$\mathcal{O}(N^2n_u(l + n_u))$</td>
</tr>
<tr>
<td>Non-condensed</td>
<td>$\mathcal{O}(N(n_u + n_x)(l + n_u + n_x))$</td>
</tr>
<tr>
<td>Sparse condensed</td>
<td>$\mathcal{O}(Nn_u(l + n_u))$</td>
</tr>
</tbody>
</table>

4.3.1 Comparison with existing formulations

Tables 4.1 and 4.2 compare the upper bound computational complexity and memory requirements for the three different QP formulations that have been discussed in this chapter. The expressions for the sparse condensed approach assume that $N > r + 1$, otherwise the matrices are dense. Hence, the sparse condensed approach is always at least as fast as the standard condensed approach in terms of computational complexity and memory requirements. Taking a conservative assumption for the largest possible nilpotency index $r = n$, the expressions suggest that if the number of states is larger than the number of inputs the new formulation presented in this section will provide an improvement over the non-condensed approach both in terms of computation and memory usage. Both these approaches will outperform the standard condensed approach for $N$ large. These predictions are confirmed by Figure 4.1.

The flop count of an algorithm is proportional to the computational effort required, but the computational time will largely depend on the specific implementation and computing platform. The operations to be carried out using the sparse condensed approach are all banded linear algebra for which efficient software libraries exist and efficient hardware implementations are possible [138], hence we do not consider this to be a limiting factor.

Being able to directly apply Cholesky instead of an indefinite factorization is another benefit over the non-condensed approach. Cholesky factorization is more numerically stable than $LDL^T$, it requires slightly less computation, and the possibility of choosing an arbitrary permutation matrix allows for a simpler pivoting procedure and the possibility of making use of the block structure inside the non-zero band to reduce computation and memory requirements further. Additional benefits over the non-condensed approach come from the possibility of adding input rate constraints to the optimal control problem (4.2)
Figure 4.1: Accurate count of the number of floating point operations per interior-point iteration for the different QP formulations discussed in this chapter. The size of the control problem is $n_u = 2$, $n_x = 6$, $l = 6$ and $r = 3$.

without increasing the state dimension and without affecting the structure of the matrices in the optimization problem (4.6). With the non-condensed approach the inclusion of rate constraints without augmenting the state increases the bandsize of $G$ and consequently the bandsize of $A_k$.

4.3.2 Limitations of the sparse condensed approach

A new $K$ needs to be computed for different $(A_d,B_d)$ pairs; however, the complexity of the procedure in [207] is $O(n_x^2 + \text{rank}(B_r)^2 n_x)$, hence the approach could still be applicable to some online time-varying and nonlinear MPC applications. For LTI systems this computation is carried out offline. A limitation for first-order methods comes from the use of variable transformation (4.3c) changing the geometry of the feasible set, which could render the projection problem in first-order methods as hard as solving the original QP.

A further potential drawback affects the numerical conditioning. It is well-known that control and signal processing problems can be ill-conditioned when there is a large mismatch between the requested sampling frequency and the dynamics of the continuous-time time system [80]. The method presented in this section is no exception to the rule, however, the conditioning is acceptable for most control problems, especially the type that we are targeting. The matrix $K$ is the deadbeat feedback gain that can control any state to the origin in $r$ steps. For systems with fast dynamics it is easier to steer the state
quickly, hence relatively small values of $K$ are necessary and the conditioning of the problem is acceptable. It is precisely applications with fast dynamics that can benefit most from methods for solving optimization problems faster that can turn the possibility of employing MPC into a feasible option.

A well-known drawback of deadbeat control is that if the sampling period is small with respect to the system’s dynamics, a very large $K$ could be necessary as more energy would be needed to steer the state to zero in less time. In the context of this paper, a large value of $K$ can result in an ill-conditioned optimization problem. Our numerical simulations have indeed confirmed that for systems with slow unstable dynamics, when sampling in the millisecond range using the sparse condensed approach, the QP problems become badly conditioned and the performance of the closed-loop system is unsatisfactory. However, for plants with stable dynamics, the control quality is good at most sampling frequency regimes, hence the problem could be solved by using a pre-stabilising gain.

4.4 Numerical results

We start by describing a widely studied benchmark example consisting of a set of oscillating masses connected by springs and dampers and attached to walls [114, 222], as illustrated by Figure 4.2. The system has $n_u$ control inputs – a maximum of one for each mass, and two states for each mass, its position and velocity. The goal of the controller is to track a reference for the position of each mass while satisfying the system limits.

For the simulations in this chapter the system consists of six masses, all of which can be actuated. The control inputs and mass positions are constrained in the ranges $[-0.5, 0.5]$ and $[-4, 4]$, respectively, hence the control problem has $n_u = 6$, $n_x = 12$ and $l = 24$. The horizon length is 0.2 seconds and the number of steps is given by $N = \lceil \frac{0.2}{T_s} \rceil$ where $T_s$ is the sampling period, i.e. if the sampling frequency increases the problem dimension also increases. The masses, spring constants and damping coefficients are 0.1kg, 150Nm$^{-1}$ and 0.01kgs$^{-1}$, respectively, resulting in a controllable plant with controllability index $r = 2$ and a maximum frequency pole at 36Hz. The matrices $Q$, $R$ and $S$ are obtained from continuous-time matrices assuming a zero-order hold. These are chosen such that the inputs and mass positions are penalized equally. The control objective in this case is to keep all masses at their rest position, i.e. $x_{ss} = 0$ and $u_{ss} = 0$.

All simulations start with all masses at their rest position, except mass 6 which is displaced to the constraint. This starting condition guarantees that input constraints will become active during the simulations, which last for 6 seconds. Sampling faster
leads to better quality control for all formulations as the controller is able to respond faster to uncertainty. However, this also means that the number of steps in the horizon increases, so the amount of computation required also increases. Figure 4.3 shows the trade-off between closed-loop control quality and computational requirements for all QP formulations described in this chapter. The plot is generated by computing the closed-loop cost and computational demands for a range of sampling frequencies. For a given control quality the new proposed approach requires less computation than with the existing formulations, and for a fixed computational power the proposed approach achieves a better control quality because it allows for faster sampling.

4.5 Other alternative formulations

Recently, a new formulation has appeared in the literature [142] that eliminates the control inputs from the optimization problem by rewriting the constraint (4.3b) in the form

\[ u_k = B_d^\dagger (x_{k+1} - A_dx_k), \quad k = 0, 1, \ldots, N - 1, \]

or when also considering (4.3c) as

\[ v_k = B_d^\dagger (x_{k+1} - (A_d + B_dK)x_k), \quad k = 0, 1, \ldots, N - 1, \]

where \( B_d^\dagger \) is the Moore-Penrose pseudo-inverse [178] when \( n_x > n_u \) and \( B_d \) is full column rank.
In this case, if we let $z := x$, we obtain an equality and inequality constrained QP with banded matrices. When forming the linear system, the coefficient matrix $A_k \in \mathbb{R}^{2(N+1)n_x \times 2(N+1)n_x}$ is an indefinite symmetric matrix that can be made banded again through appropriate interleaving of primal variables and Lagrange multipliers. This matrix has a halfband of $3n_x - n_u$ and it can be solved using a banded $LDL^T$ routine with cost growing asymptotically as $O(Nn_x^3)$.

Compared to the non-condensed approach, the size of the linear system is smaller and the halfband is also smaller when $n_x < 2n_u$, so this new formulation could provide a reduction in computational complexity for certain control problems. However, no numerical results have yet been presented and the impact on the conditioning of the optimization problem remains unclear. In addition, it is not clear how one would handle input-rate and soft state constraints under this formulation.

4.6 Summary and open questions

In this chapter, we have presented a novel way to formulate a constrained optimal control problem as a structured optimization problem that can be solved in a time that is linear in the horizon length with an interior-point method. The structure is introduced through a suitable change of variables that results in banded prediction matrices. The proposed method has been compared against the current standard approaches and it has been shown to offer reduced computational and memory requirements for most control problems. As a result, employing the proposed approach could allow one to push the boundaries of MPC to allow implementation in applications where the computational burden has so far been too great, or it could allow current MPC applications to run on cheaper commodity hardware.

The limitations of the approach have also been identified. Existing algorithms for computing the deadbeat feedback gain $K$ attempt to find the minimum nilpotency index $r$, because the goal of the feedback is to provide a closed-loop system that steers the state to zero in the least number of steps. In the context of this chapter, where deadbeat control is used as a mathematical trick to introduce structure into the optimization problem, a smaller value of $r$ results in smaller non-zero bands in the matrices. However, in order to improve the numerical conditioning of the problem, in some circumstances it may be preferable to increase the nilpotency index beyond the controllability index of the plant, especially since any $r$ smaller than $N - 1$ provides an improvement over the standard condensed approach\(^1\). A methodology that allows trading computational time and memory requirements for numerical conditioning of the resulting optimization problem could be a target for future research.

\(^1\)Note that $r \leq n_x$ will always hold.
5 Hardware Acceleration of Floating-Point Interior-Point Solvers

For a broad range of control applications that could benefit from employing predictive control, the cost and power requirements of the general purpose computing platforms necessary to meet hard real-time requirements are unfavourable. Alternative and more efficient computational methods could enable the use of MPC to be extended. In this chapter the focus is on custom circuits designed specifically for interior-point methods for predictive control. Interior-point methods can handle different classes of MPC problems and their performance is generally equally reliable regardless of the type of constraints or condition number of the problem. However, the large numerical dynamic range exhibited in most variables of the algorithm imposes the use of floating-point arithmetic. FPGAs are especially well suited to this application due to the large amount of computation for a small amount of I/O. In addition, unlike a general-purpose implementation, an FPGA can provide the precise timing guarantees required for interfacing the controller to the physical system. Unlike many embedded microprocessors, an FPGA does not preclude the use of standard double precision arithmetic. However, single precision arithmetic is used to reduce the number of hardware resources required, since this significantly reduces the size, cost and power consumption of the FPGA device needed to realise the design.

In recent years there have been several FPGA implementations of interior-point solvers for control. Most of the proposed case studies have been for low-dimensional systems — a domain in which explicit MPC could also potentially be an option. In addition, all the implementations to date employ the condensed MPC formulation, which is simpler to implement due to the dense nature of the matrix operations but has significant disadvantages, especially for medium to large problems (refer to Chapter 4). Besides, in most prior work the focus has been on demonstrating feasibility while no comparison with respect to state-of-the-art solvers for general-purpose platforms has been attempted.

The parameterisable hardware architecture, for solving sparse non-condensed MPC problems, presented in this paper has the objective of maximising throughput. As a consequence, the design has several interesting characteristics that will be discussed further in Chapter 7. Most of the acceleration is achieved through a parallel implementation of the minimum residual (MINRES) algorithm used to solve the system of linear equations occurring at each iteration of the interior-point algorithm. The implementation yields more than one order of magnitude improvements in solution times compared to a software implementation of the same algorithm running on a desktop platform. It is also shown
that by considering that the QPs come from a control formulation, it is possible to make heavy use of the sparsity and structure in the problem to save computations and reduce memory requirements by 75%.

The proposed architecture is evaluated with a detailed case study on the application of FPGA-based MPC for the control of a large airliner. Whilst arguments for the use of MPC in flight control can be found in [57, 70, 140], the focus here is on the FPGA-based methodology for implementation of the optimisation scheme rather than tuning controller parameters or obtaining formal certificates of stability for which a mature body of theory is already available. This case study considers a significantly larger plant model than all prior FPGA-based implementations, and since this plant model is open-loop unstable, is more numerically challenging. Furthermore, a complete system-on-a-chip implementation is presented where the target calculator and observer are also implemented on-chip and the data transfers with the outside world are handled by a Xilinx MicroBlaze soft-core processor [235]. It should be noted that in contrast to [20, 220, 240] where the custom circuit is used as an accelerator for parts of the QP solver with the rest implemented in software on a conventional processor, the present design uses the MicroBlaze solely as a means of bridging communication, and this could be replaced with a custom interface layer to suit the demands of a given application. This implementation is also capable of running reliably at higher clock rates than prior designs.

To demonstrate the flexibility in the trade-off between control performance and solution time, a numerical study is performed to investigate the nature of the compromise between the number of iterations of the inner MINRES algorithm, and the effect of offline model scaling and online matrix preconditioning, all of which directly influence the total solution time and the solution quality, both in terms of fidelity of the computed control input with respect to that obtained from a standard QP solver, and in terms of the resulting closed loop control performance. Despite using single precision arithmetic, the proposed design using offline scaling and online preconditioning, running on an FPGA with the circuit clocked at 250 MHz compares favourably in terms of solution quality and latency to more commonly used matrix factorisation-based algorithms implemented in double precision arithmetic running on a conventional PC at gigahertz clock frequencies.

Outline

The chapter starts by justifying the choice of optimization algorithm over other alternatives for solving QPs in Section 5.1. Previous attempts at implementing optimization solvers in hardware are examined and compared with the proposed approach in Section 5.2. A brief analysis of the complexity of the different stages of the chosen interior-point algorithm is presented in 5.3 to set the scene for the detailed analysis of the parameterisable hardware architecture in Section 5.4. General performance results are presented in Section 5.5 and Section 5.6 presents the detailed airliner case study that includes a numerical investigation. Finally, Section 5.7 summarises open questions in this area. Table 5.10, at the end of the chapter, includes a list of symbols for easy reference.
5.1 Algorithm choice

Different factors can motivate the choice of algorithm for a custom hardware design in comparison to those important for a software implementation. In sequential software, a smaller flop count leads to shorter algorithm runtimes in the absence of cache effects. In hardware it is the ratio of parallelisable work to sequential work that determines the potential speed of an implementation. Furthermore, the proportion of different types of operations can also be an important factor. Multiplication and addition have lower latency and use fewer hardware resources than division or square root operations. All of these aspects, often unfamiliar to the software or application engineer play an important role in hardware design.

Modern methods for solving QPs, which involve solving systems of linear equations, can be classified into interior-point or active-set methods, each exhibiting different properties that make them suitable for different purposes. The worst-case complexity of active-set methods increases exponentially with the problem size, often leading to a large variance in the number of iterations needed to achieve a certain accuracy. In embedded control applications there is a need for guarantees on real-time computation, hence the polynomial complexity and more predictable execution time exhibited by interior-point methods is a more attractive feature. In addition, the size of the linear systems that need to be solved at each iteration in an active-set method changes depending on which constraints are active at any given time. In a hardware implementation, this is problematic since all iterations need to be executed on the same fixed architecture. Interior-point methods are a better option for our needs because they maintain a constant predictable structure, which is easily exploited.

Logarithmic-barrier [25] and primal-dual [228] are two competing interior-point methods. From the implementation point of view, a difference to consider is that the logarithmic-barrier method requires an initial feasible point with respect to the inequality constraints (4.3d)-(4.3e) and the method fails if an intermediate solution falls outside of the feasible region. In infinite precision this is not a problem, since both methods stay in the interior of the feasible region provided they start inside it. In a real implementation, finite precision effects may lead to infeasible iterates, so in that sense the primal-dual method is more robust. Moreover, with infeasible primal-dual interior-point methods [228] there is no need to implement a Phase I procedure [25], which would require additional hardware, to initialize the algorithm with a feasible point.

Mehrotra’s primal-dual algorithm [148] has proven very efficient in software implementations. The algorithm solves two systems of linear equations with the same coefficient matrix at each iteration, thereby reducing the overall number of iterations. However, the benefits can only be attained by using factorization-based methods for solving linear systems, since the factorization can be computed only once and reused for different right-hand sides. Previous work [21,136] suggests that iterative linear solvers can be preferable over direct (factorisation-based) methods in this context, despite the problem sizes being
small in comparison to the problems for which these methods have been used historically.
Firstly, matrix-vector multiplication accounts for most of the computation at each iteration, an operation offering multiple parallelisation opportunities. Secondly, there are few division and square root operations compared to factorisation-based methods. Finally, these methods allow one to trade off accuracy for computational time by varying the number of iterations. In fact, a relatively small number of iterations can be sufficient to obtain adequate accuracies in many cases, as shown in the airliner case study in Section 5.6.

Because these methods do not allow one to amortise work when solving for different right hand sides, a simple primal-dual interior-point algorithm [226], where a single system of equations is solved per iteration, is employed instead of Mehrotra’s predictor-corrector algorithm [148], which is found in most software packages (e.g. the state-of-the-art code generation tools CVXGEN [146] and FORCES [49] for embedded interior-point solvers customised to specific problem structures). Specifically, we employ Algorithm 1 to solve a non-condensed QP in the form (4.6). The primal-dual interior-point algorithm uses Newton’s method [25] for solving the nonlinear KKT optimality conditions (2.8)-(2.9). The method solves a sequence of related linear problems. At each iteration, three tasks need to be performed: linearisation around the current point (Line 2), solving the resulting saddle-point linear system to obtain a search direction (Line 3), and performing a line search to update the solution to a new point (Line 6). A standard backtracking line search algorithm is used, with the backtracking parameter set to 0.5 and a maximum of 20 line search iterations. For more details on the derivation of the algorithm, refer to Section 2.2.1. Rather than checking a termination criterion, the number of interior-point iterations is fixed a priori since that would be the preferred practice in a deterministic real-time environment. A detailed investigation into the number of iterations needed by interior-point methods is not the subject of this thesis.

In order to accelerate the convergence of the iterative linear solver for Line 3 in Algorithm 1, it is sometimes necessary to employ a preconditioner. The hardware architecture provides support for diagonal preconditioning, i.e. instead of solving the linear system $A_k \xi_k = b_k$, we solve

$$M_k A_k M_k y_k = M_k b_k$$

$$\Leftrightarrow \tilde{A}_k y_k = \tilde{b}_k,$$

where $M_k$ is a diagonal matrix with positive entries computed at each iteration. The solution to the original problem is recovered by computing $\xi_k = M_k y_k$. The hardware implementation is outlined in Section 5.4.4 and the numerical effect of a particular preconditioner on the airliner case study is described in Section 5.6.
Algorithm 1 Primal dual interior point algorithm.

Require: \( z_0 = 0.05, \nu_0 = 0.3, \lambda_0 = 1.5, s_0 = 1.5, \sigma = 0.35. \)

1: \( \text{for } k = 0 \text{ to } I_{IP} - 1 \text{ do} \)

2: Linearization \( \mathcal{A}_k := \hat{\mathcal{A}} + \begin{bmatrix} \Phi_k & 0 \\ 0 & 0 \end{bmatrix}, \quad b_k := \begin{bmatrix} r^z_k \\ r^\nu_k \end{bmatrix} \) where

\[
\hat{\mathcal{A}} := \begin{bmatrix} H & F^T \\ F & 0 \end{bmatrix}, \quad \Phi_k := G^T W_k^{-1} G, \quad W_k^{-1} := \Lambda_k S_k^{-1}
\]

\[
r^z_k := -\Phi_k z_k - h - F^T \nu_k - G^T (\lambda_k - \Lambda_k S_k^{-1} g + \sigma \mu_k s_k^{-1}),
\]

\[
r^\nu_k := -F z_k + f,
\]

\[
\mu_k := \frac{\lambda_k^T s_k}{|I|} \quad \text{as defined in (2.11)}.
\]

3: Solve \( \mathcal{A}_k \xi_k = b_k \) for \( \xi_k := \begin{bmatrix} \Delta z_k \\ \Delta \nu_k \end{bmatrix} \)

4: \( \Delta \lambda_k := \Lambda_k S_k^{-1} (G(z_k + \Delta z_k) - g) + \sigma \mu_k s_k^{-1} \)

5: \( \Delta s_k := -s_k - (G(z_k + \Delta z_k) - g) \)

6: Line Search \( \alpha_k := \max_{0 \leq \alpha \leq 1} \alpha : \begin{bmatrix} \lambda_k + \alpha \Delta \lambda_k \\ s_k + \alpha \Delta s_k \end{bmatrix} > 0 \)

7: \( (z_{k+1}, \nu_{k+1}, \lambda_{k+1}, s_{k+1}) = (z_k, \nu_k, \lambda_k, s_k) + \alpha_k (\Delta z_k, \Delta \nu_k, \Delta \lambda_k, \Delta s_k) \)

8: \text{end for}

5.2 Related work

There have been several previous FPGA implementations of QP solvers for predictive control. The suitability of each method for FPGA implementation was studied in [120] with a sequential implementation, highlighting the advantages of interior-point methods for larger problems. Occasional numerical instability was also reported, having a greater effect on active-set methods.

A first hardware implementation of explicit MPC, based on parametric programming, was described in [109] and since then there have been many works focusing on this problem, e.g. [35, 179]. Explicit MPC is naturally less vulnerable to reduced precision effects, and can achieve high performance for small problems, with sampling intervals on the order of microseconds being reported in [109]. However, the memory and computational requirements typically grow exponentially with the problem dimension, making the scheme unattractive for handling larger problems. For instance, a problem with six states, two inputs, and two steps in the horizon required 63 MB of on-chip memory in [109], whereas our implementation would require less than 1 MB. In this thesis we only consider online numerical optimization, thereby addressing problems with more than four states.

The challenge of accelerating linear programs (LPs) on FPGAs was addressed in [12] and [125]. [12] proposed a deeply pipelined architecture based on the Simplex method. Speed-ups of around 20x were reported over state-of-the-art LP software solvers, although the method suffers from active-set pathologies when operating on large problems. Acceleration of collision detection in graphics processing was targeted in [125] with an interior-
point implementation based on Mehrotra’s algorithm [148] using single-precision floating point arithmetic. The resulting optimization problems were small; the implementation in [125] solves linear systems of order five at each iteration.

In terms of hardware QP solver implementations, as far as the author is aware, all previous work has also targeted MPC applications. The feasibility of implementing QP solvers for MPC applications on FPGAs was demonstrated in [132] with a sequential Handel-C implementation. The design was revised in [131] with a fixed-area design that exploits modest levels of parallelism in the interior-point method to approximately halve the clock cycle count. The implementation was shown to be able to respond to disturbances and achieve sampling periods comparable to stand-alone MATLAB executables for a constrained aircraft example with four states, one input, and three steps in the horizon. A comparison of the reported performance with the performance achieved by our design on a problem of the same size is given in Table 5.1. In terms of scalability, the performance becomes significantly worse than the MATLAB implementation as the size of the optimization problem grows. This could be a consequence of solving systems of linear equations using Gaussian elimination, which can be inefficient for handling large matrices. In contrast, our circuit becomes more efficient as the size of the optimization problem grows (refer to Section 5.4).

A design consisting of a soft-core (sequential) processor attached to a co-processor used to accelerate computations that allowed data reuse was presented in [115], addressing the implementation of MPC on very resource-constrained embedded systems. The emphasis was on minimizing the resource usage and power consumption. Again, a soft-core processor was used in [32] to execute a C implementation of the QP solver and demonstrate the performance on a two-state drive-by-wire system. In [20, 220, 240], a mixed software/hardware implementation is used where the core matrix computations are carried out in parallel custom hardware, whilst the remaining operations are implemented in a general purpose microprocessor. The performance was evaluated on two-state systems. In contrast, in [240] the numerically intensive linear solvers were implemented in software while custom accelerators were used for the remaining operations. In this case, a motor servo system with two states was used as a case study. The use of non-standard number

<table>
<thead>
<tr>
<th>Ref.</th>
<th>Example</th>
<th>Original Implementation</th>
<th>Our Implementation Latency</th>
<th>Throughput</th>
</tr>
</thead>
<tbody>
<tr>
<td>[131]</td>
<td>Citation Aircraft</td>
<td>330µs</td>
<td>185µs</td>
<td>8.4µs</td>
</tr>
<tr>
<td>[220]</td>
<td>Rotating Antenna</td>
<td>450µs</td>
<td>85µs</td>
<td>2.5µs</td>
</tr>
<tr>
<td>[220]</td>
<td>Glucose Regulation</td>
<td>172µs</td>
<td>60µs</td>
<td>1.4µs</td>
</tr>
</tbody>
</table>
Table 5.2: Characteristics of existing FPGA-based QP solver implementations

<table>
<thead>
<tr>
<th>Year</th>
<th>Ref.</th>
<th>Number format</th>
<th>Method</th>
<th>QP Form</th>
<th>Design Entry</th>
<th>Implementation Architecture</th>
<th>Clock Freq.</th>
<th>QP size</th>
</tr>
</thead>
<tbody>
<tr>
<td>2008</td>
<td>[131]</td>
<td>float32</td>
<td>PD-IP</td>
<td>D</td>
<td>Handel-C</td>
<td>custom HW</td>
<td>25</td>
<td>3</td>
</tr>
<tr>
<td>2009</td>
<td>[120]</td>
<td>float32</td>
<td>Active Set</td>
<td>D</td>
<td>Handel-C</td>
<td>custom HW</td>
<td>25</td>
<td>3</td>
</tr>
<tr>
<td>2009</td>
<td>[115]</td>
<td>float32</td>
<td>PD-IP</td>
<td>D</td>
<td>C/VHDL</td>
<td>100</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>2009</td>
<td>[113]</td>
<td>float32</td>
<td>Active Set</td>
<td>D</td>
<td>ASIC/FPGA</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>2009</td>
<td>[220]</td>
<td>LNS16</td>
<td>log-barrier IP</td>
<td>D</td>
<td>C/Verilog</td>
<td>50</td>
<td>3</td>
<td>6</td>
</tr>
<tr>
<td>2011</td>
<td>[34]</td>
<td>fixed/float</td>
<td>Hildreth</td>
<td>D</td>
<td>-</td>
<td>custom core</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>2011</td>
<td>[224]</td>
<td>float23</td>
<td>log-barrier IP</td>
<td>D</td>
<td>VHDL</td>
<td>custom HW</td>
<td>70</td>
<td>12</td>
</tr>
<tr>
<td>2012</td>
<td>[240]</td>
<td>float32</td>
<td>Active set</td>
<td>D</td>
<td>C/Verilog</td>
<td>HW/PowerPC</td>
<td>100</td>
<td>3</td>
</tr>
<tr>
<td>2012</td>
<td>[225]</td>
<td>float24</td>
<td>Active set</td>
<td>D</td>
<td>VHDL</td>
<td>custom HW</td>
<td>70</td>
<td>12</td>
</tr>
<tr>
<td>2012</td>
<td>[150]</td>
<td>float18</td>
<td>log-barrier IP</td>
<td>D</td>
<td>VHDL</td>
<td>custom HW</td>
<td>70</td>
<td>16</td>
</tr>
<tr>
<td>2012</td>
<td>thesis</td>
<td>float32</td>
<td>PD-IP</td>
<td>S</td>
<td>VHDL</td>
<td>custom HW</td>
<td>250</td>
<td>377</td>
</tr>
</tbody>
</table>

D and S denote dense and sparse formulations, respectively, whereas ‘-’ indicates data not reported in publication, and N/A denotes that the field is not applicable. HW denotes hardware. “Soft-core” indicates vendor provided sequential soft processor, whilst “custom core” indicates a user-designed soft processor. Symbols \( n_v \) and \( n_c \) denote the number of decision variables and number of inequality constraints, respectively.

Representations was studied in [34] with a hybrid fixed-point floating-point architecture and a non-standard MPC formulation tested on a satellite example with six states. This trend was followed in [11] with a full fixed-point implementation, although no analysis or guarantees are provided for handling the large dynamic range manifested in interior-point methods. The hardware implementation of MPC for non-linear systems was addressed in [113] with a sequential QP solver. The architecture contained general parallel computational blocks that could be scaled depending on performance requirements. The target system was an inverted pendulum with four states, one input and 60 time steps, however, there were no reported performance results. The trade-off between data word-length, computational speed and quality of the applied control was explored in an experimental manner. Recently, active-set [225] and interior-point [150,224] architectures were proposed by the same authors using (very) reduced precision floating-point arithmetic and solving a condensed QP with impressive computation times, while demonstrating its feasibility on an experimental setup with a 14th order SISO open-loop stable vibrating beam. Most of the proposed case studies for online optimization-based FPGA controllers have been for low-dimensional systems – a domain in which explicit MPC could also potentially be an option.

Table 5.2 summarizes the characteristics of FPGA-based MPC implementations up until 2012, highlighting moderate progress in the last six years. A common trend is the use of dense QP formulations in contrast with the current trends in research for structure-exploiting optimization algorithms for predictive control. The case studies presented in this chapter are orders of magnitude larger and faster than previous implementations.
5.3 Algorithm complexity analysis

In this section the complexity of the different operations at each iteration of Algorithm 1 is analyzed to help design of an efficient custom architecture for this algorithm.

The first task is to compute the coefficients of matrix $A_k$, which, after appropriate interleaving of the primal and dual variables $z$ and $\nu$, has the following structure:

\[
\begin{bmatrix}
-I_{nx} & Q_0 & S_0 & A^T_d & S_1^- \\
S_0^T & R_0 & B_d^T & S_1^- & -I_{nx} \\
A_d & B_d & -I_{nx} & S_1^- \\
S_1^- & -I_{nx} & Q_1 & S_1 & A^T_d \\
S_1^T & R_1 & B_d^T & S_2^- \\
A_d & B_d & -I_{nx} & S_2^- \\
& & & & & \cdots \\
& & & & & \\
& & & & & -I_{nx} & Q_{N-1} & S_{N-1} & A^T_d \\
& & & & & S_{N-1}^T & R_{N-1}^- & B_d^T \\
& & & & & A_d & B_d & -I_{nx} & Q_N 
\end{bmatrix},
\]

where the non-constant blocks that need to be computed are defined as:

\[
Q_N := Q_N + J_N^T W_N J_N, \quad Q_i := Q + J_i^T W_i J, \quad i = 0, 1, \ldots, N - 1,
\]

\[
R_i := R + E_i^T W_i E + (E^\top)^T W_{i+1} E^- \quad i = 0, 1, \ldots, N - 1,
\]

\[
S_i := S + J_i^T W_i E, \quad i = 0, 1, \ldots, N - 1,
\]

\[
S_i^- := (E^-)^T W_{i+1} J, \quad i = 1, \ldots, N - 1,
\]

where $W_i \in \mathbb{R}^{l \times l}$ are diagonal blocks of $W_k$.

The complexity for computing the coefficient matrix depends on the type of constraints. When there are no input-rate constraints or the state has been augmented to handle them, $E^- = 0$, so $S_i^- = 0$ for all $i$. If the constraints are separable in state and input constraints, $J_i^T W_i E$ and $E_i^T W_i J$ are zero, hence $S_i = S$ for all $i$ and does not need to be computed. A common situation is having upper and lower bounds on the inputs and states of the system. In this case, computing the matrix triple products $J_i^T W_i J$ and $E_i^T W_i E$ consists of only $2n_x$ and $2n_u$ additions, respectively. Of course, if there are no state constraints, $Q_i = Q$. Instead, if there are general state constraints, $J_i^T W_i J$ consists of two small matrix row updates plus two small matrix-matrix multiplications.

The coarser structure of $H, F$ and $G$ can also be used when calculating the vectors $r^+_k, r^-_k, \Delta \lambda_k$ and $\Delta s_k$. This leads to having to compute many small matrix-vector multiplications in standard and transposed form. The backtracking line search requires $4I_{ls} |I|$ fairly regular operations, where $I_{ls}$ is the number of allowed line search iterations.

Exploiting the finer matrix structure in a software implementation would involve complex array index arithmetic, possibly resulting in non-coherent memory reads. In a general-purpose processor, this will lead to an increased number of cache misses. Moreover, hav-
ing to perform many small matrix-vector multiplications means that there will be many
transfers of small blocks of data across the memory hierarchy resulting in time overheads.
However, in custom hardware there is a flexible memory subsystem that can be designed
such that data is always available when and where it is needed, improving data locality
and fully avoiding cache misses. Furthermore, if appropriate support is provided, there
is no difference whether matrix data is accessed by row or by column, hence standard
and transposed multiplications with the same matrix are equally efficient, which is not
generally the case in a general-purpose machine.

When solving $A_k \xi_k = b_k$ using an iterative method, most of the computations are asso-
ciated with computing a structured matrix-vector product. This kind of computation can
be carried out efficiently in a microprocessor, especially if the whole matrix can be accom-
modated inside the processor cache, as there will be next to no main memory accesses. In
addition, DSPs and some general-purpose processors include explicit hardware support for
carrying out a multiply-accumulate instruction in one cycle. However, sequential software
cannot take advantage of the easy parallelization opportunities available for this computa-
tion. A GPU’s instruction set architecture is potentially a good match for accelerating
matrix-vector multiplication. However, the lack of independence between additions in a
dot-product calculation limits the speed-up achievable with a GPU architecture when the
size of the matrix, or the number of independent dot-products, is not very large. A custom
datapath can best exploit the dataflow in this computation, allowing wider parallelization
and efficient deep pipelining.

5.4 Hardware architecture

This section describes the main architectural details for the design of the solver for prob-
lem (4.6), which is implemented using VHDL and Xilinx IP-cores for floating point arith-
metic and RAM structures. The implementation is split into two distinct blocks: one
block accelerates solving the linear equations in Line 3 of Algorithm 1 implementing a
parallel MINRES solver; the other block computes all the remaining operations.

5.4.1 Linear solver

Most of the computational complexity in each iteration of the interior-point method is
associated with solving the system of linear equations $A_k \xi_k = b_k$. After appropriate row
re-ordering, matrix $A_k$ becomes banded (5.1) and symmetric but indefinite, i.e. it has
both positive and negative eigenvalues. The size and half-bandwidth of $A_k$ in terms of the
control problem parameters are given respectively by

\[
Z := N(2n_x + n_u) + 2n_x, \quad (5.1a)
\]
\[
M := 2n_x + n_u. \quad (5.1b)
\]
Figure 5.1: Hardware architecture for computing dot-products. It consists of an array of $2M - 1$ parallel multipliers followed by an adder reduction tree of depth $\lceil \log_2(2M - 1) \rceil$. The rest of the operations in a MINRES iteration use dedicated components. Independent memories are used to hold columns of the stored matrix $A_k$ (refer to Section 5.4.3 for more details). $z^{-M}$ denotes a delay of $M$ cycles.

Notice that the number of constraints per stage $l$ does not affect the size of $A_k$, which will be shown to determine the total runtime in certain scenarios. This is another important difference between this design and previous hardware MPC implementations.

The MINRES method is a suitable iterative algorithm for solving linear systems with indefinite symmetric matrices [63]. At each MINRES iteration, a matrix-vector multiplication accounts for the majority of the computations. This kind of operation is easy to parallelize and consists of multiply-accumulate instructions, which are known to map efficiently into hardware in terms of resources.

In [21] the authors propose an FPGA implementation for solving this type of linear systems using the MINRES method, reporting speed-ups of around one order of magnitude over software implementations. Most of the acceleration is achieved through a deeply pipelined dedicated hardware block (shown in Figure 5.1) that parallelizes dot-product operations for computing the matrix-vector multiplication in a row-by-row fashion. We use this architecture in our design with a few modifications to customize it to the special characteristics of the matrices that arise in MPC. Notice that the size of the dot-products that are computed in parallel is independent of the control horizon length $N$ (refer to (5.1b)), thus computational resource usage does not scale with the horizon length.

5.4.2 Sequential block

The remaining operations in the interior-point iteration (Lines 2 and 4–7 in Algorithm 1) are undertaken by a separate hardware block, which we call Stage 1. The resulting two-
Figure 5.2: Proposed two-stage hardware architecture. Solid lines represent data flow and dashed lines represent control signals. Stage 1 performs all computations apart from solving the linear system. The input is the current state measurement $x$ and the output is the next optimal control move $u_0^*(x)$.

stage architecture is shown in Figure 5.2.

Since the linear solver will provide most of the acceleration by consuming most resources it is vital that it remains busy at all times to achieve high computational efficiency. Hence, the parallelism in Stage 1 is chosen to be the smallest possible such that the linear solver is always active. Notice that if both blocks are to be doing useful work at all times, while the linear system for a specific problem is being solved, Stage 1 has to be operating on another independent problem. In Chapter 7, several new MPC algorithms are proposed to make use of this feature.

When computing the coefficient matrix $A_k$, only the diagonal matrix $W_k$ changes from one iteration to the next, thus the complexity of this calculation is small relative to solving linear equations. If the structure of the problem is taken into account, we find that the remaining calculations in an interior-point iteration are all sparse and very simple compared to solving the linear system. Comparing the computational count of all the operations to be carried out in Stage 1 with the latency of the parallel linear solver when running for $Z$ iterations, we come to the conclusion that for most control problems of interest (medium to large problems), the optimum implementation of Stage 1 is sequential, as this will be enough to keep the linear solver busy at all times. This is a consequence of the latency of the linear solver being $\Theta(N^2)$ [21], whereas the number of operations in Stage 1 is only $\Theta(N)$. Since $O(N)$ resources are being used in the linear solver, only a constant (in this case small) amount of resources is needed to balance computation times in both hardware blocks, and thus achieve high computational efficiency.

As a consequence, Stage 1 will be idle most of the time for large problems. This is
Figure 5.3: Floating point unit efficiency of the different blocks in the design and overall circuit efficiency with \( n_u = 3 \), \( N = 20 \), and 20 line search iterations. For one and two states, three and two parallel instances of Stage 1 are required to keep the linear solver active, respectively. The linear solver is assumed to run for \( Z \) iterations.

Indeed the situation observed in Figure 5.3, where we have defined the floating point unit efficiency as

\[
\text{floating point computations per iteration} / \#\text{floating point units} \times \text{cycles per iteration}
\]

For very small problems it is possible that Stage 1 will take longer than solving the linear system. In these cases, in order to avoid having the linear solver idle, another instance of Stage 1 is synthesized to operate in parallel with the original instance and share the same control block. For large problems, only one instance of Stage 1 is required. The efficiency of the circuit increases as the problems become larger as a result of the dot-product block, which is always active by design, consuming a greater portion of the overall resources.

**Datapath**

The computational block performs any of the main arithmetic operations: addition, subtraction, multiplication and division. Xilinx Core Generator [230] was used to generate highly optimized single-precision floating point units with maximum latency to achieve a high clock frequency. Extra registers were added after the multiplier to match the latency of the adder for synchronization, as these are the most common operations. The latency of the divider is much larger (27 cycles) than the adder (12 cycles) and the multiplier (8 cycles), therefore it was decided not to match the delay of the divider path, as it would
increase the length of the execution pipeline and will reduce our flexibility for ordering computations. Idle instructions were inserted whenever division operations were needed, namely only when calculating $W_k$ and $s_k^{-1}$.

Comparison operations are also required for the line search method (Line 6 of Algorithm 1), however this is implemented by repeated comparison with zero, so only the sign bit needs to be checked and a full floating-point comparator is not needed.

The total number of floating point units in the circuit is given by Table 5.3. There are only three units per instance of Stage 1, which explains the behaviour observed in Figure 5.3.

**Control block**

Since the same computational units are being reused to perform many different operations, the necessary control is rather complex. The control block needs to provide the correct sequence of read and write addresses for the data RAMs, as well as other control signals, such as computation selection. An option would be to store the values for all control signals at every cycle in a program memory and have a counter iterating through them. However, this would take a large amount of memory. For this reason it was decided to trade a small increase in computational resources for a much larger decrease in memory requirements using complex instructions.

Frequently occurring memory access patterns have been identified and a dedicated address generator hardware block has been built to generate them from minimum storage. Each pattern is associated with a control instruction. Examples of these patterns are: simple increments $a, a+1, \ldots, a+b$ and the more complicated read patterns needed for matrix vector multiplication (standard and transposed). This approach allows storing only one instruction for a whole matrix-vector multiplication or for an arbitrary long sequence of additions.

The resulting sequential machine with custom complex instructions is close to 100% efficient, i.e. there are no cache misses or pipeline stalls, so a useful result is produced at every clock cycle. Control instructions to perform line search and linearization for one problem were stored. The sequence of instructions can be modified according to the type of constraints, to calculate the preconditioner $\mathcal{M}$ (if needed), and to recover the search direction from the result for the preconditioned system given by the linear solver. Since very few elements in $\Phi_k$ are changing from iteration to iteration, the updating of the

<table>
<thead>
<tr>
<th>Stage 1</th>
<th>3i</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dot-product (linear solver)</td>
<td>8$n_x + 4n_u - 3$</td>
</tr>
<tr>
<td>Other (linear solver)</td>
<td>27</td>
</tr>
<tr>
<td>Total</td>
<td>$8n_x + 4n_u + 24 + 3i$</td>
</tr>
</tbody>
</table>
preconditioner \( \mathcal{M} \) is not costly. When the last instruction is reached, the counter goes back to instruction 0 and iterates again for the next problem with the appropriate offsets being added to the control signals.

**Memory subsystem**

Separate memory blocks were used for data and control instructions, allowing simultaneous access and different word-lengths in a similar way to a Harvard microprocessor architecture. However, in our circuit there are no cache misses and a useful result can be produced almost every cycle. The data memories are divided in two blocks, each one feeding one input of the computational block. The intermediate results can be stored in any of these simple dual-port RAMs for flexibility in ordering computations. The memory to store the control instructions is divided into four single port ROMs corresponding to read and write addresses of each of the data RAMs. The responsibility for generating the remaining control signals is spread out over the four blocks.

### 5.4.3 Coefficient matrix storage

When implementing an algorithm in software, a large amount of memory is available for storing intermediate results. In FPGAs, there is a very limited amount of fast on-chip memory, around 4.5MBytes for high-end memory-dense Xilinx Virtex-6 devices [229]. If a particular design requires more memory than available on chip, there are two negative consequences. Firstly, if the size of the problems we can process is limited by the available on-chip memory, it means that the computational capabilities of the device are not being fully exploited, since there will be underutilised logic and DSP blocks. Secondly, if we were to try to overcome this problem by using off-chip memory, the performance of the circuit is likely to suffer since off-chip memory accesses are slow compared to the on-chip clock frequency. Specifically, for iterative linear solvers, if the coefficient matrix has to be loaded from off-chip memory at every iteration, the performance will be limited by memory bandwidth regardless of the amount of parallelisation employed. By taking into account the special structure of the matrices that are fed to the linear solver in the context of MPC, we can substantially reduce memory requirements so that this issue affects a smaller subset of problems.

The matrix \( A_k \) is banded and symmetric (after re-ordering). On-chip buffering of these type of matrices using compressed diagonal storage (CDS) can achieve substantial memory savings with minimum control overhead in an FPGA implementation of the MINRES method [22]. The memory reductions are achieved by only storing the non-zero diagonals of the original matrix as columns of the new compressed matrix. Since the matrix is also symmetric, only the right hand side of the CDS matrix needs to be stored, as the left-hand columns are just delayed versions of the stored columns. In order to achieve the same result when multiplying by a vector, the vector has to be aligned with its corresponding matrix components. It turns out that this is achieved by shifting the vector by one position at
The method described in [22] assumes a dense band; however, it is possible to achieve further memory savings by exploiting the time-invariance and multi-stage structure of the MPC problem further. The structure of the original matrix and corresponding CDS matrix for a small MPC problem with bound input constraints and general state constraints are shown in Figure 5.4, showing variables (elements that can vary from iteration to iteration of the interior-point method) and constants.

The first observation is that non-zero blocks are separated by layers of zeros in the CDS matrix. It is possible to only store one of these zeros per column and add common circuitry to generate appropriate sequences of read addresses, i.e.

$$0, 0, \cdots, 0, 1, 2, \cdots, n_u + n_x, 0, 0, \cdots, 0, n_u + n_x + 1, n_u + n_x + 2, \cdots, 2(n_u + n_x)$$

The second observation is that only a few diagonals adjacent to the main diagonal vary from iteration to iteration, while the rest remain constant at all times. This means that only a few columns in the CDS matrix contain varying elements. This has important implications, since in the MINRES implementation [21], matrices for all problems that are being processed simultaneously (see Section 5.5) have to be buffered on-chip. These memory blocks have to be double in size to allow writing the data for the next problems while reading the data for the current problems. Constant columns in the CDS matrix are common for all problems, hence the memories used to store them can be much smaller.
Finally, constant columns mainly consist of repeated blocks of size $2n_x + n_u$ (where $n_x$ values are zeros or ones), hence further memory savings can be attained by only storing one of those blocks per column.

A memory controller for the variable columns and another memory controller for the constant columns were created in order to be able to generate the necessary access patterns. The impact on the overall performance is negligible, since these controllers consume few resources compared with floating point units and they do not slow down the circuit.

If we consider a dense band, storing the coefficient matrix using CDS would require

$$2P(N(2n_x + n_u) + 2n_x) + 2n_x$$

elements, where $P$ is the number of problems being processed simultaneously (see Section 5.5). By taking into account the sparsity of matrices arising in MPC, it is possible to only store

$$2P(1 + N(n_u + n_x) + n_x) + (1 + n_u + n_x)(n_u + n_x)$$

elements. Figure 5.5 compares the memory requirements for storing the coefficient matrices on-chip when considering: a dense matrix, a banded symmetric matrix and an MPC matrix (all in single-precision floating-point). Memory savings of approximately 75% can be incurred by considering the in-band structure of the MPC problem compared to the standard CDS implementation. In practice, columns are stored in BlockRAMs of discrete sizes, therefore actual savings can vary in an FPGA implementation. Observe that exploitation of this kind of structure would not have been possible with factorization-based methods due to fill-in effects.

5.4.4 Preconditioning

Two options for implementing online preconditioning can be considered. The first option is to compute the preconditioned matrix $\tilde{A}_k$ in the sequential block and store it in the linear solver. This requires no extra computational resources; however, it imposes a significant extra computational load on the sequential block, which can slow down the overall execution and lead to low computational efficiency. It also prohibits the use of the customised reduced storage scheme just presented, since the non-zero elements that were previously constant between iterations are no longer constant in the preconditioned matrix.

The second option, which the present implementation adopts, only computes the preconditioner $M$ in the sequential block. The original matrix $A_k$ is stored in RAM in the linear solver block, and the preconditioner is applied on-the-fly by a bank of multipliers inserted at the memory output, as shown by Figure 5.6. This requires approximately three times as much computation per MINRES iteration; however, this computation is not on the critical path, i.e. memories storing the matrix can be read earlier, so the pre-
conditioning procedure has no effect on execution speed. The reduced storage scheme is retained at the cost of a significant increase in the number of multipliers. There is a clear trade-off between the extra resources needed to implement this procedure (see Table 5.8 in Section 5.6) and the amount of acceleration gained through a reduction in iteration count, which is again investigated in Section 5.6.

5.5 General performance results

In this section the focus is on investigating the scaling of performance and resource requirements with the problem dimension. A general problem setup with no preconditioning and dense state constraints is assumed and the performance is compared to a software microprocessor implementation of the same algorithm to evaluate the efficiency of the hardware design. In Section 5.6, the performance will be evaluated in detail in the context of a real-
istic benchmark for a specific instance of the solver and compared against state-of-the-art solutions.

5.5.1 Latency and throughput

Another benefit of FPGA technology for real-time applications is the ability to provide cycle accurate computation time guarantees. For the current design, computation time is given by

\[ \frac{IP PZ (IMR + c)}{f_c} \text{ seconds,} \]  

where \( f_c \) is the FPGA clock frequency and \( c \) is related to the proportion of time spent by the sequential block relative to the linear solver and varies with different implementations. \( IP \) and \( IMR \) are the number of interior-point and MINRES iterations, respectively, and

\[ P := \left\lceil \frac{2Z + M + 12[\log_2(2M - 1)] + 230}{Z} \right\rceil. \]  

For details on the derivation of (5.3), refer to [21]. The linear term results from the row by row processing for the matrix-vector multiplication (\( Z \) dot-products) and serial-to-parallel conversions – one of order \( Z \) and another of order \( M \), whereas the logarithmic term arises from the depth of the adder reduction tree in the dot-product block. The constant term comes from the other operations in the MINRES iteration.

If one’s objective is to maximize throughput to maximize hardware efficiency, then \( c = IMR \), since the latency of both main hardware blocks has to be the same. In that time the controller will be able to output the result to \( 2P \) problems (refer to Chapter 7 for more details on how to exploit this feature). It is important to note that \( P \) converges to a small number \((P = 3)\) as the size of \( A_k \) increases, thus for large problems only \( 2P = 6 \) independent threads are required to fully utilize the hardware.

5.5.2 Input/output requirements

Stage 1 is responsible for handling the chip I/O. The block reads the current state measurement \( x \) as \( n_x \) 32-bit floating point values sequentially through a 32-bit parallel input data port. Outputting the \( n_u \) 32-bit values for the optimal control move \( u_0^*(x) \) is handled in a similar fashion. When processing \( 2P \) problems, the average I/O requirements are given by

\[ \frac{2P(32(n_x + n_u))}{\text{Latency given by (5.2)}} \text{ bits/second.} \]

For the range of problems that we have considered in this section, the I/O requirements range from 0.2 to 10 kbits/second, which is well within any standard FPGA platform interface, such as PCI Express. The combination of a very computationally intensive task with very low I/O requirements, highlights the affinity of the FPGA for MPC computation.
5.5.3 Resource usage

The design was synthesized using Xilinx XST and placed and routed using Xilinx ISE 12 targeting a Virtex 6 SX 475T FPGA [229]. Figure 5.7 shows the different resources scaling with problem size. For fixed $n_u$ and $N$, the number of floating point units is $\Theta(n_x)$, illustrated by the linear growth in registers, look-up tables and embedded DSP blocks. The memory requirements are $\Theta(n_x^2)$, which explain the quadratic asymptotic growth observed in Figure 5.7. The jumps occur when the number of elements to be stored in the RAMs for variable columns exceeds the size of Xilinx BlockRAMs. The number of QP problems being processed simultaneously only affects the memory requirements.

5.5.4 FPGA vs software comparison

Post place-and-route results showed that a clock frequency above 250MHz is achievable with very small variations for different problem sizes, since the critical path is inside the control block in Stage 1. Figure 5.8 shows the latency and throughput performance of the FPGA and latency results for a microprocessor implementation. For the software benchmark, we have used a direct C sequential implementation, compiled using GCC -O4 optimizations running on a Intel Core2 Q8300 with 3GB of RAM, 4MB L2 cache, and a clock frequency of 2.5GHz running Linux. Note that for matrix operations of this size, this approach produces better performance software than using libraries such as Intel MKL.

The FPGA implementation starts to outperform the microprocessor as soon as there is enough parallelism to overcome the clock frequency disadvantage (this happens when $n_x > 3$ for the considered problem dimensions). The performance gap widens as the size of
the optimization problem increases as a result of increased parallelism in the linear solver. The FPGA throughput curve represents the number of interior-point iterations per second when processing several problems simultaneously.

The normalized CPU curve in Figure 5.8 illustrates the performance of a sequential implementation running at the same frequency as the FPGA, hence can be used to compare the number of cycles needed in both implementations. For largest problem considered, comparing against an efficient microprocessor implementation of the same algorithm, the current FPGA implementation can provide approximately $15\times$ reduction in latency and $85\times$ improvement in throughput if there are enough available independent problems. In terms of clock cycles, there will be an extra order of magnitude performance improvement.

Xilinx XPower analyzer [231] was used to estimate the device power for the FPGA implementation. The tool gives a conservative power consumption estimate based on the design’s operating frequency and post-place-and-route resource utilization data. For the range of problems considered in Figure 5.9, the device power varied almost linearly from 6.7 Watts to 20.7 Watts. This is a consequence of the computational resources growing linearly with the number of states. It is important to note that no power optimization flags were turned on during synthesis since the main goal is high performance. In order to include the energy consumed by the FPGA’s peripherals, the idle power consumption of
Figure 5.9: Energy per interior-point iteration for the CPU, and FPGA implementations when solving one problem and $2^P$ problems, where $P$ is given by (5.3). Problem parameters are $n_u = 3$, $N = 20$ and $f_c = 250$MHz.

a Xilinx ML605 board was measured to be 7.9 Watts and added it to the FPGA’s power requirements.

For the high performance CPU implementation running Linux, the average power drained from the mains supply was approximately 76 Watts. Figure 5.9 combines measurements of execution time with the power consumption estimates to calculate the energy efficiency of each implementation. For most problems, the FPGA is both faster than the CPU and consumes less power, hence the overall energy performance is significantly better. For the largest problem reported, the FPGA provides a $38 \times$ improvement in energy efficiency when processing one problem or $227 \times$ improvement when processing several problems simultaneously.

### 5.6 Boeing 747 case study

In this implementation case study, the control of the roll, pitch and airspeed of a nonlinear Simulink-based model of the rigid-body dynamics of a Boeing 747-200 with individually manipulable controls surfaces of $[57, 127]$ is considered.

#### 5.6.1 Prediction model and cost

A prediction model of the form (4.3b) is obtained by linearisation of the nonlinear model about an equilibrium trim point for straight and level flight at an altitude of 600 meters.
and an airspeed of 133 meters per second, discretised with a sample period of $T_s = 0.2$ seconds. The linearised model considers 14 states (roll rate, pitch rate, yaw rate, airspeed, angle of attack, sideslip angle, roll, pitch, yaw, altitude, and four engine power states). Yaw angle and altitude are neglected in the prediction model used for the predictive controller, since they do not affect the roll, pitch and airspeed (leaving 12 remaining states). The 17 inputs considered consist of four individually manipulable ailerons, left spoiler panels, right spoiler panels, four individually manipulable elevators, a stabiliser, upper and lower rudder, and four engines. The effects of landing gear and flaps are not considered as these substantially change the local linearisation. The disturbance input matrix $B_w$ is selected to describe a zero-order-hold state disturbance on the first 10 states. The cost function (4.2) is chosen with $S = 0$, $\sigma_1 = \sigma_2 = 0$, and the remaining weights as described in Table 5.4. The constraints on the inputs are summarised in Table 5.5.

### Table 5.4: Cost function

<table>
<thead>
<tr>
<th>Matrix</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>$Q$</td>
<td>diag(7200, 1200, 1400, 8, 1200, 2400, 4800, 4800, 0.005, 0.005, 0.005, 0.005)</td>
</tr>
<tr>
<td>$R$</td>
<td>diag(0.002, 0.002, 0.002, 0.002, 0.003, 0.003, 0.02, 0.02, 0.02, 21, 0.05, 0.05, 3, 3, 3, 3)</td>
</tr>
<tr>
<td>$Q_N$</td>
<td>Solution to discrete-time algebraic Riccati equation</td>
</tr>
</tbody>
</table>

### Table 5.5: Input constraints

<table>
<thead>
<tr>
<th>Input</th>
<th>Feasible region</th>
<th>Units</th>
</tr>
</thead>
<tbody>
<tr>
<td>1,2</td>
<td>Right, left inboard aileron</td>
<td>$[-20, 20]$</td>
</tr>
<tr>
<td>3,4</td>
<td>Right, left outboard aileron</td>
<td>$[-25, 15]$</td>
</tr>
<tr>
<td>5,6</td>
<td>Right, left spoiler panel array</td>
<td>$[0, 45]$</td>
</tr>
<tr>
<td>7,8</td>
<td>Right, left inboard elevator</td>
<td>$[-23, 17]$</td>
</tr>
<tr>
<td>9,10</td>
<td>Right, left outboard elevator</td>
<td>$[-23, 17]$</td>
</tr>
<tr>
<td>11</td>
<td>Stabiliser</td>
<td>$[-12, 3]$</td>
</tr>
<tr>
<td>12,13</td>
<td>Upper, lower rudder</td>
<td>$[-25, 25]$</td>
</tr>
<tr>
<td>14–17</td>
<td>Engines 1–4</td>
<td>$[0.94, 1.62]$</td>
</tr>
</tbody>
</table>

5.6.2 Target calculator

For nominal offset-free steady state tracking of reference setpoints it is common to use a target calculator (refer to Figure 2.2) to calculate $x_{ss}$ and $u_{ss}$ [141, 159, 175], that satisfy

$$A_d x_{ss} + B_d u_{ss} + B_w \hat{w} = x_{ss},$$  \hspace{1cm} (5.4a)

$$C_r x_{ss} = r,$$  \hspace{1cm} (5.4b)

and

$$u_{\text{min}} \leq u_{ss} \leq u_{\text{max}}, \quad x_{\text{min}} \leq x_{ss} \leq x_{\text{max}},$$  \hspace{1cm} (5.4c)
where \( C_r \in \mathbb{R}^{n_r \times n_r} \), and \( r \in \mathbb{R}^{n_r} \) is a vector of \( n_r \) reference setpoints to be tracked without offset. A feasible solution to (5.4) is not guaranteed to exist. Let

\[
A_s := \begin{bmatrix} (A_d - I) & B_d \\ C_r & 0 \end{bmatrix}, \quad B_s := \begin{bmatrix} -B_w & 0 \\ 0 & I \end{bmatrix}, \quad \theta_s := \begin{bmatrix} x_{ss}^T, u_{ss}^T \end{bmatrix}, \quad b_s := \begin{bmatrix} \hat{w} \\ r \end{bmatrix},
\]

and \( W > 0 \) be a weighting matrix. The solution of

\[
\min_{\theta_s} \frac{1}{2} \theta_s^T A_s^T W A_s \theta_s - b_s^T B_s^T W A_s \theta_s
\]

subject to (5.4c) will find a solution satisfying the equality constraints if one exists and return a least-squares approximation if one does not. This is a dense QP with no equality constraints; however, it is not guaranteed that \( A_s^T W A_s > 0 \), hence the QP might not be strictly convex. The remaining degrees of freedom can be exploited by defining \( Q := Q \oplus R, A_s^+ \) to be a matrix whose columns form an orthogonal basis of \( \text{Ker}(A_s) \) and \( H_s := A_s^T W A_s + A_s^+ A_s^T Q A_s^+ A_s^T \). The (now strictly convex) target calculation problem can now be posed as

\[
\min_{\theta_s} \frac{1}{2} \theta_s^T H_s \theta_s - b_s^T B_s^T W A_s \theta_s
\]

subject to (5.4c). The optimal values \( x_{ss}^*(r) \) and \( u_{ss}^*(r) \) are then used as the setpoints in the regulation problem (4.2).

The target calculator is configured with \( C_r = [e_4, e_7, e_8]^T \), where \( e_i \) is the \( i \)th column of the 29 \( \times \) 29 element identity matrix, in order that the references to be tracked are airspeed, roll, and pitch angle. The weighting matrix \( W \) is selected as an appropriately sized identity matrix.

The computational time of this relatively simple QP is almost negligible compared to the MPC regulation QP solver, hence high-level hardware design tools are used for its design and its details are omitted in this thesis. Refer to [86] for further details.

### 5.6.3 Observer

The control system described by Figure 2.2 can also include an observer. The twelve states \( x \) of the model (4.3b) are assumed measurable, along with the two variables that were neglected in the prediction model: the altitude, and the yaw angle. The disturbance \( w \) cannot be measured. As is standard practice in predictive control, an observer is therefore used to estimate \( w \) as \( \hat{w} \). The observer includes a one step ahead prediction to allow the combined target calculator and predictive regulator a deadline of one sampling period for computation.
5.6.4 Online preconditioning

Empirical evidence suggests that the following simple diagonal preconditioner, which normalizes the 1-norm of the rows of the matrix and will be discussed in detail in Chapter 8, significantly reduces the number of MINRES iterations necessary to achieve a satisfactory solution to Line 3 of Algorithm 1 for this example problem and other MPC problems. The diagonal entries of the preconditioner are given by

$$M_{ii} := 1/\sqrt{\sum_{j=1}^{Z} |A_{ij}|}. \quad (5.7)$$

5.6.5 Offline pre-scaling

For each primal-dual interior-point (PDIP) iteration, the convergence of the MINRES algorithm used to solve $A_k \xi_k = b_k$, and the accuracy of the final estimate of $\xi_k$ are influenced by the eigenvalue distribution of $A_k$. When no scaling is performed on the prediction model and cost matrices for this application, and no preconditioning is applied online, large inaccuracy in the estimates of $\xi_k$ leads the PDIP algorithm to not converge to a satisfactory solution. Increasing the number of MINRES iterations fails to improve the solution, yet increases the computational burden.

Preconditioning applied online at each iteration of the PDIP algorithm can accelerate convergence, and reduce the worst-case solution error of $\xi_k$. In [85], offline pre-scaling was used in lieu of an on-line preconditioner, with the control performance demonstrated competitive with respect to the use of conventional factorisation-based algorithms on a general purpose platform. The rationale behind the pre-scaling procedure is now stated and numerical results presented to demonstrate that combining systematic offline pre-scaling with online preconditioning yields better performance compared to mutually exclusive use.

Matrix $A_k$ is not constant, but $W_k^{-1}$ is diagonal. Since there are only upper and lower bounds on inputs, the varying component of $A_k$, $\Phi_k$, only has diagonal elements. Moreover, as $k \to I_{IP} - 1$, the elements of $W_k^{-1}$ corresponding to inactive constraints approach zero. Therefore, despite the diagonal elements of $W_k^{-1}$ corresponding to active constraints becoming large, as long as only a handful of these exist at any point, the perturbation to $\hat{A}$ is of low rank, and will have a relatively minor effect on the convergence of MINRES.

Hence, rescaling the control problem to improve the conditioning of $\hat{A}$ should also improve the conditioning of $A_k$ in some sense.

Prior to scaling, for $N = 12$, the condition number of $\hat{A}$ is $1.77 \times 10^7$. The objective of the following procedure is to obtain diagonal matrices $T_Q > 0$ and $T_R > 0$ to scale the linear state space prediction model and quadratic cost weighting matrices as follows:

$$A_d \leftarrow T_Q A_d T_Q^{-1}, \quad B_d \leftarrow T_Q B_d T_R^{-1}, \quad B_w \leftarrow T_Q B_w, \quad Q \leftarrow T_Q^{-1} Q T_Q^{-1}, \quad R \leftarrow T_R^{-1} R T_R^{-1}, \quad u_{\text{min}} \leftarrow T_R u_{\text{min}}, \quad u_{\text{max}} \leftarrow T_R u_{\text{max}}.$$
This substitution is equivalent to

\[ \hat{A} \leftarrow \hat{M} \hat{A} \hat{M}, \quad \text{where} \]

\[ \hat{M} := \left( I_N \otimes \left( T_Q^{-1} \oplus T_R^{-1} \right) \right) \oplus \left( T_Q^{-1} \right) \oplus \left( I_{N+1} \otimes T_Q \right). \] (5.8)

By constraining \( T_Q = \text{diag}(t_Q) \) and \( T_R = \text{diag}(t_R) \), the diagonal structure of \( \Phi_k \) is retained. The transformation (5.8) is a function of both \( T_Q \) and its inverse, and both of these appear quadratically, so it is therefore likely that minimisation of any particular function of \( \hat{M} \hat{A} \hat{M} \) is not (in general) going to be particularly well conditioned. In [18] some guidelines are provided for desirable scaling properties. In particular, it is desirable to normalise the rows and columns of \( \hat{A} \) so that they are all of similar magnitude.

Whilst not exactly the original purpose, it should be noted that if the online preconditioner (5.7) is applied repeatedly (i.e. re-preconditioning the same matrix multiple times) to a general square matrix of full rank, the 1-norm of each of the rows converges asymptotically to unity. The method proposed here for normalising \( \hat{A} \) follows naturally but with the further caveat that the structure of \( \hat{M} \) is imposed to be of the form (5.9). Consequently, it is not (in general) possible to scale \( \hat{A} \) such that all row norms are equal to an arbitrary value. Instead, the objective is to reduce the variation in row (and column) norms. Empirical testing suggests that normalising the 2-norm of the rows of \( \hat{A} \) (subject to (5.9)) gives the most accurate solutions from Algorithm 1 for the present application.

Noting the structure of \( \hat{A} \), define the following vectors:

\[ s_x := \left\{ s_x \in \mathbb{R}^n : s_x,\{i\} = \left( \sum_{j=1}^n Q_{ij}^2 + \sum_{j=1}^n A_{d,ji}^2 + 1 \right)^{1/2} \right\}, \]
\[ s_u := \left\{ s_u \in \mathbb{R}^m : s_u,\{i\} = \left( \sum_{j=1}^m R_{ij}^2 + \sum_{j=1}^n B_{d,ji}^2 + 1 \right)^{1/2} \right\}, \]
\[ s_N := \left\{ s_N \in \mathbb{R}^n : s_N,\{i\} = \left( \sum_{j=1}^n Q_{N,ij}^2 + 1 \right)^{1/2} \right\}, \]
\[ s_\lambda := \left\{ s_\lambda \in \mathbb{R}^n : s_\lambda,\{i\} = \left( \sum_{j=1}^n A_{d,ij}^2 + \sum_{j=1}^m B_{d,ij}^2 + 1 \right)^{1/2} \right\}. \]

Also, define elementwise,

\[ l_1 := \sqrt{s_u/\mu}, \]
\[ l_2 := \left\{ l_2 \in \mathbb{R}^n > 0 : l_2^4 = \left( \frac{(N s_x + s_N)/(1 + N s_\lambda))} \right) \right\}, \]
where \( \mu := (N \sum s_x + \sum s_N + N \sum s_\lambda + n_x)/(2(N + 1)n_x) \), and apply Algorithm 2.

Table 5.6 shows properties of \( \hat{A} \) that influence solution quality, before and after application of the prescaling with \( \epsilon = 10^{-7} \). These are the condition number of \( \hat{A} \), the standard deviation of the row 1– and 2–norms, and the standard deviation of the magnitude of the eigenvalues, which are substantially reduced by the scaling.

Figure 5.10 shows three metrics for the quality of the solution from the MINRES-
**Algorithm 2** Offline prescaling algorithm

**Require:** $A_d, B_d, Q, R, P, \epsilon$ and $t_Q \leftarrow 1_n,$ and $t_R \leftarrow 1_m$

1: **repeat**
2: Calculate $l_1, l_2$ as functions of current data, and define $L_1 := \text{diag}(l_1), L_2 := \text{diag}(l_2)$.
3: Update:

\[
\begin{align*}
& t_Q \leftarrow L_2 t_Q, \quad t_R \leftarrow L_1 t_R, \quad A_d \leftarrow L_2 A_d L_2^{-1}, \quad B_d \leftarrow L_2 B_d L_1^{-1}, \\
& Q \leftarrow L_2^{-1} Q L_2^{-1}, \quad P \leftarrow L_2^{-1} P L_2^{-1}, \quad R \leftarrow L_1^{-1} R L_1^{-1}.
\end{align*}
\]
4: **until** $(\|l_2 - 1\| < \epsilon) \cap (\|l_1 - 1\| < \epsilon)$
5: **Output:** $T_Q := \text{diag}(t_Q), T_R := \text{diag}(t_R)$.

**Table 5.6:** Effects of offline preconditioning

| Scaling | $\text{cond}(A)$ | $\text{std} \|A_{(i,:)}\|_1$ | $\text{std} \|A_{(i,:)}\|_2$ | $\text{std} |\lambda_i(A)|$ |
|---------|------------------|------------------|------------------|------------------|
| Original | $1.77 \times 10^7$ | $5.51 \times 10^3$ | $4.33 \times 10^3$ | $4.35 \times 10^3$ |
| Scaled   | $2.99 \times 10^4$ | $0.6845$         | $0.5984$         | $0.6226$         |

Based PDIP solver over the duration of a closed-loop simulation with a prediction horizon $N = 12$. The number of MINRES iterations per PDIP iteration is varied for four different approaches to preconditioning (none, offline, online, and combined online and offline). Whilst these experiments were performed in software, a theoretical computation time using (5.2) with the value of $c$ given by Table 5.7 for the FPGA implementation, is also shown.

With neither preconditioning nor offline scaling, the control performance is unacceptable. Even when the number of MINRES iterations is equal to $2Z = 2 \times 516 = 1032$, the mean stage cost over the simulation is high (the controller failed to stabilise the aircraft), and the worst case control error in comparison to a conventional PDIP solver using double precision arithmetic and a factorisation-based approach is of the same order as the range of the control inputs. Using solely online preconditioning, control performance (in terms of the cost function) does not start to deteriorate significantly until the number of MINRES iterations is reduced to $\lfloor 0.25Z \rfloor = 129$, although at this stage, the worst case relative accuracy is still poor (but mean relative accuracy is tolerable). With only offline preconditioning, worst case relative control error does not deteriorate until the number of MINRES iterations is reduced to $\lfloor 0.75Z \rfloor = 387$ and control performance does not

**Table 5.7:** Values for $c$ in (5.2) for different implementations.

<table>
<thead>
<tr>
<th>$N$</th>
<th>Online preconditioning</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Yes</td>
</tr>
<tr>
<td>5</td>
<td>29</td>
</tr>
<tr>
<td>12</td>
<td>40</td>
</tr>
</tbody>
</table>
Figure 5.10: Numerical performance for a closed-loop simulation with $N = 12$, using PC-based MINRES-PDIP implementation with no preconditioning (top left), offline preconditioning only (top right), online preconditioning only (bottom left), and both (bottom right). Missing markers for the mean error indicate that at least one control evaluation failed due to numerical errors.

deteriorate until this is reduced to $|0.1Z| = 51$. If the online and offline approaches are combined, control performance is maintained with $|0.03Z| = 15$ iterations, and worst case control accuracy is maintained with $|0.08Z| = 41$ iterations, showing that one can achieve substantial computation reductions by using iterative solvers.

5.6.6 FPGA-in-the-loop testbench

The hardware-in-the-loop experimental setup used to test the predictive controller design has two goals: providing a reliable real-time closed-loop simulation framework for controller design verification; and demonstrating that the controller could be plugged into a plant presenting an appropriate interface. Figure 5.11 shows a schematic of the experimental setup. The QP solver, running on a Xilinx FPGA ML605 evaluation board [233], controls the nonlinear model of the B747 aircraft running in SIMULINK on a PC. At every sampling instant $k$, the observer estimates the next state $\hat{x}_{k+1|k}$ and disturbance $\hat{w}_{k+1|k}$. For the testbench, the roll, pitch and airspeed setpoints comprising the reference signal $r$ in the target calculator (5.4) that the predictive controller is designed to track, are provided by simple linear control loops, with the roll and pitch setpoints as a function of a reference yaw angle and reference altitude respectively, and the airspeed setpoint passed through a low-pass filter. The vectors $\hat{x}_{k+1|k}$, $\hat{w}_{k+1|k}$ and $r$ are represented as a sequence of single-precision floating point numbers in the payload of a UDP packet via an S-function and this is transmitted over 100 Mbit/s Ethernet. The FPGA returns the control action in another UDP packet. This is applied to the plant model at the next sampling instant.
Figure 5.11: Hardware-in-the-loop experimental setup. The computed control action by the QP solver is encapsulated into a UDP packet and sent through an Ethernet link to a desktop PC, which decodes the data packet, applies the control action to the plant and returns new state, disturbance and trajectory estimates. lwip stands for light-weight TCP/IP stack.

On the controller side, the transferred data is captured by the Physical Layer Device (PHY) on an ML605 evaluation board, implementing the physical layer of the Ethernet stack, and then transmitted to the FPGA chip. The data link layer is implemented in hardware with a Media Access Control (Ethernet MAC) provided by the FPGA manufacturer. The transport and network layers of the UDP stack are provided by lwIP and run on an embedded soft processor IP-core (Xilinx MicroBlaze). The decoded UDP packet is routed to a mixed software-hardware application layer.

On the FPGA the two custom hardware circuits implementing the QP solvers for target calculation and MPC regulation are connected to a Xilinx MicroBlaze soft core processor, upon which a software application bridges the communication between the Ethernet interface and the two QP solvers. As well as being simpler to implement, this architecture provides some system flexibility in comparison to a dedicated custom interface, with a small increase in FPGA resource usage (Table 5.8) and communication delay, and allows easy portability to other standard interfaces, e.g., SpaceWire, CAN bus, etc., as well as an option for direct monitoring of controller behaviour.

Table 5.8 shows the FPGA resource usage of the different components in the system-on-a-chip testbench, as well as the proportion of the FPGA used for two mid-range devices with approximately the same silicon area from the last two technology generations (the newer FPGA offers more resources per unit area, meaning a smaller, cheaper, lower power model can be chosen). The difference in the proportion of the FPGA used for the Virtex 6 and Virtex 7 devices emphasises the continuous increase in transistor density from which new generations of FPGA technology continue to benefit from. The linear solver uses the majority of the resources in the MPC QP solver, while the MPC QP solver consumes substantially more resources than the target calculator, since it is solving a larger
Synthesis estimate of absolute and percentage resource usage of the FPGA mounted on the Xilinx ML605 (round brackets) and Xilinx VC707 (square brackets) Evaluation Boards. An field-programmable gate array (FPGA) consists of look-up tables (LUT), registers (REG), embedded RAM blocks (BRAM) and multiplier blocks (DSP48E).

5.6.7 Evaluation

A closed-loop system with the FPGA in the loop, controlling the nonlinear model of the Boeing 747 from [127] is compared with running the complete control system on a conventional computer using factorisation-based methods. The MPC regulator QP solver is first evaluated separately, and then trajectory plots of the closed loop trajectories for the complete system are presented. The reference trajectory is continuous, piecewise continuous in its first derivative, and consists of a period of level flight, followed by a 90 degree change in heading, then by a 200 metre descent, followed by a 10 metre per second deceleration.

Solution times and control quality metrics for the regulator QP solver are presented for a 360 second simulation, with $N = 12$ and $N = 5$ in Table 5.9. Based on the numerical results in the previous subsection, for $N = 12$, the number of MINRES iterations per PDIP iteration is set to $I_{MR} = 51$. For $N = 5$, $I_{MR} = 30$. This is higher than was empirically determined to be necessary; however, the architecture of the QP solver requires that the MINRES stage must run for at least as long as the sequential stage. The control accuracy metrics presented are

$$e_{\text{max}} := \max_i \left| u^F_{\{i\}} - u^*_\{i\} \right| / \left( u_{\text{max},\{i\}} - u_{\text{min},\{i\}} \right)$$

$$e_\mu := \text{mean}_i \left| u^F_{\{i\}} - u^*_\{i\} \right| / \left( u_{\text{max},\{i\}} - u_{\text{min},\{i\}} \right)$$

where $u^F(k)$ is the calculated control input, and $u^*(k)$ is the hypothetical true solution and the subscript `\{i\}` indicates an elementwise index. Since the true solution is not possible to obtain analytically, the algorithm of [184], implemented using MATLAB Coder, is used

### Table 5.8: FPGA resource usage.

<table>
<thead>
<tr>
<th></th>
<th>MicroBlaze Target calculator</th>
<th></th>
<th>MicroBlaze Target calculator</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>LUT</strong></td>
<td></td>
<td>9081 (6%) [3%]</td>
<td>4469 (3%) [1%]</td>
</tr>
<tr>
<td><strong>REG</strong></td>
<td></td>
<td>7814 (3%) [1%]</td>
<td>9211 (3%) [2%]</td>
</tr>
<tr>
<td><strong>BRAM</strong></td>
<td></td>
<td>40 (10%) [4%]</td>
<td>5 (1%) [0%]</td>
</tr>
<tr>
<td><strong>DSP48E</strong></td>
<td></td>
<td>5 (1%) [0%]</td>
<td>66 (9%) [2%]</td>
</tr>
</tbody>
</table>

103
Table 5.9: Comparison of FPGA-based MPC regulator performance (with baseline floating point target calculation in software)

<table>
<thead>
<tr>
<th>Implementation</th>
<th>QP Solver</th>
<th>Bits</th>
<th>N</th>
<th>$l_{MR}$</th>
<th>$e_{max}$</th>
<th>$e_{\mu}$</th>
<th>Mean cost</th>
<th>Max Solution time</th>
<th>Clock cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>F /P-MINRES</td>
<td>32</td>
<td>12</td>
<td>51</td>
<td></td>
<td>$9.67 \times 10^{-4}$</td>
<td>$3.02 \times 10^{-5}$</td>
<td>5.2236</td>
<td>12</td>
<td>2.89 $\times 10^{7}$</td>
</tr>
<tr>
<td>PC/RWR1998</td>
<td>64</td>
<td>12</td>
<td>--</td>
<td></td>
<td>--</td>
<td>--</td>
<td>5.2247</td>
<td>23</td>
<td>5.59 $\times 10^{7}$</td>
</tr>
<tr>
<td>PC/FORCES</td>
<td>64</td>
<td>12</td>
<td>--</td>
<td></td>
<td>$5.89 \times 10^{-3}$</td>
<td>$1.69 \times 10^{-4}$</td>
<td>5.2250</td>
<td>13</td>
<td>3.09 $\times 10^{7}$</td>
</tr>
<tr>
<td>UB/FORCES</td>
<td>32</td>
<td>12</td>
<td>--</td>
<td></td>
<td>$3.83 \times 10^{-3}$</td>
<td>$7.31 \times 10^{-5}$</td>
<td>5.2249</td>
<td>1911</td>
<td>1.91 $\times 10^{8}$</td>
</tr>
<tr>
<td>F /P-MINRES</td>
<td>32</td>
<td>5</td>
<td>30</td>
<td></td>
<td>$9.10 \times 10^{-4}$</td>
<td>$2.95 \times 10^{-5}$</td>
<td>5.2203</td>
<td>4</td>
<td>1.09 $\times 10^{7}$</td>
</tr>
<tr>
<td>PC/RWR1998</td>
<td>64</td>
<td>5</td>
<td>--</td>
<td></td>
<td>--</td>
<td>--</td>
<td>5.2204</td>
<td>11</td>
<td>2.64 $\times 10^{7}$</td>
</tr>
<tr>
<td>PC/CVXGEN</td>
<td>64</td>
<td>5</td>
<td>--</td>
<td></td>
<td>$1.04 \times 10^{-3}$</td>
<td>$1.84 \times 10^{-5}$</td>
<td>5.2203</td>
<td>3</td>
<td>7.20 $\times 10^{6}$</td>
</tr>
<tr>
<td>PC/FORCES</td>
<td>64</td>
<td>5</td>
<td>--</td>
<td></td>
<td>$5.00 \times 10^{-3}$</td>
<td>$1.24 \times 10^{-4}$</td>
<td>5.2207</td>
<td>6</td>
<td>1.44 $\times 10^{7}$</td>
</tr>
<tr>
<td>UB/CVXGEN</td>
<td>32</td>
<td>5</td>
<td>--</td>
<td></td>
<td>??</td>
<td>??</td>
<td>??</td>
<td>(260)</td>
<td>(2.69 $\times 10^{7}$)</td>
</tr>
<tr>
<td>UB/FORCES</td>
<td>32</td>
<td>5</td>
<td>--</td>
<td></td>
<td>$4.14 \times 10^{-3}$</td>
<td>$8.01 \times 10^{-5}$</td>
<td>5.2205</td>
<td>823</td>
<td>8.23 $\times 10^{7}$</td>
</tr>
</tbody>
</table>

(FPGA QP solver (F) running at 250 MHz, PC (PC) at 2.4 GHz and MicroBlaze (UB) at 100 MHz. (--) indicates a baseline. (??) indicates that meaningful data for control could not be obtained). P-MINRES indicates preconditioned MINRES. RWR1998 indicates the algorithm of [184].

as a baseline.

The metrics are presented alongside those for custom software QP solvers generated using the state-of-the-art CVXGEN [146] (for $N = 5$ only since for $N = 12$ the problem was too large to handle) and FORCES [49] (for $N = 12$ and $N = 5$) tools. PC-based comparisons are made using double precision arithmetic on a laptop with a 2.4 GHz Intel Core 2 Duo processor. The code from CVXGEN and FORCES is modified to use single precision arithmetic and timed running directly on the 100MHz MicroBlaze soft core on the FPGA for the number of iterations observed necessary on the PC. Whilst obtaining results useful for control from the single precision modification to the CVXGEN solver proved to be too challenging, the timing result is presented assuming random data for the number of iterations needed on the PC. The MicroBlaze used for the software solvers is configured with throughput (rather than area) optimizations, single precision floating point unit (including square root), maximum cache size (64 KB for data and 64 KB for instructions), and maximum cache line length (8 words).

For $N = 12$, the FPGA-based QP solver (at 250 MHz) is slightly faster than the PC-based QP solver generated using FORCES (at 2.4 GHz) based on wall-clock time but approximately 10× faster on a cycle-by-cycle basis. It is also approximately 65× faster than the FORCES solver on the MicroBlaze (at 100 MHz), which would fail to meet the real-time deadline of $T_s = 0.2$ seconds by an order of magnitude. By contrast, the clock frequency for the FPGA-based QP solver could be reduced by a factor of 15 (reducing power requirements, and making a higher FPGA resource sharing factor possible), or the sampling rate increased by the same factor (improving disturbance rejection) whilst still meeting requirements. Worst-case and mean control error are competitive. A similar trend is visible for $N = 5$ with the FPGA-based solver only marginally slower than the CVXGEN solver on the PC in terms of wall-clock time.

The maximum communication time over Ethernet, experimentally obtained by bypass-

\[\text{Double precision floating point arithmetic would be emulated in software in the MicroBlaze processor, and not provide a useful timing comparison.}\]
ing the interface with the QP solvers in the software component is 0.67 milliseconds. The values for FPGA-based implementation in Table 5.9 are normalised by subtracting this, since it is independent of the QP solver.

Trajectories from the closed-loop setup, with $N = 12$ for the entire system-on-a-chip running on the FPGA are shown in Figure 5.12. The reference trajectory is tracked, inputs constraints are enforced during transients, and the zero-value lower bound on the spoiler panels is not violated in steady state. A video demonstration of the setup can be found in [105].

### 5.7 Summary and open questions

This chapter has described a parameterizable FPGA architecture for solving QP optimization problems in linear time-invariant MPC using primal-dual interior-point methods. Various design decisions have been justified based on the significant exploitable structure in the problem. The main source of acceleration is a parallel iterative linear solver block, which reduces the latency of the main computational bottleneck in the optimization method. Results show that a significant reduction in latency is possible compared to a sequential software implementation, which could translate to high sampling frequencies and better quality control.

This chapter has also demonstrated the implementation of a system-on-a-chip MPC control system, including predictive control regulation and steady-state target calculation on an FPGA. A Xilinx MicroBlaze soft-core processor is used to bridge communication between the two custom QP solvers, and the outside world over Ethernet. The controller is tested in closed-loop controlling a non-linear simulation of a large airliner – a plant with substantially more states and inputs than any previous FPGA-based predictive controller. A numerical investigation shows that with preconditioning and the correct plant model scaling, a relatively small number of linear solver iterations is required to achieve sufficient control accuracy for this application. The whole system fits comfortably on a mid-range FPGA, and lower clock frequencies could be used whilst still meeting real-time control deadlines.

The operations outside of the linear solver are currently computed in a sequential fashion in a complex instruction machine that is significantly more efficient than a soft-core MicroBlaze for this specific computation. There exist now heterogeneous FPGA architectures that include a higher performance hard RISC ARM processor embedded in custom logic. Future work could investigate the possibility of porting the proposed architecture to these heterogeneous devices, where the custom logic would only implement the parallel linear solver operations and the ARM processor would handle everything else including communications with the outside world.

Even though the FPGA logic is capable of handling single precision floating-point arithmetic, being able to implement the computationally intensive part of the optimization solver using fixed-point arithmetic would considerably improve the computational effi-
Figure 5.12: Closed loop roll, pitch, yaw, altitude and airspeed trajectories (top) and input trajectory with constraints (bottom) from FPGA-in-the-loop testbench.
### Table 5.10: Table of symbols

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>$n_u$</td>
<td>input dimension</td>
</tr>
<tr>
<td>$n_x$</td>
<td>state dimension</td>
</tr>
<tr>
<td>$N$</td>
<td>control horizon length</td>
</tr>
<tr>
<td>$\mathcal{A}_k$</td>
<td>KKT matrix</td>
</tr>
<tr>
<td>$\hat{\mathcal{A}}_k$</td>
<td>preconditioned KKT matrix</td>
</tr>
<tr>
<td>$\mathcal{M}_k$</td>
<td>preconditioner</td>
</tr>
<tr>
<td>$\sigma$</td>
<td>centrality parameter</td>
</tr>
<tr>
<td>$H$</td>
<td>QP Hessian</td>
</tr>
<tr>
<td>$G$</td>
<td>QP inequality constraint matrix</td>
</tr>
<tr>
<td>$F$</td>
<td>QP equality constraint matrix</td>
</tr>
<tr>
<td>$A_d$</td>
<td>discrete-time state-transition matrix</td>
</tr>
<tr>
<td>$B_d$</td>
<td>discrete-time input matrix</td>
</tr>
<tr>
<td>$B_w$</td>
<td>discrete-time disturbance matrix</td>
</tr>
<tr>
<td>$Q$</td>
<td>state penalty matrix</td>
</tr>
<tr>
<td>$R$</td>
<td>input penalty matrix</td>
</tr>
<tr>
<td>$S$</td>
<td>cross penalty matrix</td>
</tr>
<tr>
<td>$J$</td>
<td>state constraint matrix</td>
</tr>
<tr>
<td>$E$</td>
<td>input constraint matrix</td>
</tr>
<tr>
<td>$E^-$</td>
<td>input-rate constraint matrix</td>
</tr>
<tr>
<td>$I$</td>
<td>number of inequality constraints</td>
</tr>
<tr>
<td>$I_{ls}$</td>
<td>number of line search iterations</td>
</tr>
<tr>
<td>$Z$</td>
<td>size of the KKT matrix</td>
</tr>
<tr>
<td>$M$</td>
<td>halfband of the KKT matrix</td>
</tr>
<tr>
<td>$P$</td>
<td>number of problems that can be solved simultaneously in the linear solver block</td>
</tr>
<tr>
<td>$x_{ss}$</td>
<td>state setpoint</td>
</tr>
<tr>
<td>$u_{ss}$</td>
<td>input setpoint</td>
</tr>
<tr>
<td>$\hat{w}$</td>
<td>disturbance estimate</td>
</tr>
</tbody>
</table>

The efficiency of the resulting solution. Chapter 8 is a first step in this direction. Additional investigation into the numerical precision necessary for interior-point methods to behave in a reliable way would allow to explore further the efficiency trade-offs that are possible in custom hardware.
6 Hardware Acceleration of Fixed-Point First-Order Solvers

The intense computational demand imposed by MPC precludes its use in applications that could benefit considerably from its advantages, especially in those that have fast required response times and in those that must run on resource-constrained, embedded computing platforms with low cost or low power requirements. In this chapter the focus is on the design and analysis of custom circuits for first-order optimization solvers, which are often a more efficient alternative to the methods implemented in Chapter 5 for well-conditioned problems with simple constraint sets. Compared to alternative solution methods for QPs (e.g. active-set or interior-point schemes), first-order methods do not require the solution of a linear system of equations at every iteration, which is often a limiting factor for embedded platforms with modest computational capability. They have very simple computational structures which allow for efficient parallel computing and communication architectures. In addition, first-order methods have certain features, such as the possibility of deriving division-free variants, that make them amenable to fixed-point implementation.

A potential disadvantage of these methods is that they only exhibit asymptotic linear convergence compared to quadratic convergence for second-order methods, e.g. interior-point methods. However, it has been observed that for real-time control applications medium-accuracy solutions are often sufficient for good control performance [222], so it is not clear that this theoretical disadvantage has an important practical impact. A more important disadvantage is that the performance of first-order methods is largely affected by the condition number of the problem and the nature of the constraint set, which restricts their use to a smaller subset of MPC problems consisting of relatively well-conditioned problems with easy to project on feasible sets.

On the other hand, the simplicity of first-order methods invites theoretical analysis that has practical relevance. Whereas for interior-point methods the theoretical bounds on the number of iterations necessary to achieve a certain suboptimality gap are very far from their observed behaviour, for some first-order methods it is possible to derive practical convergence bounds that can be used for certifying solvers a priori. There has recently been a large amount of research activity in this area, both in the primal [114, 191] and dual [74, 162, 176, 193] domains. While all the works mentioned consider the problem of certification under exact arithmetic, this chapter analyses first-order methods to determine the maximum amount of error due to finite precision computations to guide low level implementation choices on embedded platforms. This kind of analysis is currently not fea-
Possible for interior-point methods, and decisions on the necessary precision in computations can only be made from empirical observations.

There are different first-order methods and optimization formulations that are suitable for handling different kinds of MPC problems. We will present a set of parameterized automatic generators of custom computing architectures for solving each type. For input-constrained problems, we describe architectures for Nesterov’s fast gradient method, and for state-constrained problems we consider architectures based on the alternating direction method of multipliers (ADMM). Even though these methods are conceptually very different, they share the same computational patterns and similar computing architectures can be used to implement them efficiently. These architectures are extended to support warm starting procedures and the projection operations required in the presence of soft constraints.

Since for a reliable operation using fixed-point arithmetic it is crucial to prevent overflow errors, we derive theoretical results that guarantee the absence of overflow in all variables of the fast gradient method. Furthermore, we present an error analysis of both the fast gradient method and ADMM under (inexact) fixed-point computations in a unified framework. This analysis underpins the numerical stability of the methods for hardware implementations and can be used to determine a priori the minimum number of bits required to achieve a given solution accuracy specification, resulting in minimal resource usage.

A set of design rules are presented for efficient implementation of the proposed methods, such as a scaling procedure for accelerating the convergence of ADMM and criteria for determining the size of the Lagrange multipliers. The proposed architectures are characterised in terms of the achievable performance as a function of the amount of resources available. As a proof of concept, generated solver instances are demonstrated for several linear-quadratic MPC problems, reporting achievable controller sampling rates in excess of 1 MHz, while the controller can still be implemented on a low cost embeddable device.

Outline

The chapter starts by describing the different methods and formulations for the different kinds of MPC problems in Section 6.1. The fixed-point theoretical analysis is the subject of Section 6.2, and the hardware architectures are presented in Section 6.3. The proposed architectures and analysis are evaluated in several case studies in Section 6.4. Section 6.5 summarises open questions in this area.

6.1 First-order solution methods

This section describes two different first-order optimization methods for solving the optimal control problem (4.2) efficiently. In particular, the primal fast gradient method (FGM) will be applied in cases where only input-constraints are present, i.e. $S = 0$ and $X_\Delta = \mathbb{R}^{n_x}$ with respect to the MPC problem setup introduced in Section 4.1. A dual
method based on the alternating direction method of multipliers (ADMM) will be applied for cases in which both state- and input-constraints are present.

6.1.1 Input-constrained MPC using the fast gradient method

The fast gradient method is an iterative solution method for smooth convex optimization problems first published by Nesterov in the early 80s [164], which requires the objective function to be strongly convex [25, §9.1.2]. The method can be applied to the solution of MPC problem (4.2) if the future state variables \( x_i \) are eliminated by expressing them as a function of the initial state, \( x \), and the future input sequence (so-called condensing [139]), resulting in the problem

\[
\min f(z; x) := \frac{1}{2} z^T H_F z + z^T \Phi x
\]

subject to \( z \in K \),

where \( z := (u_0, \ldots, u_{N-1}) \in \mathbb{R}^n \), \( n = N n_u \), the Hessian \( H_F \in \mathbb{R}^{n \times n} \) is positive definite under the assumptions in Section 4.1, and the feasible set is given as \( K := \mathbb{U} \times \ldots \times \mathbb{U} \). The current state only enters the gradient of the linear term of the objective through the matrix \( \Phi \in \mathbb{R}^{n \times n_u} \).

We consider the constant step scheme II of the fast gradient method in [165, §2.2.3]. Its algorithmic scheme for the solution of (6.1), optimized for parallel execution on parallel hardware, is given in Algorithm 3. Note that the state-independent terms \((I - \frac{1}{L} H_F), \frac{1}{L} \Phi\) and \((1 + \beta)\) can all be computed offline and that the product \( \frac{1}{L} \Phi x \) must only be evaluated once. The core operations in Algorithm 3 are the evaluation of the gradient (implicit in line 2) and the projection operator of the feasible set, \( \pi_K \), in line 3. Since for MPC problems the set \( K \) is the direct product of the \( N n_u \)-dimensional sets \( \mathbb{U} \), it suffices to consider \( N \) independent projections that can be performed independently. For the specific case of a box constraint on the control input, every such projection corresponds to \( n_u \) scalar projections on intervals, each computable analytically. In this case, the fast gradient method requires only multiplication and addition, which are considerably faster and use significantly fewer resources than division when implemented using digital circuits.

It can be inferred from [165, Theorem 2.2.3] that for every state \( x \), Algorithm 3 generates a sequence of iterates \( \{z_i\}_{i=1}^{I_{\text{max}}} \) such that the residuals \( f(z_i; x) - f^*(x) \) are bounded by

\[
\min \left\{ 1 - \sqrt{\frac{1 - \frac{4\kappa}{(L \sqrt{\kappa} + i)^2}}{\kappa}}, \frac{4\kappa}{(L \sqrt{\kappa} + i)^2} \right\} \cdot 2( f(z_0; x) - f^*(x) ),
\]

for all \( i = 0, \ldots, I_{\text{max}} \), where \( \kappa \) denotes the condition number of \( f \), or an upper bound of it, given by \( \kappa = L/\mu \), where \( L \) and \( \mu \) are a Lipschitz constant for the gradient of \( f \) and convexity parameter of \( f \), respectively. Note that the convexity parameter \( f \) for a strongly convex quadratic objective function as in (6.1) corresponds to the minimum eigenvalue of \( H_F \). Based on this convergence result, which states that the bound exhibits
Algorithm 3 Fast gradient method for the solution of MPC problem (6.1) at state \( x \) (optimized for parallel hardware)

**Require:** Initial iterate \( z_0 \in \mathbb{K}, y_0 = z_0 \), upper (lower) bound \( L (\mu > 0) \) on maximum (minimum) eigenvalue of Hessian \( H_F \), step size \( \beta = (\sqrt{L} - \sqrt{\mu}) / (\sqrt{L} + \sqrt{\mu}) \)

1: for \( i = 0 \) to \( I_{\text{max}} - 1 \) do
2: \( t_i := (I - \frac{1}{L} H_F) y_i - \frac{1}{L} \Phi x \)
3: \( z_{i+1} := \pi_\mathbb{K}(t_i) \)
4: \( y_{i+1} := (1 + \beta) z_{i+1} - \beta z_i \)
5: end for

the best of a linear and a sublinear rate, one can derive a certifiable and practically relevant iteration bound \( I_{\text{max}} \) such that the final residual is guaranteed to be within a specified level of suboptimality for all initial states arising from a bounded set [191]. It can further be proved that there is no other variant of a gradient method with better theoretical convergence [165], i.e. the fast gradient method is an optimal gradient method, in theory.

The fast gradient method is particularly attractive for application to MPC in embedded control system design due both to the relative ease of implementation and to the availability of strong performance certification guarantees. However, its use is limited to cases in which the projection operation \( \pi_\mathbb{K} \) is simple, e.g. in the case of box-constrained inputs. Unfortunately, the inclusion of state constraints changes the geometry of the feasible set \( \mathbb{K} \) such that the projection subproblem is as difficult as the original problem, since the constraints are no longer separable in \( u_i \). In the next section we therefore describe an alternative solution method in the dual domain that avoids these complications, though at the expense of some of the strong certification advantages.

### 6.1.2 Input- and state-constrained MPC using ADMM

In the presence of state constraints, first-order methods can be used again to solve the dual problem via Lagrange relaxation of the equality constraints.

For dual problems we do not work in the condensed format (6.1), but rather maintain the state variables \( x_k \) in the vector of decision variables \( z := (u_0, \ldots, u_{N-1}, x_0, \delta_0, \ldots, x_N, \delta_N) \in \mathbb{R}^n, n = N(n_u + n_x + |S|) + n_x + |S| \), resulting in the problem

\[
\begin{align*}
\hat{f}^*(x) &= \min_z f(z; x) := \frac{1}{2} z^T H_A z + z^T h \\
\text{subject to } z &\in \mathbb{K}, Fz = b(x).
\end{align*}
\]

The affine constraint \( Fz = b(x) \) models the dynamic coupling of the states \( x_k \) and \( u_k \) via the state update equation (4.3b), and is at the root of the difficulty in projecting the variables \( z \) onto the constraints in the fast gradient method.

If one imposes \((Q, Q_N) \in \mathbb{R}^{n \times n} \) to be positive definite\(^1\), the fast gradient method can be used again to solve the dual problem [193]. The algorithmic scheme for the case

---

\(^1\)The dual function is, in general, non-smooth when \( Q \) and \( Q_N \) are allowed to be positive semidefinite as in Section 4.1
Algorithm 4 Dual fast gradient method for the solution of MPC problem (6.3) at state \( x \)

Require: Initial iterate \( z_0 \in K, y_0 = z_0 \), upper (lower) bound \( L \) (\( \mu > 0 \)) on maximum (minimum) eigenvalue of Hessian \( H_A \), step size \( \beta = (\sqrt{L} - \sqrt{\mu})/(\sqrt{L} + \sqrt{\mu}) \)

1: for \( i = 0 \) to \( I_{\text{max}} - 1 \) do
2: \( t_i := -H_A^{-1}(h + F^T y_i) \)
3: \( z_{i+1} := \pi_K(t_i) \)
4: \( \nu_{i+1} := y_i + \frac{1}{\mu} (F z_{i+1} - b(x)) \)
5: \( y_{i+1} := (1 + \beta) \nu_{i+1} - \beta \nu_i \)
6: end for

when \( H_A \) is positive and diagonal is described in Algorithm 4. However, in this case the dual function is not strongly concave and consequently the convergence speed is severely affected. A quadratic regularizing term can be added to the Lagrangian to improve convergence, but this prevents the use of distributed operations for computing the gradient of the dual function (implicit in lines 2 and 3 of Algorithm 4), adding a significant computational overhead. We therefore seek an alternative approach in the dual domain.

The alternating direction method of multipliers (ADMM) [24] partitions the optimization variables into two (or more) groups to maintain the possibility of decoupled projection. In applying ADMM to the specific problem (6.1), we maintain an additional copy \( y \) of the original decision variables \( z \) and solve the problem

\[
\begin{align*}
    f^*(x) &= \min_{z,y} f(z, y; x) := \frac{1}{2} y^T H_A y + y^T h + I_A(y; x) + I_K(z) + \frac{\rho}{2} \|y - z\|^2 \\
    \text{subject to } z &= y,
\end{align*}
\]

where \((z, y) \in \mathbb{R}^{2n}\) contain copies of all input, state and slack variables. The functions \( I_A : \mathbb{R}^n \times \mathbb{R}^{nx} \to \{0, +\infty\} \) and \( I_K : \mathbb{R}^n \to \{0, +\infty\} \) are indicator functions for the sets described by the equality and inequality constraints, respectively, e.g.

\[
I_A(y; x) := \begin{cases} 0 & \text{if } Fy = b(x), \\ \infty & \text{otherwise}, \end{cases}
\]

where \( K := \mathbb{U} \times \ldots \times \mathbb{U} \times \mathbb{X}_\Delta \times \ldots \times \mathbb{X}_\Delta \). The current state \( x \) enters the optimization problem through (6.6). The inclusion of the regularizing term \((\rho/2)\|y - z\|^2\) has no impact on the solution to (6.4) (equivalently (6.3)) due to the compatibility constraint \( y = z \), but it does allow one to drop the smoothness and strong convexity conditions on the objective function, so that one can solve control problems with more general cost functions such as those with 1- or \( \infty \)-norm stage costs.

Note that there are many possible techniques for copying and partitioning of variables in ADMM. In the context of optimal control, the choice given in (6.4) results in attractive computational structures [171].
The dual problem for (6.4) is given by
\[
\max_{\nu} g(\nu) := \inf_{z,y} L_{\rho}(z,y,\nu) := \frac{1}{2} y^T H A y + y^T h + I_k(y; x) + I_k(z) + \nu^T (y - z) + \frac{\rho}{2} \|y - z\|^2.
\]
ADMM solves this dual problem using an approximate gradient method by repeatedly carrying out the steps
\[
\begin{align*}
y_{i+1} &:= \arg \min_y L_{\rho}(z_i, y, \nu_i), \quad (6.7a) 
z_{i+1} &:= \arg \min_z L_{\rho}(z, y_{i+1}, \nu_i), \quad (6.7b) 
\nu_{i+1} &:= \nu_i + \rho(y_{i+1} - z_{i+1}). \quad (6.7c)
\end{align*}
\]
The gradient of the dual function is approximated by the expression \(y_{i+1} - z_{i+1}\) in (6.7c), which employs a single Gauss-Seidel pass instead of a joint minimization to allow for decoupled computations. Choosing the regularity parameter \(\rho\) also as the step-length arises from Lipschitz continuity of the (augmented) dual function. There are at present no universally accepted rules for selecting the value of the penalty parameter however, and it is typically treated as a tuning parameter during implementation.

Our overall algorithmic scheme for ADMM for the solution of (6.4) based on the sequence of operations (6.7a)–(6.7c), optimized for parallel execution on parallel hardware, is given in Algorithm 5. The core computational tasks are the equality-constrained optimization problem (6.7a) and the inequality-constrained, but separable, optimization problem (6.7b).

In the case of the equality-constrained minimization step (6.7a), a solution can be computed from the KKT conditions by solving the linear system
\[
\begin{bmatrix}
H_A + \rho I & F^T \\
F & 0
\end{bmatrix}
\begin{bmatrix}
y_{i+1} \\
\lambda_{i+1}
\end{bmatrix}
= \begin{bmatrix}
-h - \nu_i + \rho z_i \\
b(x)
\end{bmatrix}.
\]
Note that only the vector \(y_{i+1}\), and not the multiplier \(\lambda_{i+1}\), arising from the solution of this linear system is required for our ADMM method. The most efficient method to solve for \(y_{i+1}\) is to invert the (fixed) KKT matrix offline, i.e. to compute
\[
\begin{bmatrix}
M_{11} & M_{12} \\
M_{12}^T & M_{22}
\end{bmatrix}
= \begin{bmatrix}
H_A + \rho I & F^T \\
F & 0
\end{bmatrix}^{-1},
\]
and then to obtain \(y_{i+1}\) online from \(y_{i+1} = M_{11} (-h - \nu_i + \rho z_i) + M_{12} b(x)\) as in Line 2 of Algorithm 5. Observe that the product \(M_{12} b(x)\) needs to be evaluated only once, and that this matrix is always invertible when \(\rho > 0\) since \(F\) has full row rank.

The inequality-constrained minimization step (6.7b) results in the projection operation in Line 3 of Algorithm 5. In the presence of soft state constraints, this operation requires independent projections onto a truncated two-dimensional cone, which can be efficiently parallelized and require no divisions. We describe efficient implementations of this projection operation in parallel hardware in Section 6.3.
Algorithm 5 ADMM for the solution of MPC problem (6.1) at state $x$ (optimized for parallel hardware)

Require: Initial iterate $z_0 = z^* - \nu_0 = \nu^* - \rho$, where $z^*$ and $\nu^*$ are the shifted solutions at the previous time instant (see Section 6.3), and $\rho$ is a constant power of 2.

1: for $i = 0$ to $I_{\text{max}} - 1$ do
2:   $y_{i+1} := M_{11}(-h + \rho z_i - \nu_i) + M_{12}b(x)$
3:   $z_{i+1} := \pi_{\mathcal{K}}(y_{i+1} + \frac{1}{\rho} \nu_i)$
4:   $\nu_{i+1} := \rho y_{i+1} + \nu_i - \rho z_{i+1}$
5: end for

This variant of ADMM is known to converge; see [17, §3.4; Prop. 4.2] for general convergence results. More recently, a bound on the convergence rate was established in [19], where it was shown that the error in ADMM, for a different error function, decreases as $1/i$, where $i$ is the number of iterations. This result still compares unfavorably relative to the known $1/i^2$ convergence rate for the fast gradient method in the dual domain. However, the observed convergence behavior of ADMM in practice is often significantly faster than for the fast gradient method [24].

6.1.3 ADMM, Lagrange multipliers and soft constraints

Despite its generally excellent empirical performance, ADMM can be observed to converge very slowly in certain cases. In particular, for MPC problems in the form (6.1), convergence may be very slow in those cases where there is a large mismatch between the magnitude of the optimal Lagrange multipliers $\nu^*$ for the equality constraint (6.5) and the magnitude of the primal iterates ($z_i, y_i$). The reason is evident from the ADMM multiplier update step (6.7c); the existence of very large optimal multipliers $\nu^*$ necessitates a large number of ADMM iterations when the difference $(z_i - y_i)$ remains small at each iteration and $\rho \approx 1$.

This effect is of particular concern for MPC problem instances with soft constraints. If one denotes by $z_\delta$ those components of $z$ associated with the slack variables $\{\delta_1, \ldots, \delta_N\}$ (with similar notation for $y_\delta$), then the objective function (6.4) features a term $\sigma_1 \cdot 1^T y_\delta$, with the exact penalty term $\sigma_1$ typically very large. The equality constraints (6.5) include the matching condition $z_\delta - y_\delta = 0$, with associated Lagrange multiplier $\nu_\delta$. Recalling the usual sensitivity interpretation of the optimal multiplier $\nu_{\delta}^*$, one can conclude that $\nu_{\delta}^* \approx \sigma_1 \cdot 1$ in the absence of unusual problem scaling$^2$.

For soft constrained problems, we avoid this difficulty by rescaling those components of the matching condition (6.5) to the equivalent condition $(1/\sigma_1)(z_\delta - y_\delta) = 0$, which results in a rescaling of the associated optimal multipliers to $\nu_{\delta}^* \approx 1$. The aforementioned convergence difficulties due to excessively large optimal multipliers are then avoided.

$^2$If one sets the regularization parameter $\rho = 0$ in (6.4) and $\sigma_2 = 0$, then it can be shown that this approximation becomes exact.
6.2 Fixed-point aspects of first-order solution methods

This section starts by motivating the use of fixed-point arithmetic from a hardware efficiency perspective and then isolates potential error sources under this arithmetic. We concentrate on two types of errors. For overflow errors we provide analysis to guarantee that they cannot occur in the fast gradient method, whereas for arithmetic round-off errors we prove that there is a converging upper bound on the total incurred error in either of the two methods. The results we obtain hold under the assumptions in Section 6.2.3 and guarantee reliable operation of first-order methods on fixed-point platforms.

6.2.1 The performance gap between fixed-point and floating-point arithmetic

Modern computing platforms must allow for a wide range of applications that operate on data with potentially large dynamic range, i.e. the ratio of the smallest to largest number to be represented. For general purpose computing, floating-point arithmetic provides the necessary flexibility. A floating-point number consists of a sign bit, an exponent and a mantissa. The exponent value moves the binary point with respect to the mantissa. The dynamic range – the ratio of the smallest to largest representable number – grows doubly exponentially with the number of exponent bits, therefore it is possible to represent a wide range of numbers with a relatively small number of bits. However, because two operands can have different exponents it is necessary to perform denormalization and normalization operations before and after every addition or subtraction, leading to greater resource usage and longer delays.

In contrast, fixed-point numbers have a fixed number of bits for the integer and fraction fields, i.e. the exponent does not change with time and it does not have to be stored. Fixed-point hardware is the same as for integer arithmetic, hence the circuitry is simple and fast, but the representable dynamic range is limited. However, if the dynamic range in the data is also limited and fixed, a 32-bit fixed-point processor can provide more precision than a 32-bit floating-point processor because there are no bits wasted for the exponent.

Figure 3.5 (p. 51) in Section 3.1.4 showed that a floating-point adder includes more hardware blocks than just the block performing the binary addition. In FPGAs the shift operations in the floating-point adder are especially problematic. Because there is no explicit hard support for this operation it has to be implemented using reconfigurable resources, which results in signals having to traverse many reconfigurable blocks, incurring long delays. In contrast, FPGAs do have explicit hardware for supporting the carry chains in binary integer additions, hence this operation incurs a very small delay. Table 6.1 shows the resource usage and arithmetic delay of different adder implementations in an FPGA. Approximately one order of magnitude saving in resources and one order of magnitude reduction in delay are possible by moving to a fixed-point implementation. For multipliers, the difference is not as large but it is still very significant. Furthermore, there is still a lack of floating-point support in some high-level FPGA design flows, such as LabVIEW...
Table 6.1: Resource usage and input-output delay of different fixed-point and floating-point adders in Xilinx FPGAs running at approximately the same clock frequency. 53 and 24 fixed-point bits can potentially give the same accuracy as double and single precision floating-point, respectively.

<table>
<thead>
<tr>
<th></th>
<th>Registers</th>
<th>LUTs</th>
<th>Latency</th>
</tr>
</thead>
<tbody>
<tr>
<td>double</td>
<td>1046</td>
<td>911</td>
<td>14</td>
</tr>
<tr>
<td>float</td>
<td>557</td>
<td>477</td>
<td>11</td>
</tr>
<tr>
<td>fixed53</td>
<td>53</td>
<td>53</td>
<td>1</td>
</tr>
<tr>
<td>fixed24</td>
<td>24</td>
<td>24</td>
<td>1</td>
</tr>
</tbody>
</table>

FPGA, due to the instantiation of floating-point units quickly exhausting the capacity of modest size devices.

In terms of other devices in the embedded computing spectrum, the cost of fixed-point DSPs is in the region of five times less than floating-point devices for the same operations per second capability. Of course, the power consumption is also significantly smaller. In the microcontroller domain, there exist 32-bit fixed-point devices for less than one US dollar.

6.2.2 Error sources in fixed-point arithmetic

The benefits of fixed-point arithmetic motivate its use in first-order methods to realise fast and efficient implementations of Algorithms 3 and 5 on FPGAs or other low cost and low power devices with no floating-point support, such as embedded microcontrollers, fixed-point DSPs or PLCs. However, reduced precision representations and fixed-point computations incur several types of errors that must be accounted for. These include:

**Quantization Errors**

Finite representation errors arise when converting the problem and algorithm data from high precision to reduced precision data formats. Potential consequences include loss of problem convexity, change of optimal solution and a lack of feasibility with respect to the original problem.

**Overflow Errors**

Overflow errors occur whenever the number of bits for the integer part in the fixed-point representation is too small, and can cause unpredictable behavior of the algorithm.

**Arithmetic Errors**

Unlike with floating-point arithmetic, fixed-point addition and subtraction operations involve no round-off error provided there is no overflow and the result has the same number of fraction bits as the operands [223]. For multiplication, the exact product of two numbers with \( b \) fraction bits can be represented using \( 2b \) fraction bits, hence a \( b \)-bit truncation
of a 2’s complement number incurs a round-off error bounded from below by \(-2^{−b}\). Recall that in 2’s complement arithmetic, truncation incurs a negative error both for positive and negative numbers.

We focus on overflow and arithmetic errors next and derive results which hold for the following setup and assumptions.

### 6.2.3 Notation and assumptions

We will use \(\hat{\cdot}\) throughout in order to distinguish quantities in a fixed-point representation from those in an exact representation and under exact arithmetic. Throughout, we assume for simplicity that all variables and problem data are represented using the same number of fraction bits \(b\). We further assume that the feasible sets under finite precision satisfy \(\hat{K} \subseteq K\), so that solutions in fixed point arithmetic do not produce infeasibility in the original problem due to quantization error.

We conduct separate analyses of both overflow and arithmetic errors for the fast gradient method (Algorithm 3) and ADMM (Algorithm 5). In both cases, the central requirement is to choose the number of fraction bits \(b\) large enough to ensure satisfactory numerical behavior. We therefore employ two different sets of assumptions depending on the numerical method in question:

**Assumption 1** (Fast Gradient Method / Algorithm 3). *The number of fractions bits \(b\) and a constant \(c \geq 1\) are chosen large enough such that*

1. The matrix

   \[
   H_n = \frac{1}{c \cdot \lambda_{\text{max}}(\hat{H}_F)} \cdot \hat{H}_F,
   \]

   has a fixed-point representation \(\hat{H}_n\) with all of its eigenvalues in the interval \((0, 1]\), where \(\hat{H}_F\) is the fixed-point representation of the Hessian \(H_F\), with \(\lambda_{\text{max}}(\hat{H}_F)\) its maximum eigenvalue.

2. The fixed-point step size \(\hat{\beta}\) satisfies

   \[
   1 > \hat{\beta} \geq \left(\sqrt{\kappa(\hat{H}_n)} - 1\right) \left(\sqrt{\kappa(\hat{H}_n)} + 1\right)^{-1} \geq 0,
   \]

   where \(\kappa(\hat{H}_n)\) is the condition number of \(\hat{H}_n\).

**Assumption 2** (ADMM / Algorithm 5). *The number of fractions bits \(b\) is chosen large enough such that*

1. The matrix

   \[
   \begin{pmatrix}
   \hat{M}_{11} & \hat{M}_{12} \\
   \hat{M}_{T12} & \hat{M}_{22}
   \end{pmatrix}
   \begin{pmatrix}
   \rho I & \hat{F}^T \\
   \hat{F} & 0
   \end{pmatrix}
   \]

   has a fixed-point representation \(\hat{\cdot}\) with all of its eigenvalues in the interval \((0, 1]\), where \(\hat{\cdot}\) is the fixed-point representation of the variables under quantization.
is positive semidefinite, where $\rho$ is chosen such that it is exactly representable in $b$ bits.

ii) The quantization errors in the matrix $\hat{F}$, which is derived from the linear model of the plant (refer to Section 4.2.1), are insignificant compared to the errors arising from model uncertainty.

Observe that it is always possible to select $b$ sufficiently large to satisfy all of the preceding assumptions, implying that the above conditions represent a lower bound on the number of fraction bits required in a fixed-point implementation of our two algorithms to ensure that our stability results are valid. Assumptions 1.(i) and 2.(i) ensure that the objective functions (6.1) (for the fast gradient method) and (6.4) (for ADMM) remain strongly convex and convex, respectively, despite any quantization error.

In the case of the fast gradient method, Assumption 1.(ii) guarantees that the true condition number of $\hat{H}_n$ is not underestimated, in which case the convergence result of the fast gradient method in (6.2) would be invalid. In fact, the assumption ensures that the effective condition number for the convergence result is given by

$$
\kappa_n = \left( \frac{1 + \hat{\beta}}{1 - \hat{\beta}} \right)^2 \geq \kappa(\hat{H}_n).
$$

(6.8)

6.2.4 Overflow errors

In order to avoid overflow errors in a fixed-point implementation, the largest absolute values of the iterates’ and intermediate variables’ components must be known or upper-bounded a priori in order to determine the number of bits required for their integer parts. For the static problem data $(I - \hat{H}_n), \hat{\Phi}_n, 1 + \hat{\beta}, \hat{\beta}, \hat{M}_{11}, \text{ or } \hat{M}_{12}$, the number of integer bits is easily determined by the maximum absolute value in each expression.

Overflow Error Bounds in the Fast Gradient Method

In the case of the fast gradient method, it is possible to bound analytically the largest absolute values of all of the dynamic data, i.e. the variables that change with every iteration. We will denote by $\hat{\Phi}_n$ the fixed-point representation of

$$
\hat{\Phi}_n = \frac{1}{c \cdot \lambda_{\max}(\hat{H}_F)} \cdot \Phi.
$$

We summarize the upper bounds on variables appearing in the fast gradient method in the following proposition.

**Proposition 2.** If problem (6.1) is solved by the fast gradient method using the appropriately adapted Algorithm 3, then the largest absolute values of the iterates and intermediate variables are given by

$$
\|\hat{z}_{i+1}\|_{\infty} \leq \bar{z} := \max \{ \|\hat{z}_{\min}\|_{\infty}, \|\hat{z}_{\max}\|_{\infty} \},
$$
\[
\|\hat{y}_{i+1}\|_\infty \leq \bar{y} := \hat{z} + \hat{\beta}\|\hat{\max} - \hat{\min}\|_\infty,
\]
\[
\|(I - \hat{H}_n) \hat{y}_i\|_\infty \leq \bar{y}_{\text{inter}} := \|I - \hat{H}_n\|_\infty \cdot \bar{y},
\]
\[
\|\hat{x}\|_\infty \leq \bar{x} := \max_{x \in \hat{X}_0}\|x\|_\infty,
\]
\[
\|\hat{\Phi}_n \hat{x}\|_\infty \leq \bar{h} := \|\hat{\Phi}_n\|_\infty \cdot \bar{x}, \quad \text{and}
\]
\[
\|t_i\|_\infty \leq \bar{\ell} := \bar{y}_{\text{inter}} + \bar{h},
\]

for all \(i = 0, 1, \ldots, I_{\text{max}} - 1\). The set \(\hat{X}_0\) is chosen such that for every state in exact arithmetic \(x \in X_0\) we have \(\hat{x} \in \hat{X}_0\).

\textbf{Proof.} Follows from interval arithmetic and properties of the vector/matrix \(\|\cdot\|_\infty\)-norm. \qed

Note that normalization of the objective as introduced in Section 6.2.3 has no effect on the maximum absolute values of the iterates. Furthermore, the bound in (6.9) also applies for the intermediate elements/cumulative sums in the evaluation of the matrix-vector product. Observe that most of the bounds stated in Proposition 2 are tight.

\section*{Overflow Error Bounds in ADMM}

If problem (6.4) is solved using ADMM via Algorithm 5, then we do not know of any general method to upper bound the Lagrange multiplier iterates \(\nu_i\) analytically, and consequently are unable to establish analytic upper bounds on all expressions involving \textit{dynamic} data. In this case, one must instead estimate the undetermined upper bounds through simulation and add a safety factor when allocating the number of integer bits. As a result, with ADMM, we trade analytical guarantees on numerical behavior for the capability to solve more general problems.

\subsection*{6.2.5 Arithmetic round-off errors}

We next derive an upper bound on the deviation of an optimal solution \(\hat{z}^*\) produced via a fixed-point implementation of either Algorithm 3 or 5 from the optimal solutions produced from the same algorithms implemented using exact arithmetic. In both cases, we denote by \(\hat{z}_i\) a \textit{fixed-point} iterate. We wish to relate these iterates to the iterates \(z_i\) generated under \textit{exact arithmetic}, by establishing a bound in the form
\[
\|\hat{z}_i - z_i\| = \|\eta_i\| \leq \Delta_i
\]
with \(\lim_{i \to \infty} \Delta_i\) finite, where \(\eta_i := \hat{z}_i - z_i\) is the solution error attributable to arithmetic round-off error up to the \(i^{th}\) iteration. Consequently, we can show that inaccuracy in the computed optimal solution induced by arithmetic errors in either algorithm are bounded, which is a crucial prerequisite for reliable operation of first-order methods on fixed-point platforms.
In both cases, we use a control-theoretic approach based on standard Lyapunov methods to derive bounds on the solution error arising specifically from fixed-point arithmetic error. For simplicity of exposition, in the following analysis we consider only those errors arising from arithmetic errors (occurring at all iterations) and neglect errors arising from quantization of the problem data (occurring only once). This choice does not alter substantively the results presented for either algorithm. Our approach is in contrast to (and more direct than) other approaches to error accumulation analysis in the fast gradient method such as [9,201], which consider inexact gradient computations but do not address arithmetic round-off errors explicitly. In the case of ADMM, we are not aware of any existing results relating to error accumulation in fixed-point arithmetic.

**Stability of arithmetic errors in the primal fast gradient method**

We consider first the numerical stability of the fast gradient method, by examining in detail the arithmetic error introduced at each step of a fixed-point implementation of Algorithm 3.

At iteration $i$, the error in line 2 of Algorithm 3 is given by

$$
\hat{t}_i - t_i = (I - \hat{H}_n)(\hat{y}_i - y_i) + \epsilon_{t,i},
$$

where $\epsilon_{t,i}$ is a vector of errors from the matrix-vector multiplication. Since there are $n$ round-off errors in the computation of every component, $\epsilon_{t,i}$ is componentwise in the interval $[-n2^{-b},0]$.

For the projection in line 3, and recalling that $\hat{K} \subseteq K$ is a box, no arithmetic error is introduced. Indeed, one can easily verify that the error $\hat{t}_i - t_i$ can only be reduced by multiplication with a diagonal matrix $\text{diag}(\epsilon_{\pi,i})$, with $\epsilon_{\pi,i}$ componentwise in the interval $[0,1]$.

Finally, in line 4, the error induced by fixed-point arithmetic is

$$
\hat{y}_{i+1} - y_{i+1} = (1 + \hat{\beta})\eta_{i+1} - \hat{\beta}\eta_i + \epsilon_{y,i},
$$

where two scalar-vector multiplications incur error $\epsilon_{y,i}$ with components in $[-2^{-b},2^{-b}]$ (addition and subtraction). Defining the initial error residual terms $\eta_{-1} = \eta_0 = \hat{z}_0 - z_0$, and setting $\hat{z}_0 - z_0 = \hat{y}_0 - y_0$, one can derive the two-step recurrence

$$
\eta_{i+1} = \text{diag}(\epsilon_{\pi,i})(I - \hat{H}_n)(\eta_i + \hat{\beta}(\eta_i - \eta_{i-1}) + \epsilon_{y,i-1}) + \epsilon_{t,i}
$$

for the accumulated arithmetic error at each iteration. Note that the error $\eta_i$ at each iteration is inherently bounded by the box $\hat{K}$. However, in the absence of the projection operation of line 3 and the associated error truncation, these errors remain bounded. To show this, we can express the evolution of the arithmetic error using the two-step
and then show that this linear system is stable. Recalling Assumption 1, which bounds the eigenvalues of \( \hat{H}_n \) in the interval \((0, 1]\) and \( \hat{\beta} \) in the interval \([0, 1)\), we can use the following result:

**Lemma 1.** Let \( C \) be any symmetric positive definite matrix with maximum eigenvalue less than or equal to one. For every constant \( \gamma \) in the interval \([0, 1]\) the matrix

\[
M = \begin{bmatrix} (1 + \gamma)(I - C) & -\gamma(I - C) \\ I & 0 \end{bmatrix}
\]

is Schur stable, i.e. its spectral radius \( \rho(M) \) is less than one.

**Proof.** Assume the eigenvalue decomposition \( I - C = V^T \Lambda V \), with \( \Lambda \) diagonal with entries \( \lambda_i \in [0, 1) \). The eigenvalues of \( M \) are unchanged by left- and right-multiplication by \( \begin{bmatrix} V \\ V \end{bmatrix} \) and its transpose. It is therefore sufficient to examine instead the spectral radius of

\[
M_D = \begin{bmatrix} (1 + \gamma)\Lambda & -\gamma \Lambda \\ I & 0 \end{bmatrix}.
\]

Since this matrix has exclusively diagonal blocks, its eigenvalues coincide with those of the two-by-two submatrices

\[
M_{D,i} = \begin{bmatrix} (1 + \gamma)\lambda_i & -\gamma \lambda_i \\ 1 & 0 \end{bmatrix}, \quad \text{for } i = 1, \ldots, n,
\]

and it is sufficient to prove that every such submatrix has spectral radius less than one. Note that the eigenvalues of \( M_{D,i} \) are the roots of the characteristic equation

\[
\mu^2 - (1 + \gamma)\lambda_i \mu + \lambda_i \gamma = 0.
\]

It is easily verified that a sufficient condition for any quadratic equation in the form

\[
x^2 + 2bx + c = 0
\]

to have roots strictly inside the unit disk is for its coefficients to satisfy i) \(|b| < 1\), ii) \(c < 1\) and iii) \(2|b| < c + 1\). For the eigenvalue solutions to (6.11), this amounts to i)
\[(1 + \gamma)\lambda_i/2 < 1, \text{ ii) } \lambda_i \gamma < 1 \text{ and iii) } (1 + \gamma)\lambda_i < \gamma \lambda_i + 1. \text{ All three conditions are easily confirmed for the case } \lambda_i \in [0, 1), \gamma \in [0, 1].\]

**Stability of arithmetic errors in ADMM**

As in the preceding section, for ADMM one can analyze in detail the arithmetic error introduced at each step of a fixed-point implementation of Algorithm 5.

Defining \(\eta_i := \hat{z}_i - z_i, \gamma_i := \hat{\nu}_i - \nu_i\), a similar analysis to that of the preceding section produces the two-step error recurrence

\[
\begin{bmatrix}
\eta_{i+1} \\
\gamma_{i+1}
\end{bmatrix}
=:
\begin{bmatrix}
\rho \text{ diag} (\epsilon_{\pi,i}) \hat{M}_{11} & -\text{diag} (\epsilon_{\pi,i}) (\hat{M}_{11} - \frac{1}{\rho} I) \\
\rho^2 \hat{M}_{11} (I - \text{diag} (\epsilon_{\pi,i})) (I - \rho \hat{M}_{11}) (I - \text{diag} (\epsilon_{\pi,i})) & \epsilon_{y,i} I
\end{bmatrix}
\begin{bmatrix}
\eta_i \\
\gamma_i
\end{bmatrix}
+ \begin{bmatrix}
\text{diag} (\epsilon_{\pi,i}) \\
\rho (I - \text{diag} (\epsilon_{\pi,i}))
\end{bmatrix}
\begin{bmatrix}
\epsilon_{y,i} I \\
\epsilon_{\nu,i}
\end{bmatrix},
\]

(6.12)

where: \(\epsilon_{y,i} \in [-n2^{-b}, 0]^n\) is a vector of multiplication errors arising from Algorithm 5, line 2; \(\epsilon_{\pi,i} \in [0, 1]^n\) is a vector of error reduction scalings arising from the projection operation in line 3; and \(\epsilon_{\nu,i} \in [-2^{-b}, 2^{-b}]^n\) a vector of multiplication errors arising from 4 with \(\epsilon_{\nu,-1} = 0\). Note that one can show that even when \(\hat{K}\) is not a box in the presence of soft state constraints, the error can only be reduced by the projection operation. The initial iterates of the recurrence relation are \(\eta_{-1} = \eta_0\), where \(\eta_0 := \hat{z}_0 - z_0\).

As in the case of the fast gradient method, these arithmetic errors are inherently bounded by the constraint set \(\hat{K}\). In the absence of these bounding constraints (so that \(\text{diag} (\epsilon_{\pi,i}) = I\), one can still establish that the arithmetic errors are bounded via examination of the eigenvalues of the matrix

\[
\mathcal{N} := \begin{bmatrix}
\rho \hat{M}_{11} & -(\hat{M}_{11} - \frac{1}{\rho} I) \\
0 & 0
\end{bmatrix}.
\]

(6.13)

Recalling Assumption 2, we have the following result:

**Lemma 2.** The matrix \(\mathcal{N}\) in (6.13) is Schur stable for any \(\rho > 0\).

**Proof.** The eigenvalues of (6.13) are either 0 or \(\rho \lambda_i (\hat{M}_{11})\), so it is sufficient to show that the symmetric matrix \(\hat{M}_{11}\) satisfies \(\rho \|\hat{M}_{11}\| < 1\). Recalling that

\[
\begin{bmatrix}
\hat{M}_{11} & \hat{M}_{12} \\
\hat{M}_{12}^T & \hat{M}_{22}
\end{bmatrix}
= \begin{bmatrix}
\hat{Z} & \hat{F}^T
\end{bmatrix}^{-1} \begin{bmatrix}
\hat{Z} & \hat{F}
\end{bmatrix},
\]

where \(\hat{Z} := \hat{H}_A + \rho I > 0\), the matrix inversion lemma provides the identity

\[
\hat{M}_{11} = \hat{Z}^{-\frac{1}{2}} \left[ I - \hat{Z}^{-\frac{1}{2}} \hat{F}^T (\hat{F} \hat{Z}^{-\frac{1}{2}} \hat{F}^T)^{-1} \hat{F} \hat{Z}^{-\frac{1}{2}} \right] \hat{Z}^{-\frac{1}{2}}
\]
where $\hat{P}$ is a projection onto the kernel of $\hat{F} \hat{Z}^{-\frac{1}{2}}$, hence $\|\hat{M}_{11}\| \leq \|\hat{Z}^{-\frac{1}{2}}\|\|\hat{P}\|\|\hat{Z}^{-\frac{1}{2}}\| = \|\hat{Z}^{-1}\|$. It follows that

$$\rho\|\hat{M}_{11}\| \leq \rho\|\hat{H}_A + \rho I\|^{-1} \leq \rho \frac{1}{\lambda_{\text{min}}(\hat{H}_A) + \rho} \leq 1,$$

where $\lambda_{\text{min}}(\hat{H}_A)$ is the smallest eigenvalue of the positive semidefinite matrix $\hat{H}_A$. If $\hat{H}_A$ is actually positive definite, then the preceding inequality is strict and the proof is complete.

Otherwise, to prove that the inequality is strict we must show that $1/\rho$ is not an eigenvalue for $\hat{M}_{11}$ (which is positive semidefinite by virtue of (6.14)). Assume the contrary, so that there exists some eigenvector $v$ of $\hat{M}_{11}$ with eigenvalue $1/\rho$, and some additional (arbitrary) vector $q$ that solves the linear system

$$\begin{bmatrix} v \\ q \end{bmatrix} = \begin{bmatrix} \hat{Z} & \hat{F}^T \\ \hat{F} & 0 \end{bmatrix}^{-1} \begin{bmatrix} \rho \cdot v \\ 0 \end{bmatrix}.$$

Any solution must then satisfy both $\hat{H}_A v \in \text{Im}(\hat{F}^T)$ and $v \in \text{Ker}(\hat{F})$. Consequently $v^T \hat{H}_A v = 0$, which requires $v \in \text{Ker}(\hat{H}_A)$ since $\hat{H}_A$ is positive semidefinite. Recall that any such $v$ can be decomposed into $v = (u_0, \ldots, u_{N-1}, x_0, \delta_0, \ldots, x_N, \delta_N)$. If the quadratic penalty for each $\delta_i$ is positive definite, then $v \in \text{Ker}(\hat{H}_A)$ requires each $\delta_i = 0$.

Since $\hat{F}v = 0$, the remaining components of $v$ must correspond to a sequence of state and inputs compatible with the system dynamics in (4.3b), starting from an initial state $x_0 = 0$. Any solution $v \neq 0$ would then require at least one component $u_i \neq 0$. Then $v^T \hat{H}_A v \geq u_i^T R u_i > 0$ since $R$ is assumed positive definite, a contradiction.

Arithmetic Error Bounds for the Fast Gradient Method and ADMM

Finally, for both the fast gradient method and ADMM we can use Lemmas 1 and 2 to establish an upper bound on the magnitude of error $\eta_i$ for any arithmetic round-off errors that might have occurred up to iteration $i$.

**Proposition 3.** Let $b$ be the number of fraction bits and $n$ be the dimension of the decision vector. Consider the error dynamics due to arithmetic round-off in (6.10) or in (6.12), assuming no error reduction from projection. The magnitude of any accumulation of round-off errors up to iteration $i$, $\|\eta_i\| = \|\hat{z}_i - z_i\|$, is upper-bounded by

$$\bar{\eta}_i = \|EA^i\| \left\| \begin{bmatrix} \eta_0 \\ \eta_0 \end{bmatrix} \right\| + 2^{-b} \sqrt{n(1+n^2)} \sum_{k=0}^{i-1} \|EA^{i-1-k}B\|$$

for all $i = 0, \ldots, I_{\text{max}} - 1$, where matrix $E = \begin{bmatrix} I & 0 \end{bmatrix}$.
Proof. From the one-step recurrence (6.10) or (6.12) we find that

$$\xi_i = A^i \xi_0 + \sum_{k=0}^{i-1} A^{i-1-k} B v_k, \quad i = 0, 1, \ldots, I_{\text{max}} - 1,$$

such that the result is obtained from applying the properties of the matrix norm. Observe that $2^{-b} \sqrt{n(1+n^2)}$ is the maximum magnitude of $v_k$ for any $k = 0, \ldots, i - 1$.

Since the matrix $A$ is Schur stable, the bound in (6.15) converges. Indeed, the effect of the initial error $\xi_0$ decays according to

$$\|EA^i\| \propto \rho(A)^i,$$

(6.16)

whereas the term driven by arithmetic round-off errors in every iteration behaves according to

$$\sum_{k=0}^{i-1} \|EA^{i-1-k}B\| \propto \frac{1}{1 - \rho(A)} - \frac{\rho(A)^i}{1 - \rho(A)}.$$

(6.17)

This result can be used to choose the number of bits $b$ a priori to meet accuracy specifications on the minimiser.

6.3 Embedded hardware architectures for first-order solution methods

Amdahl’s law [4] states that the potential acceleration of an optimization algorithm through parallelization is limited by the fraction of sequential dependencies in the algorithm. First-order optimization methods such as the fast gradient method and ADMM have a smaller number of sequential dependencies than interior-point or active-set methods. In fact, a very large fraction of the computation involves a single readily parallelisable matrix-vector multiplication, hence the expected benefit from parallelisation is substantial. Our implementations of both the fast gradient method (Algorithm 3) and ADMM (Algorithm 5) differ somewhat from more conventional implementations of these methods in order to minimise sequential dependencies. Observe that in both of our algorithms, the computations of the individual vector components are independent and the only communication occurs during matrix-vector multiplication. This allows for efficient parallelisation given the custom computing and communication architectures discussed next. Specifically, we describe a tool that takes as inputs the data type, number of bits, level of parallelism and the delays of an adder/subtractor ($l_A$) and multiplier ($l_M$) and automatically generates a digital architecture described in the VHDL hardware description language.
Figure 6.1: Fast gradient compute architecture. Boxes denote storage elements and dotted lines represent $Nn_u$ parallel vector links. The dot-product block $\hat{v}^T\hat{w}$ and the projection block $\pi_{\hat{K}}$ are depicted in Figures 6.2 and 6.4 in detail. FIFO stands for first-in first-out memory and is used to hold the values of the current iterate for use in the next iteration. In the initial iteration, the multiplexers allow $\hat{x}$ and $\Phi_n$ through and the result $\hat{\Phi}_n\hat{x}$ is stored in memory. In the subsequent iterations, the multiplexers allow $\hat{y}_i$ and $I - \hat{H}_n$ through and $\hat{\Phi}_n\hat{x}$ is read from memory.

6.3.1 Hardware architecture for the primal fast gradient method

For a fixed-point data type, the parameterised architecture implementing Algorithm 3 for problem (6.1) is depicted in Figure 6.1. The matrix-vector multiplication is computed in the block labeled $\hat{v}^T\hat{w}$, which is shown in detail in Figure 6.2. It consists of an array of $Nn_u$ parallel multipliers followed by an adder reduction tree of depth $\lceil \log_2 Nn_u \rceil$. The architecture for performing the projection operation on the set $\hat{K}$ is shown in Figure 6.4. It compares the incoming value with the upper and lower bounds for that component. Based on the result, the component is either saturated or left unchanged.

The amount of parallelism in the circuit is parameterised by the parameter $P$. In Figure 6.1, $P = 1$, meaning that there is parallelism within each dot-product but the $Nn_u$ dot-products required for matrix-vector multiplication are computed sequentially. If the level of parallelization is increased to $P = 2$, there will be two copies of the shaded circuit in Figure 6.1 operating in parallel, one computing the odd components of $\hat{y}_i$ and $\hat{z}_i$, the other computing the even. The different blocks communicate through a serial-to-parallel shift register that accepts $P$ serial streams and outputs $Nn_u$ parallel values for matrix-vector multiplication. These $Nn_u$ values are the same for all blocks. It takes $\lceil Nn_u P \rceil$ clock cycles to have enough data to start a new iteration, hence the number of clock cycles needed to compute one iteration of the fast gradient method for $P \in \{1, \ldots, Nn_u\}$ is

$$L_F := \lceil Nn_u P \rceil + l_A \lceil \log_2 Nn_u \rceil + 2l_M + 3l_A + 1. \quad (6.18)$$

Expression (6.18) suggests that there will be diminishing returns to parallelization – a consequence of Amdahl’s law. However, (6.18) also suggests that if there are enough
Figure 6.2: Hardware architecture for dot-product block with parallel tree architecture (left), and hardware support for warm-starting (right). Support for warm-starting adds one cycle delay. The last entries of the vector are padded with $w_N$, which can be constant or depend on previous values.

Figure 6.3: ADMM compute architecture. Boxes denote storage elements and dotted lines represent $n_A$ parallel vector links. The dot-product block $\hat{v}^T\hat{w}$ and the projection block $\pi_K$ are depicted in Figures 6.2 and 6.5 in detail. FIFO stands for first-in first-out memory and is used to hold the values of the current iterate for use in the next iteration. In the initial iteration, the multiplexers allow In the initial iteration, the multiplexers allow $x$ and $M_{12}$ through and the result $M_{12}b(x)$ is stored in memory.

resources available, the effect of the problem size on increased computational delay is only logarithmic in the worst case. As Moore’s law continues to deliver devices with greater transistor densities, the possibility of implementing algorithms in a fully parallel fashion for medium size optimization problems is becoming a reality.

6.3.2 Hardware architecture for ADMM

Algorithm 5 shares the same computational patterns with Algorithm 3. Matrices $\hat{M}_{11}$ and $\hat{M}_{12}$ have the same dense structure as matrices $I - \hat{H}_n$ and $\hat{\Phi}_n$, hence the high-level architecture is very similar, as illustrated in Figure 6.3. The differences lie in the size of the matrices, which affect the number of clock cycles to compute one iteration

$$L_A := \left\lceil \frac{n_A}{P} \right\rceil + l_A \lceil \log_2 (n_A) \rceil + l_M + 6l_A + 2,$$  \hspace{1cm} (6.19)
where $n_A := N(n_u + n_x + |S|) + n_x + |S|$, warm-starting support for variables $z$ and $\nu$ (shown in Figure 6.2), and the projection block for supporting soft state constraints described in Figure 6.5. This block performs the projection of the pair $(x, \delta)$ onto the set satisfying $\{|x - c| \leq r + \delta, \delta \geq 0\}$ by using an explicit solution map for the projection operation and computing the search procedure efficiently. In fact, only $l_A$ extra cycles are needed compared to the standard hard-constrained projection. The block performs a set of comparisons that are used to drive the select signal of a multiplexer.

Note that since multiplication and division by powers of two requires no resources in hardware (just a reinterpretation of an array of signals), if $\rho$ is restricted to be a power of two, no hardware multipliers are required in ADMM outside of the matrix-vector multiplication block. Table 6.2 compares the resources required to implement the two architectures. Again, with ADMM we trade higher resource requirements and longer delays for the capability to solve more general problems.

Note that in a custom hardware implementation of either of our two methods, the number of execution cycles per iteration is exact. We also employ a fixed number of iterations in our implementations of both algorithms, rather than implementing a numerical conver-

<table>
<thead>
<tr>
<th></th>
<th>Fast gradient</th>
<th>ADMM</th>
</tr>
</thead>
<tbody>
<tr>
<td>multipliers</td>
<td>$P[Nn_u + 2]$</td>
<td>$Pn_A$</td>
</tr>
<tr>
<td>adders/subtracters</td>
<td>$P[Nn_u + 3]$</td>
<td>$P[n_A + 15]$</td>
</tr>
<tr>
<td>memory blocks</td>
<td>$P[Nn_u + n_x + 4]$</td>
<td>$P[n_A + 8]$</td>
</tr>
<tr>
<td>size of memory blocks</td>
<td>$\lceil \frac{Nn_u}{P} \rceil$</td>
<td>$\lceil \frac{n_A}{P} \rceil$</td>
</tr>
</tbody>
</table>
gence test, since such convergence tests represent a somewhat self-defeating computational bottleneck in a hard real-time context. Providing cycle accurate completion guarantees is critical for reliability in high-speed real-time applications [121].

6.4 Case studies

This section presents two case studies to evaluate the custom architectures and theoretical bounds described in this chapter. Firstly, we consider the input-constrained optimal control of a real world model of an atomic force microscope (AFM)\(^3\) where the optimization problem is solved via the fast gradient method. This system is an example of a highly dynamic positioning system requiring a sampling rate in excess of 1MHz. Secondly, for easier comparison with the existing literature, we consider a widely studied benchmark example consisting of a set of oscillating masses attached to walls, as described in Section 4.4, for both FGM and ADMM.

6.4.1 Optimal control of an atomic force microscope

We consider the control of an AFM in which the overall objective is to obtain a topographical image of a sample specimen by measuring and manipulating the vertical clearance of a cantilever beam from the surface of the sample. The considered AFM system is depicted schematically in Figure 6.6, in which the specific control objective during the imaging process is to maintain a constant reference distance \( r = 50 \text{ nm} \) of the cantilever tip from the sample surface. The varying height \( d \) of the imaged sample can be controlled via the vertical displacement \( u \) of a piezoelectric plate actuator supporting the sample.

We use an experimentally obtained AFM system model from [116] whose frequency response is shown in Figure 6.7, along with the frequency response of a 12\(^{th}\) order LTI SISO model of the system. We use a state-space representation of this model in observer staircase form, so that the first state is directly proportional to the controlled error signal \( r - (d + y) \), in order to facilitate tuning of the controller via manipulation of the MPC objective function. We assume an input constraint \( u \in [0, 12.5] \), representing the allowable input voltage range of the piezoelectric actuator. For the purposes of evaluating our FGM implementation of MPC, we assume that the system state is available from some external estimator. We choose a diagonal cost matrix \( Q \) and scalar \( R \) such that the system achieves the simulated closed-loop behavior exemplified by Figure 6.8 when the controller is implemented in a standard reference tracking configuration. To achieve good closed-loop performance, we target controller sampling rates in excess of 1 MHz.

Our goal is to choose the minimum number of bits and fast gradient iterations such that the closed-loop performance is satisfactory while minimizing the amount of resources needed to achieve the desired sampling frequencies. Figure 6.9 shows the convergence behaviour of the fast gradient method for one sample in the simulation with an actively

\(^3\)I wish to thank Abu Sebastian of IBM Zürich and Stefan Kuiper for experimental data and technical advice related to the AFM example.
Figure 6.6: Schematic diagram of the atomic force microscope (AFM) experiment. The signal $u$ is the vertical displacement of the piezoelectric actuator, $d$ is the sample height, $r$ is the desired sample clearance, and $y$ is the measured cantilever displacement.

Figure 6.7: Bode diagram for the AFM model (dashed, blue), and the frequency response data from which it was identified (solid, green).
Figure 6.8: Typical cantilever tip deflection (nm, top), control input signal (Volts, middle) and sample height variation (nm, bottom) profiles for the AFM example.

Table 6.3: Relative percentage difference between the tracking error for a double precision floating-point controller using $I_{\text{max}} = 400$ and different fixed-point controllers.

<table>
<thead>
<tr>
<th>$I_{\text{max}} \backslash b$</th>
<th>10</th>
<th>12</th>
<th>14</th>
<th>16</th>
<th>18</th>
<th>20</th>
<th>22</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>55.18</td>
<td>33.25</td>
<td>29.13</td>
<td>28.74</td>
<td>29.28</td>
<td>29.25</td>
<td>30.65</td>
</tr>
<tr>
<td>20</td>
<td>16.13</td>
<td>0.88</td>
<td>0.06</td>
<td>0.02</td>
<td>0.02</td>
<td>0.02</td>
<td>0.02</td>
</tr>
<tr>
<td>25</td>
<td>17.56</td>
<td>0.96</td>
<td>0.05</td>
<td>0.01</td>
<td>0.01</td>
<td>0.01</td>
<td>0.01</td>
</tr>
<tr>
<td>30</td>
<td>17.57</td>
<td>0.96</td>
<td>0.04</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>35</td>
<td>17.42</td>
<td>0.95</td>
<td>0.04</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
</tr>
</tbody>
</table>

constrained solution. The maximum attainable accuracy for different numbers of bits is determined by the residual round-off error $\eta_i$, whose maximum magnitude, as predicted by Proposition 3 and (6.16) and (6.17), converges to a finite value.

Table 6.5 shows the relative difference in closed-loop tracking performance for different fixed-point controllers compared to a double precision floating-point controller executing 400 fast gradient iterations at each sample (considered to achieve optimal tracking). It is clear that 15 iterations are not enough for satisfactory tracking. Assuming that a relative tracking error smaller than $0.1\%$ is desirable, using 20 fast gradient iterations and 14 fraction bits would be the optimal choice.

Table 6.4: Resource usage and potential performance at 400MHz (Virtex6) and 230MHz (Spartan6) with $I_{\text{max}} = 20$.

<table>
<thead>
<tr>
<th>$P$</th>
<th>multipliers</th>
<th>V6 $T_{\text{r}}$ (\textmu s)</th>
<th>S6 $T_{\text{r}}$ (\textmu s)</th>
<th>S6 chip</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>18</td>
<td>1.30</td>
<td>2.26</td>
<td>LX16</td>
</tr>
<tr>
<td>2</td>
<td>36</td>
<td>0.90</td>
<td>1.57</td>
<td>LX25</td>
</tr>
<tr>
<td>3</td>
<td>54</td>
<td>0.80</td>
<td>1.39</td>
<td>LX45</td>
</tr>
<tr>
<td>4</td>
<td>72</td>
<td>0.70</td>
<td>1.21</td>
<td>LX75</td>
</tr>
<tr>
<td>6</td>
<td>108</td>
<td>0.65</td>
<td>1.13</td>
<td>LX75</td>
</tr>
<tr>
<td>7</td>
<td>126</td>
<td>0.60</td>
<td>1.04</td>
<td>LX75</td>
</tr>
<tr>
<td>16</td>
<td>288</td>
<td>0.55</td>
<td>0.96</td>
<td>-</td>
</tr>
</tbody>
</table>
Figure 6.9: Convergence of the fast gradient method under different number representations.

For a fixed number of iterations one can calculate the execution time deterministically according to (8.18). The FPGA designs can be clocked at more than 400 MHz using chips from Xilinx’s high-performance Virtex 6 family or at more than 230 MHz using the low cost and low power Spartan 6 family. Table 6.6 shows the achievable sampling times on the two families for different levels of parallelisation. The resource usage is stated in terms of the number of embedded multiplier blocks since this is the limiting resource in these designs. With Virtex 6 devices one can achieve sampling times beyond 1 MHz for \( P = 2 \) and close to 2 MHz for \( P = 16 \) (maximum parallelism), whereas for Spartan 6 devices well over 600 kHz sampling frequencies are achievable with \( P = 2 \) and close to 1 MHz for \( P = 7 \). For Virtex 6, all designs fit inside the smallest device in the family (LX75T), whereas for Spartan 6 technology a variety of chips will be suitable for different designs. Note that the devices in the low power family will have power ratings in the region of 1 Watt.

### 6.4.2 Spring-mass-damper system

We consider a widely studied benchmark example consisting of a set of oscillating masses attached to walls [114,222], as illustrated by Figure 4.2. In this case, the system is sampled every 0.5 seconds assuming a zero-order hold and the masses and the spring constants have a value of 1kg and 1Nm\(^{-1}\), respectively. The system has four control inputs and two states for each mass, its position and velocity, for a total of eight states. The goal of

\[\|z^*(\hat{x}) - \hat{z}_i\|_2\]

\[\text{Number of fast gradient iterations } i\]

\[\text{double } b = 12\]
\[\text{b = 15}\]
\[\text{b = 18}\]
\[\text{b = 24}\]
\[\text{b = 32}\]

\[\text{Figure 6.9: Convergence of the fast gradient method under different number representations.}\]
the controller, with parameters $N = 10$, $Q = I$ and $R = I$, is to track a reference for the position of each mass while satisfying the system limits.

We first consider the case where the control inputs are constrained to the interval $[-0.5, 0.5]$ and the optimization problem (6.1) with 40 optimization variables is solved via the fast gradient method. Secondly, we consider additional hard constraints on the rate of change in the inputs on the interval $[-0.1, 0.1]$ and soft constraints on the states corresponding to the mass positions on the interval $[-0.5, 0.5]$. The remaining states are left unconstrained. The state is augmented to enforce input-rate constraints, and the further inclusion of slack variables increases the dimension of the state vector to $n_x = 16$.

Note that for problems of this size, MPC control designs based on parametric programming [13, 35] are generally not tenable, necessitating online optimization methods. The resulting problem with 216 optimization variables in the form (6.4) is solved via ADMM. The closed-loop trajectories using an MPC controller based on a double precision solver running to optimality are shown in Figure 6.10, where all the constraints become active for a significant portion of the simulation. We do not include any disturbance model in our simulation, although the presence of an exogenous disturbance signal would not lead to infeasibility since the MPC implementation includes only soft-constrained states. Trajectories arising from closed-loop simulation using a controller based on our fixed-point methods are indistinguishable from those in Figure 6.10, so are excluded for brevity.

As a reference for later comparison, an input-constrained problem with two inputs and 10 states, formulated as an optimization problem of the form (6.1) with 40 variables, was solved in [114] using the fast gradient method in approximately 50 $\mu$seconds. In terms of state-constrained implementations, a problem with three inputs and 12 states, formulated as a sparse quadratic program with hard state constraints and 300 variables, was solved in [222] using an interior-point method reporting computing times in the region of 5 milliseconds, while the state constraints remained inactive. In both cases, the solvers were implemented in software on high-performance desktop machines.

Our goal is to choose the minimum number of bits and solver iterations such that the closed-loop performance is satisfactory while minimising the amount of resources needed to achieve certain sampling frequencies. Figure 6.11 shows the convergence behavior of the fast gradient method and ADMM for two samples in the simulation with an actively constrained solution. The theoretical error bounds on the residual round-off error $\eta_i$, given by (6.15), allow one to make practical predictions for the actual error for a given number of bits, which, as predicted by Proposition 3 and (6.16) and (6.17), converges to a finite value. Table 6.5 shows the relative difference in closed-loop tracking performance for different fixed-point fast gradient and ADMM controllers compared to the optimal controller. Assuming that a relative error smaller than 0.05% is desirable, using 15 solver iterations and 16 fraction bits would be a suitable choice for the fast gradient method. The problem (6.4) solved via ADMM appears more vulnerable to reduced precision implementation, although satisfactory control performance can still be achieved using a surprisingly small number of bits. In this case, employing more than 18 fraction bits or more than 40
Figure 6.10: Closed-loop trajectories showing actuator limits, desirable output limits and a time-varying reference. On the top plot 21 samples hit the input constraints. On the bottom plot 11, 28 and 14 samples hit the input, rate and output constraints, respectively. The plots show how MPC allows for optimal operation on the constraints.
Figure 6.11: Theoretical error bounds given by (6.15) and practical convergence behavior of the fast gradient method (left) and ADMM (right) under different number representations.
Table 6.5: Percentage difference in average closed-loop cost with respect to a standard double precision implementation. In each table, \( b \) is the number of fraction bits employed and \( I_{\text{max}} \) is the (fixed) number of algorithm iterations. In certain cases, the error increases with the number of iterations due to increasing accumulation of round-off errors.

<table>
<thead>
<tr>
<th>( I_{\text{max}} ) ( \backslash ) ( b )</th>
<th>10</th>
<th>12</th>
<th>14</th>
<th>16</th>
<th>18</th>
<th>20</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>5.30</td>
<td>2.76</td>
<td>2.87</td>
<td>3.03</td>
<td>3.05</td>
<td>3.06</td>
</tr>
<tr>
<td>10</td>
<td>14.53</td>
<td>0.14</td>
<td>0.06</td>
<td>0.18</td>
<td>0.20</td>
<td>0.02</td>
</tr>
<tr>
<td>15</td>
<td>17.04</td>
<td>0.35</td>
<td>0.25</td>
<td>0.04</td>
<td>0.00</td>
<td>0.01</td>
</tr>
<tr>
<td>20</td>
<td>16.08</td>
<td>0.15</td>
<td>0.19</td>
<td>0.06</td>
<td>0.01</td>
<td>0.00</td>
</tr>
<tr>
<td>25</td>
<td>17.27</td>
<td>0.15</td>
<td>0.19</td>
<td>0.05</td>
<td>0.01</td>
<td>0.00</td>
</tr>
<tr>
<td>30</td>
<td>16.90</td>
<td>0.31</td>
<td>0.21</td>
<td>0.03</td>
<td>0.02</td>
<td>0.00</td>
</tr>
<tr>
<td>35</td>
<td>18.44</td>
<td>0.19</td>
<td>0.22</td>
<td>0.05</td>
<td>0.01</td>
<td>0.00</td>
</tr>
</tbody>
</table>

FGM

<table>
<thead>
<tr>
<th>( I_{\text{max}} ) ( \backslash ) ( b )</th>
<th>10</th>
<th>12</th>
<th>14</th>
<th>16</th>
<th>18</th>
<th>20</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>53.49</td>
<td>0.18</td>
<td>1.17</td>
<td>0.68</td>
<td>0.57</td>
<td>0.58</td>
</tr>
<tr>
<td>15</td>
<td>47.84</td>
<td>0.46</td>
<td>1.08</td>
<td>0.63</td>
<td>0.51</td>
<td>0.49</td>
</tr>
<tr>
<td>20</td>
<td>44.79</td>
<td>0.76</td>
<td>0.95</td>
<td>0.57</td>
<td>0.45</td>
<td>0.42</td>
</tr>
<tr>
<td>25</td>
<td>47.03</td>
<td>0.98</td>
<td>0.86</td>
<td>0.51</td>
<td>0.39</td>
<td>0.37</td>
</tr>
<tr>
<td>30</td>
<td>45.17</td>
<td>1.02</td>
<td>0.82</td>
<td>0.46</td>
<td>0.35</td>
<td>0.32</td>
</tr>
<tr>
<td>35</td>
<td>46.02</td>
<td>1.07</td>
<td>0.81</td>
<td>0.43</td>
<td>0.31</td>
<td>0.28</td>
</tr>
<tr>
<td>40</td>
<td>46.87</td>
<td>1.29</td>
<td>0.74</td>
<td>0.41</td>
<td>0.28</td>
<td>0.25</td>
</tr>
</tbody>
</table>

ADMM iterations results in insignificant improvements.

For the implementation of ADMM there are a number of tuning parameters left to the control designer. Setting the regularization parameter to \( \rho = 2 \) simplifies the implementation and provided good convergence behavior. The maximum observed value for the Lagrange multipliers \( \nu \) was 7.8, so the penalty parameter \( \sigma_1 \) was set to \( \sigma_1 = 8 \) to obtain an exact penalty formulation as described by Theorem 1. In Section 6.1.3 it was noted that the convergence of ADMM can be very slow when there is large mismatch between the size of the primal and dual variables. This problem can be largely avoided by scaling the matching condition (6.5) with a diagonal matrix, where the entries associated with the soft-constrained states and the slack variables are assigned \( \sigma \) and the rest are assigned 1. This scaling procedure correspond to variable transformations \( y = D\tilde{y} \) and \( z = D\tilde{z} \) that can be applied offline.

In order to evaluate the potential computing performance the architectures described in Section 6.3 were implemented in FPGAs. For a fixed number of iterations one can calculate the execution time of the solver deterministically according to (8.18) or (6.19). Table 6.6 shows the achievable sampling times on the two families for different levels of parallelisation. For the input-constrained problem solved via the fast gradient method, one can achieve sampling rates beyond 1 MHz with Virtex 6 devices using a modest amount
of parallelisation. One can also achieve sampling rates in the region of 700 kHz with Spartan 6 devices consuming in the region of 1 W of power. For the state-constrained problem solved via ADMM, since the number of variables is significantly larger, larger devices are needed and longer computational times have to be tolerated. In this case, achievable sampling times range from 40kHz to 200kHz for different Virtex 6 devices.

Note that the fastest performance numbers reported in the literature are in the millisecond region, several orders of magnitude slower than what is achievable using the techniques presented in this chapter.

Table 6.6: Resource usage and potential performance at 400MHz (Virtex6) and 230MHz (Spartan6) with 15 and 40 solver iterations for FGM and ADMM, respectively. The suggested chips in the bottom two rows of each table are the smallest with enough embedded multipliers to support the resource requirements of each implementation.

<table>
<thead>
<tr>
<th>FGM</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>8</td>
<td>16</td>
<td>32</td>
</tr>
<tr>
<td>multipliers</td>
<td>42</td>
<td>84</td>
<td>126</td>
<td>168</td>
<td>336</td>
<td>672</td>
<td>1344</td>
</tr>
<tr>
<td>V6 $T_s$ (µs)</td>
<td>1.95</td>
<td>1.20</td>
<td>0.98</td>
<td>0.82</td>
<td>0.64</td>
<td>0.56</td>
<td>0.53</td>
</tr>
<tr>
<td>S6 $T_s$ (µs)</td>
<td>3.39</td>
<td>2.09</td>
<td>1.70</td>
<td>1.43</td>
<td>1.10</td>
<td>0.98</td>
<td>0.91</td>
</tr>
<tr>
<td>V6 chip</td>
<td>LX75</td>
<td>LX75</td>
<td>LX75</td>
<td>LX75</td>
<td>LX130</td>
<td>LX240</td>
<td>SX315</td>
</tr>
<tr>
<td>S6 chip</td>
<td>LX45</td>
<td>LX75</td>
<td>LX75</td>
<td>LX100</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ADMM</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>multipliers</td>
<td>216</td>
<td>432</td>
<td>648</td>
<td>864</td>
<td>1080</td>
<td>1296</td>
<td>1512</td>
</tr>
<tr>
<td>V6 $T_s$ (µs)</td>
<td>23.40</td>
<td>12.60</td>
<td>9.00</td>
<td>7.20</td>
<td>6.20</td>
<td>5.40</td>
<td>4.90</td>
</tr>
<tr>
<td>S6 $T_s$ (µs)</td>
<td>40.70</td>
<td>21.91</td>
<td>15.65</td>
<td>12.52</td>
<td>10.78</td>
<td>9.39</td>
<td>8.52</td>
</tr>
<tr>
<td>V6 chip</td>
<td>LX75</td>
<td>LX130</td>
<td>LX240</td>
<td>LX550</td>
<td>SX315</td>
<td>SX315</td>
<td>SX475</td>
</tr>
<tr>
<td>S6 chip</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

### 6.5 Summary and open questions

This chapter has proposed several custom computational architectures for different first-order optimization methods that can handle linear-quadratic MPC problems with input, input-rate, and soft state constraints. First-order methods are very well-suited for custom hardware acceleration because the algorithms are based on matrix-vector multiplication with few extra sequential dependencies and they can be fully implemented using fixed-point arithmetic. Implementation of the proposed architectures in FPGAs has shown that satisfactory control performance at a sample rate beyond 1 MHz is achievable even on low-end devices, opening up new possibilities for the application of MPC on resource-constrained embedded systems.

This chapter has also provided a unified error analysis framework for different first-order methods that can be used to obtain practical estimates for the expected error in the min-
imiser given the computing precision. Such a tool can be used for offline controller design. The case studies have demonstrated that the algorithms remain numerically reliable at very low bit-widths.

A target for future work in this area could be to extend the error analysis such that one can choose the computing precision a priori given a closed-loop tracking error specification. This would require to link the error in the minimiser to the error in the function value and then characterise the expected effect of suboptimal solutions on the closed-loop trajectories.

In order to extend the applicability of the proposed architectures to a wider set of MPC problems, a larger set of efficient projection architectures for common simple sets need to be designed. Automatic ways of deriving efficient parallel projection architectures given a mathematical description of a set would also be desirable.

Further work could also investigate the use of the error analysis framework to derive new algorithms or problem transformations such that the resulting reduced precision implementations are less vulnerable to error accumulation. Being able to implement the MPC controller using less bits while retaining reliability would allow one to use less resources or to increase the sampling frequencies for a given amount of resources.
7 Predictive Control Algorithms for Parallel Pipelined Hardware

Complex custom circuits, especially those implemented in FPGAs, often have to be deeply pipelined to achieve high clock frequencies. When the implemented algorithms are iterative, as opposed to stream processing applications, long pipelines can result in low hardware utilization, since a new iteration cannot start until the data from the previous iteration is ready, often resulting in parts of the circuit remaining idle for a significant fraction of time. While this shortcoming also affects the fixed-point architectures for first-order solvers presented in Chapter 6, the impact is much more pronounced on the floating-point interior-point architectures of Chapter 5 due to the very long latencies associated with floating-point arithmetic units in FPGAs and the characteristics of the interior-point algorithm.

However, long pipelines can also be used to our advantage. It is possible to use the idle computational power to time-multiplex multiple problems into the same circuit to hide the pipeline latency and keep arithmetic units busy at all times [124]. The result of applying the proposed technique is that the interior-point architectures can solve several independent QP problems simultaneously, while using the same hardware resources as for solving only one QP problem. Unlike with software platforms, custom hardware gives the necessary cycle accurate execution control to ensure that solving several problems does not have any performance impact on the time taken to solve just one problem.

In recent years, following the advent of widespread parallel processing capabilities, there have been some algorithmic developments in the MPC community to attempt to make use of these new capabilities in a different manner than by parallelising a given standard algorithm [129, 134]. The proposed techniques often make use of suboptimal solutions (more than one) with respect to the original control problem (4.2) but result in improved closed-loop behaviour as a consequence of faster sampling, leading to improved disturbance rejection capabilities.

Even though most of the developments originally targeted multi-core general-purpose processors, the concepts are still valid for exploiting the unusual characteristics of our interior-point architectures. This chapter will show how these new MPC algorithms can be adapted for implementation in deeply pipelined architectures for improving the computational efficiency of the optimization solver. We will also propose new unconventional ways to make use of this available slack computational power to improve the performance of the control system.
This chapter starts by explaining the concept of pipelining, how long pipelines arise, and their consequences in Section 7.1. Section 7.2 gives an overview of several methods for filling the pipeline in interior-point architectures with a special focus on a scheme labelled parallel multiplexed MPC. Section 7.3 summarises open questions in this area.

### 7.1 The concept of pipelining

In Chapter 3, pipelining was briefly discussed in the context of general-purpose processors. In this section we will go into more details about the benefits and consequences of pipelining in custom datapaths.

In synchronous digital circuitry, the content or output of a register is updated with its input value at every rising (or falling) edge of the clock signal. Between clock edges the contents of the registers are held constant. As a consequence, for the digital circuit to behave in a reliable way, the clock period has to be longer than the longest propagation delay between two registers. This is required to make sure that all register inputs have the final value of the logical operations between all registers at the clock rising edge trigger.

Consider Figure 7.1 a) which shows a sequence of logic gates with a propagation delay of $t_d$ seconds between two registers. If this is the longest combinatorial path in the circuit, then $f_c < \frac{1}{t_d}$, and inputs can be accepted every $\frac{1}{f_c}$ seconds. One method to increase the clock frequency and the rate at which the circuit can process inputs is to insert more registers in the combinatorial path. When the registers are placed such that the propagation delay between all registers is the same, as shown in Figure 7.1 b), the overall input-output delay or latency does not increase under the assumption of negligible register delays. In this case the clock frequency can be increased by a factor of six and the throughput, or the rate at which inputs can be processed, also increases by a factor of six. However, in general it is not possible to place registers such that the delay between all registers is the same. The situation shown in Figure 7.1 c) is more common. In this case $t_3$ is the longest delay path, the throughput is given by $\frac{1}{t_3}$ and the input-output latency is given by $6t_3 > t_d$, hence one trades an increase in latency for an increase in throughput.

#### 7.1.1 Low- and high-level pipelining

Pipelining can occur at the register transfer level or at the algorithmic level. At the lowest level, pipelining consists of inserting registers between logic gates. At a slightly higher level of abstraction one can think, for example, about inserting registers inside a binary multiplier. For instance, a 4-bit by 4-bit multiplication operation consists of conditionally adding four shifted versions of one of the 4-bit inputs. An array implementation of this multiplier would consist of a sequence of four 4-bit adders. A pipelined implementation of the array multiplier would include registers between the binary adders, which would increase the multiplication latency but would allow the processing of inputs at a faster...
One can also think about pipelining in higher-level operations. For example, the MIN-RES solver implementation in [21] is deeply pipelined, trading an increase in the latency of one MINRES iteration for the capability of increasing the rate at which independent linear systems can be processed. At an even higher level of abstraction, consider the interior-point hardware architecture described in Chapter 5, which consists of two separate hardware blocks, one preparing linear equations and the other solving them. In order to increase the rate at which independent QP problems can be processed, the design inserts a high-level register bank between the two blocks. For maximising the efficiency of the circuit the delay of the two blocks has to be ideally the same, as in Figure 7.1 b), which guided design decisions.

7.1.2 Consequences of long pipelines

In some applications, throughput is the most important measure. For example, in digital audio processing or in packet switching, a small increase in input-output latency can be tolerated if it allows one to significantly increase the audio sampling rate or the rate at which packets can be processed by a router. Custom hardware for these applications can often have very long pipelines without incurring performance penalties.

For iterative latency-sensitive applications, long delays are problematic since a new iteration cannot start before the data from the previous iteration is available. In MPC, latency is the most important measure as it determines how fast one can sample from the system’s sensors and respond to disturbances and setpoint changes. Floating-point units have very long delays that result in large overall iteration delays in the linear solver.
inside the interior-point solvers. The consequence of a long pipeline, as is the case with
general-purpose processors, is that many independent operations are needed to make an
efficient utilisation of the hardware resources. Our task in this chapter is to derive MPC
strategies that can make better use of the long pipeline in the floating-point interior-point
solvers of Chapter 5.

7.2 Methods for filling the pipeline

This section describes methods for increasing the utilization of the execution pipeline.
It starts by proposing an unconventional sampling technique that is independent of the
control strategy or algorithm used. The chapter then discusses estimation problems and
distributed optimization algorithms, also independent of the use of MPC. The last three
subsections present MPC-specific algorithms that require solving many similar indepen-
dent convex QPs.

The implementation of the method described in Section 7.2.6 in a pipelined architecture
is described and evaluated in detail. For the remaining methods, we discuss the way in
which they could be implemented in the architectures of Chapter 5 but experimental
results are not yet available.

7.2.1 Oversampling control

In control systems such as the one described in Figure 2.2, a sensor measurement is taken
and the corresponding control action is usually applied once the controller computation
has terminated, say after $T_c$ seconds. With standard sampling schemes one has to wait
for the computation to terminate before one can sample again, hence the sampling time
$T_s$ is bounded from below by $T_c$.

One can also ignore that constraint and sample again to initiate the computation of
another control action before the previous computation has terminated. Under this intra-
delay sampling scheme [26, 27], $T_s < T_c$. Figure 7.2 illustrates the difference between the
standard sampling and oversampling schemes. Note that the implementation of intra-delay
sampling relies on the availability of a computational platform that supports concurrent
computations. With respect to Figure 7.2 two different control computations will always
be occurring simultaneously. The latency of the control action will still be $T_c$ but the rate
at which control actions are being applied (throughput) will double.

This sampling scheme is common for streaming computations, such as those occurring
in signal processing, but can also provide several advantages for control. By increasing the
sampling frequency, the controller can handle higher bandwidth disturbances. Oversam-
pling control also reduces the maximum reaction time to disturbances, given by $T_c + T_s$,
which could lead to better disturbance rejection. In addition, since control actions will be
applied at a faster rate, the control trajectory is expected to be smoother, which could
help to overcome slew rate limitations in the actuators.
Attempting to reduce the sampling time only by parallelising the implementation has certain limitations. With standard sampling $T_c$, and hence $T_s$, become a function of the exploited parallelism, but Amdahl’s law (refer to Section 3.1.3) states that the acceleration through parallelisation is severely limited by the nature of the algorithm. With oversampling control, $T_s$ depends on the amount of concurrent computations that can be executed, which is independent of the parallelisation opportunities in the control algorithm, giving greater flexibility.

This scheme could be implemented using an architecture such as the one described in Chapter 5. In this case, the concurrent control computations would not necessitate extra hardware resources but they will use the slack computing power in the pipelined architecture. Parallelism in the interior-point architecture reduces the computing latency $T_c$ and pipelining allows one to reduce the sampling time $T_s$ even further. Of course, the oversampling factor is limited by the number of independent QP problems that can be handled simultaneously by the architecture, given by (5.3).

A GPU could also be a suitable candidate for implementing this oversampling scheme, since the independent concurrent operations could be used to fill its long pipelines. However, in this case the independent QP computations will not be synchronised. In fact, at a given time all QP solvers will be at a different iteration, and the input and output data arrival and departure times will be different. This lack of synchronisation between kernels...
could be problematic for a GPU implementation. In custom hardware we have complete control of I/O and computation scheduling, hence this scheme could be implemented efficiently.

7.2.2 Moving horizon estimation

When the state vector is not fully measurable, the process of estimation involves reconstructing the state vector (and disturbance) from current and past sensor measurements at the system’s output. The estimate is then passed on to the controller to compute a correcting action (refer to Figure 2.2).

Moving horizon estimation (MHE) is an optimization-based technique analogous to model predictive control for the estimation problem [89, 183, 186]. Instead of predicting the state and input trajectories into the future, the MHE strategy starts with an estimated value of the state in the past and reconstructs the state trajectory in the time window between the estimated value and the current time instant, taking into account the physical constraints of the system and bounds on the disturbances. Given

i) a noisy linear map between the system’s internal state and the system’s output, i.e. $y_k \approx Cx_k$,

ii) a guess $x$ of the state at the start of the estimation window,

iii) the previous $N$ control commands, $u_k$ for $k = 1, \ldots, N$, and

iv) the current and previous $N$ output measurements, $y_k$ for $k = 0, 1, \ldots, N$,

an MHE estimator solves a constrained $N$-stage optimal estimation problem in the form

$$ J^*(w_1, \ldots, w_N, x_0, \ldots, x_N) := $$

$$ \min \frac{1}{2} (x_0 - x)^T \tilde{Q}(x_0 - x) + \frac{1}{2} \sum_{k=1}^{N} (y_k - Cx_k)^T Q(y_k - Cx_k) + w_k^T R w_k $$

(7.1)

subject to

$$ x_{k+1} = A_d x_k + B_d u_k + w_k, \quad k = 0, 1, \ldots, N - 1, \quad (7.2a) $$

$$ x_k \in \mathbb{X}, \quad k = 0, 1, \ldots, N, \quad (7.2b) $$

$$ y_k - C x_k \in \mathbb{V}, \quad k = 0, 1, \ldots, N, \quad (7.2c) $$

$$ w_k \in \mathbb{W}, \quad k = 1, \ldots, N. \quad (7.2d) $$

where $\mathbb{X}$, $\mathbb{V}$ and $\mathbb{W}$ are convex sets and the matrices $Q$, $R$ and $\tilde{Q}$ are chosen such that the problem is convex with a unique solution and the estimator remains stable [122]. If a feasible solution exists, the state estimate to be sent to the controller is the optimal value $x_N^*$. 

143
In a standard control system, the state is estimated in $T_e$ seconds, the input command is computed in $T_c$ seconds and then the system is sampled again. One can increase the rate at which state estimates are available by sampling the system’s output faster and starting new MHE problem instances before the $T_e + T_c$ cycle has been completed. Since all estimation problems are independent, they can be solved concurrently.

The increased throughput in the estimator could be used to continuously update the state estimate for the controller, while it is computing the control action, so that it can make use of more current state information. Since, when using a non-condensed formulation, the state estimate only changes some coefficients of the equality constraints (refer to Section 4.2.1), which only affect the right-hand vector of the linear systems solved in an interior-point method, updating the state estimate in an interior-point solver is straightforward and involves no computation.

The problem (7.1)–(7.2) is a quadratic program in the same form as the optimal control problem (4.2)–(4.3), hence can be solved using the same methods. An interior-point architecture designed with the same principles as the one described in Chapter 5 would have the characteristic of being able to solve several MHE problems simultaneously without requiring extra hardware resources.

### 7.2.3 Distributed optimization via first-order methods

When one solves an equality constrained QP using a first-order method one has to solve the dual problem (refer to Section 2.2.3). In this case, the computation of the gradient of the dual function (2.31) is itself an optimization problem of the form (2.32). If the cost function is separable, this task can be decomposed into several independent optimization problems that can be solved concurrently using an architecture such as the one described in Chapter 5. Unfortunately, for ADMM, the steps for computing the approximation of the dual gradient are not independent (see (2.33)–(2.34)), hence they cannot be carried out concurrently.

### 7.2.4 Minimum time model predictive control

A minimum time MPC formulation includes the horizon length as a decision variable for applications where it is desirable to reach a given state in the minimum number of steps. Since the horizon length is a discrete integer variable the resulting optimization problem is non-convex.

Example applications that can benefit from this formulation due its finite-time completion guarantees include aerial vehicle manoeuvres [189] and spacecraft rendezvous [88]. In such applications one might want to minimize, for instance,

$$\sum_{i=0}^{N-1} (1 + \|u_i\|_1)$$

subject to the system’s dynamics, $N \leq N_{\text{max}}$, and $x_N$ being equal to the target final
Having a variable control horizon enables a balance between fuel usage and manoeuvre completion time. Simulations have shown that fuel consumption using this MPC formulation is favourable compared to conventional control methods.

Even though problem (7.3) is non-convex, since \( N \) is restricted to a small discrete set, it is feasible to pose the variable horizon problem as a sequence of quadratic (or linear) convex programs. These problems are all independent and can be solved in parallel or in a pipelined architecture [86] such as the one described in Chapter 5. Since the problems are of increasing size with the horizon length, the architecture would need to be designed for \( N = N_{\text{max}} \).

### 7.2.5 Parallel move blocking model predictive control

The main shortcoming of MPC is its very high computational demand. One approach for reducing the computational load is to approximate the original optimization problem (4.2) and work with suboptimal solutions. The schemes that are presented in the following two subsections build on this concept.

With move blocking model predictive control the approximation consists of forcing the control input trajectory to remain constant over a larger period than one sampling interval, thereby reducing the degrees of freedom in the optimization problem [139]. This effectively reduces the number of steps in the prediction horizon. Figure 7.3 illustrates the strategy with three hold intervals \( m_0, m_1 \) and \( m_2 \) consisting of two, three and four sampling intervals, respectively. The effective prediction horizon is \( \hat{N} = 3 \) and

\[
\sum_{i=0}^{\hat{N}-1} m_i = N
\]

(7.4)

always holds. The solution to the approximated problem can be computed faster, hence this scheme gives the freedom to trade suboptimality of the control action with the achievable sampling period. Exploring this trade-off makes sense because faster sampling gives better disturbance rejection.

Since the control inputs are only allowed to change at specific points, the number of state variables is also reduced. In fact, constraints can only be enforced at these discrete intervals, hence the state constraint set \( X_\Delta \) has to be adjusted according to the hold interval lengths \( m_i \) to guarantee constraint satisfaction within that interval. For details on this procedure, see [241]. When the hold intervals lengths are not equal the time-invariant MPC problem (4.2) becomes time-varying. The dynamics constraint (4.3b) would also have different matrices \( A_{d,k} \) and \( B_{d,k} \) for \( k = 0, 1, \ldots, \hat{N} - 1 \).

Whereas the MPC scheme that we describe in the next subsection can only be applied to systems with more than one input, move blocking MPC only requires \( N > 1 \), which is true for most MPC problems. Since the computational savings result from a reduction in the horizon length from \( N \) to \( \hat{N} \), the magnitude of the savings depends to a large extent on the MPC formulation used (see Chapter 4 for details on the different formulations).
Recall that the computational effort for solving condensed QPs grows cubically in the horizon length but only linearly with the non-condensed formulation.

While move blocking can offer computational savings, it is well-known that if the hold intervals are not uniform, the strategy will lack feasibility and stability guarantees [30]. With standard MPC schemes, recursive feasibility is guaranteed by shifting the solution at the previous time instant and setting $u_{N-1} = Kx_N$. When $x_N$ is forced to lie inside a terminal invariant set with respect to the gain $K$, this step guarantees that the new $x_N$ will also be feasible. Since the objective function is also a Lyapunov function the scheme is also guaranteed to be stable (see [147] for more details on the stability of MPC). For non-uniform move blocking, the shifting argument does not generally apply because the points where the inputs are held constant in the shifted vector do not correspond to the points where the inputs are held constant in the solution at the previous time step.

Parallel move blocking [134,135] is a scheme that can reduce the suboptimality of the implemented control action and can also retain the feasibility and stability guarantees of standard MPC. The strategy solves several optimization problems in parallel with different blocking or hold constraints. The input sequence that results in the lowest open-loop cost is selected and the problems are solved again at the next sampling instant. Note that the parallel version reduces the suboptimality over sequential move blocking because the cost function can only be equal or lower than the case where only one move blocking problem is solved. In addition, the shifting argument for establishing recursive feasibility of parallel move blocking holds by appropriate selection of the set of hold constraints [134,135].

If the number of effective prediction horizon steps $N$ is the same for all move blocking problems to be solved simultaneously, then the scheme generates a set of independent optimization problems of the same size and structure, which can be solved efficiently with an architecture such as the one described in Chapter 5.
7.2.6 Parallel multiplexed model predictive control

Multiplexed MPC (MMPC) was proposed to reduce the computational burden of MPC [130, 190]. This section extends the MMPC algorithm in a way such that the interior-point architecture proposed in Chapter 5, which is capable of solving several QP problems simultaneously, can be exploited. This version of MMPC will be referred to as parallel MMPC.

The original formulation of MMPC was derived for implementation on a single core sequential processor, solving one QP problem per sampling interval. The key idea is that, for an $n_u$-input plant, instead of optimizing over all the $n_u$ input channels in one large QP, the input trajectories are optimized one channel at a time, in a pre-planned periodic sequence, and the control moves updated as soon as the solution becomes available. This results in a smaller QP at each sampling instant leading to reduced online computational load, which in turn enables faster sampling and a faster response to disturbances, despite finding a sub-optimal solution to the original optimization problem [128]. Figure 7.4 illustrates the difference between standard MPC and a multiplexed MPC scheme for a two-input plant. With MMPC the plan for each input is only allowed to change every two sampling intervals in a time-multiplexed fashion.

This scheme is closer to industrial practice in cases where there is a complex plant with network constraints, meaning that all control inputs cannot be updated simultaneously due to limitations in the communication channels between the actuators and the controller. The parallel MMPC scheme that we describe in this section helps to choose which inputs are best to update at any given sampling interval. Algorithm 6 outlines the key steps in parallel MMPC and Figure 7.5 gives an illustrative example for a two-input system.

As can be seen from Algorithm 6, parallel MMPC uses MMPC as an elementary building block. In parallel MMPC, for a plant with $n_u$ inputs, there can be up to $n_u$ copies of MMPC at a given time, each operating independently and in parallel, optimizing with respect to a different subset of control moves. The set of control moves which produces the smallest
Algorithm 6 Parallel MMPC

1. Initialize by optimizing over all the control moves.
2. Store the planned moves (N moves for each input).

while 1 do

3. Apply the first control move for all inputs and shift the plan.
4. Obtain new measurement $x$.
5. Solve $n_u$ different copies of MMPC in parallel. For each copy, optimize with respect to different subsets of control moves.
6. Evaluate and select from these $n_u$ copies of MMPC, the set of control moves that gives the smallest cost.
7. Update the plan for the set of input channels that gives the smallest cost. Retain the previous plan for the other input channels.

end while

Figure 7.5: Parallel multiplexed MPC scheme for a two-input system. Two different multiplexed MPC schemes are solved simultaneously. The angular lines represent when the input command is allowed to change.

cost is selected and applied to the plant. The process is repeated at the next updating instant. The resulting updating sequence does not follow a pre-planned sequence and is not necessarily periodic.

Note that Step 1 in Algorithm 6 involves solving for inputs across all input channels. This type of initialization requirement is common in distributed MPC. Subsequent optimizations use this initial solution, but optimize with respect to a subset of control moves. The stability property of MMPC does not depend on the optimality of this initial solution, only on its feasibility [129]. For parallel MMPC, we state its stability properties in the following proposition:

**Proposition 4.** Parallel MMPC, obtained by implementing Algorithm 6, gives closed-loop stability.

**Proof.** The proof follows standard argument used by most MPC stability proofs, which depends on the constrained optimization being feasible at each step. In the proposed parallel MMPC algorithm, the default MMPC is always evaluated at every iteration, among the $n_u$ parallel copies of MMPC. It then follows that closed-loop stability can be achieved by applying the default MMPC, which is stabilizing. This gives the worst
case since the parallel MMPC algorithm ensures that switching to a different MMPC will further reduce the cost.

Performance evaluation

We evaluate the potential achievable acceleration from employing parallel MMPC by using the slack computational power in the interior-point hardware architecture described in Chapter 5. First, we study the dependence on the problem dimension and then present a case study for the spring-mass-damper system introduced in Section 4.4.

Figure 7.6 compares the computational times for standard MPC and parallel multiplexed MPC when taking advantage of the parallel computational channels provided by the architecture proposed in Chapter 5. Systems with a larger number of inputs will benefit most from employing the multiplexed MPC formulation, as the reduction in size of the QP problem will be larger. The latency expression (5.2) consists of quadratic, linear and constant terms with respect to the number of inputs \( n_u \). If \( n_u \) is small compared to the number of states \( n_x \) and the horizon length \( N \), the constant term dominates and the improvement from using multiplexed MPC diminishes as a consequence. When \( n_u \) is large relative to \( n_x \) and \( N \), the quadratic and linear terms gain more weight, hence the improvement becomes very significant.

To illustrate the performance improvement achievable with parallel MMPC, we apply the hardware controller to the spring-mass system described in Figure 4.2. The example system consists of 18 equal masses (0.15kg) connected by equal springs (1Nm\(^{-1}\)) and no damping. The system has 36 states. Each mass can be actuated by a horizontal force \( (n_u = 18) \) and the reference for the outputs to track is the zero position for all masses. The continuous time regulator matrices are chosen as \( Q_c = I \), \( R_c = I \) and \( S_c = 0 \).

When the horizon length \( T_h \) is specified in seconds, sampling faster leads to more steps in the horizon and larger optimization problems to solve at each sampling instant. For the example system we found that a horizon of \( T_h = 3.1 \) seconds was sufficient. Table 7.1 shows the sampling interval and computational delays for the FPGA implementations for different number of steps in the horizon. For each implementation, the operating sampling interval is chosen to be the smallest possible such that the computational delay allows solving the optimization problem before the next sample needs to be taken. For this example system, employing parallel MMPC allows sampling 22\% faster than with conventional MPC.

Even though the sampling frequency upgrade is modest, there is a reduction in control cost, as shown by the simulation results presented in Figure 7.7. In addition, employing parallel MMPC not only leads to lower sampling intervals but also lower resource usage, since the optimization problems are smaller (as shown in Table 7.2). The extra resources could be used to increase the level of parallelism and achieve greater speed-ups.
Figure 7.6: Computational time reduction when employing multiplexed MPC on different plants. Results are normalised with respect to the case when $n_u = 1$. The number of parallel channels is given by (5.3), which is: a) 6 for all values of $n_u$; b) 14 for $n_u = 1$, 12 for $n_u \in (2, 5]$, 10 for $n_u \in (6, 13]$ and 8 for $n_u \in (14, 25]$. For parallel multiplexed MPC the time required to implement the switching decision process was ignored, however, this would be negligible compared to the time taken to solve the QP problem.
Table 7.1: Computational delay for each implementation when $I_{IP} = 14$ and $I_{MINRES} = Z$. The gray region represents cases where the computational delay is larger than the sampling interval, hence the implementation is not possible. The smallest sampling interval that the FPGA can handle is 0.281 seconds (3.56Hz) when computing parallel MMPC and 0.344 seconds (2.91Hz) when computing conventional MPC. The relationship $T_s = \frac{T_h}{N}$ holds.

<table>
<thead>
<tr>
<th>$N$</th>
<th>FPGA$_1$</th>
<th>FPGA$_{MMPC}$</th>
<th>Sampling interval, $T_s$</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>0.166</td>
<td>0.120</td>
<td>0.442</td>
</tr>
<tr>
<td>8</td>
<td>0.211</td>
<td>0.152</td>
<td>0.388</td>
</tr>
<tr>
<td>9</td>
<td>0.262</td>
<td>0.188</td>
<td>0.344</td>
</tr>
<tr>
<td>10</td>
<td>0.318</td>
<td>0.227</td>
<td>0.310</td>
</tr>
<tr>
<td>11</td>
<td>0.379</td>
<td>0.270</td>
<td>0.281</td>
</tr>
<tr>
<td>12</td>
<td>0.446</td>
<td>0.318</td>
<td>0.258</td>
</tr>
</tbody>
</table>

Figure 7.7: Comparison of the closed-loop performance of the controller using conventional MPC (solid) and parallel MMPC (dotted). The horizontal lines represent the physical constraints of the system. The closed-loop continuous-time cost represents $\int_0^s x(s)^T Q_c x(s) + u(s)^T R_c u(s) \, ds$. The horizontal axis represents time in seconds.

Table 7.2: Size of QP problems solved by each implementation. Parallel MMPC solves six of these problems simultaneously.

<table>
<thead>
<tr>
<th></th>
<th>Decision Variables</th>
<th>Constraints</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPC</td>
<td>522</td>
<td>684</td>
</tr>
<tr>
<td>Parallel MMPC</td>
<td>465</td>
<td>498</td>
</tr>
</tbody>
</table>
7.3 Summary and open questions

Complex floating-point datapaths can lead to very long execution pipelines on FPGAs. For iterative applications, one way to improve the hardware utilisation is to time-multiplex several independent problems onto the same datapath to hide the pipeline latency. When this approach is applied to the interior-point architectures described in Chapter 5, the resulting circuit can solve several independent QP problems using the same resources and dissipating the same power as when solving only one QP problem.

In this chapter we have described several strategies to make use of this special feature to further improve the computational efficiency of optimal decision makers for control applications. For some methods, the need to solve many problems arises from an increase in the sampling frequency beyond the limits assumed in conventional control systems. For other reduced complexity schemes, solving several problems helps to reduce the suboptimality of the computed control action and provide guarantees that cannot be provided by solving just one problem.

We have shown how all of the presented schemes can be implemented on our hardware architectures. A detailed study has shown how employing one of these new strategies, which breaks the original problem into smaller subproblems, allows one to save resources and achieve greater acceleration, leading to better quality control. An implementation of the remaining proposed strategies is still needed to verify their feasibility and effectiveness.

More work is needed to explore the limits and trade-offs in the proposed approaches to aid offline design decisions. For instance, it is still not yet clear how much one can oversample before no extra benefit is attained, or the control scheme becomes unstable. Quantifying the loss in optimality introduced by blocking constraints or by updating only a subset of the input channels also remains an open question. A better understanding of these trade-offs in conjunction with a characterisation of the disturbance rejection capabilities as a function of the sampling period and the disturbance profile would help to optimally tune the free parameters in these novel control schemes for improved closed-loop performance.

Other novel methods that can take advantage of the special features of parallel pipelined hardware are likely to have an impact on controller implementations on future computing platforms.
8 Algorithm Modifications for Efficient Linear Algebra Implementations

Chapters 5 and 6 describe hardware architectures for different optimization algorithms for improving the computational efficiency of embedded solvers and hence extend the range of applications that can benefit from MPC. While Chapter 6 showed how fixed-point arithmetic implementations of first-order solvers can have a dramatic effect on the efficiency of the resulting solution, unfortunately, fixed-point implementation is not straightforward for interior-point methods due to the fundamental characteristics of the original algorithm.

In this chapter we focus on improving the efficiency of the main computational bottleneck in interior-point methods – the solution of systems of linear equations arising when solving for the search direction. As in Chapter 5, we consider iterative methods for solving linear systems. The Lanczos iteration [117] is the key building block in modern iterative numerical methods for computing eigenvalues or solving systems of linear equations involving symmetric matrices. These methods are typically used in scientific computing applications, for example when solving large sparse linear systems of equations arising from the discretization of partial differential equations (PDEs) [46]. In this context, iterative methods are preferred over direct methods, because they can be easily parallelised and they can better exploit the sparsity in the problem to reduce computation and, perhaps more importantly, memory requirements [81]. However, these methods are also interesting for small- and medium-scale problems arising in real-time embedded applications, like real-time optimal decision making. In this domain, on top of the advantages previously mentioned, iterative methods allow one to trade off computation time for accuracy in the solution and enable the possibility of terminating the method early to meet real-time deadlines.

In both cases more efficient forms of computation, in the form of new computational architectures and algorithms that allow for more efficient architectures, could enable new applications in many areas of science and engineering. In high-performance computing (HPC), power consumption is quickly becoming the key limiting factor for building the next generation of computing machines [91]. In embedded computing, cost, power consumption, computation time, and size constraints often limit the complexity of the algorithms that can be implemented, limiting the capabilities of the embedded solution.

Porting floating-point algorithm implementations to fixed-point arithmetic is an effective way to address these limitations. Because fixed-point numbers do not require mantissa alignment, the circuitry is significantly simpler and faster. The smaller delay in arithmetic
operations leads to lower latency computation and shorter pipelines. The smaller resource requirements lead to either more performance through parallelism for a given silicon budget, or a reduction in silicon area leading to lower power consumption and cost. This latter observation is especially important, since the cost of chip manufacturing increases at least quadratically with silicon area [92]. It is for this reason that fixed-point architectures are ubiquitous in high-volume low-cost embedded platforms, hence any new solution based on increasingly complex sophisticated algorithms must be able to run on fixed-point architectures to achieve high-volume adoption. In the HPC domain, heterogeneous architectures integrating fixed-point processing could help to lessen the effects of the power wall, which is the major hurdle in the road to exascale computing [104].

However, while fixed-point arithmetic is widespread for simple digital signal processing operations, it is typically assumed that floating-point arithmetic is necessary for solving general linear systems or general eigenvalue problems, due to the potentially large dynamic range in the data and consequently on the algorithm variables. Furthermore, the Lanczos iteration is known to be sensitive to numerical errors [172], so moving to fixed-point arithmetic could potentially worsen the problem and lead to unreliable numerical behaviour.

To be able to take advantage of the simplicity of fixed-point circuitry and achieve cost, power and computation time reductions, the complexity burden shifts to the algorithm design process [101]. In order to have a reliable fixed-point implementation one has to be able to establish bounds on all variables of the algorithm to avoid online shifting, which would negate any speed advantages, and avoid overflow errors. In addition, the bounds should be of the same order to minimise loss of precision when using constant word-lengths.

There are several tools in the design automation community for handling this task [39]. However, because the Lanczos iteration is a nonlinear iterative algorithm, all state-of-the-art bounding tools fail to provide practical bounds. Unfortunately, most linear algebra kernels (except extremely simple operations) are of this type and they suffer from the same problem.

This chapter proposes a novel scaling procedure to tackle the fixed-point bounding problem for the nonlinear and recursive Lanczos kernel. The procedure gives tight bounds for all variables of the Lanczos process regardless of the properties of the original KKT matrix, while minimizing the computational overhead. The proof, based on linear algebra, makes use of the fact that the scaled matrix has all eigenvalues inside the unit circle. This kind of analysis is currently well beyond the capabilities of state-of-the-art automatic methods [149]. We then discuss the validity of the bounds under finite precision arithmetic and give simple guidelines to be used with existing error analysis [172] to ensure that the absence of overflow is maintained under inexact computation.

The main result is then extended to the MINRES method – a Lanczos-based algorithm for solving linear equations involving symmetric indefinite matrices, and it is expected that the same scaling approach can be used for bounding variables in other nonlinear recursive linear algebra kernels based on matrix-vector multiplication. In this chapter we
also discuss the applicability to the Arnoldi method \cite{6}, a generalization of the Lanczos kernel for non-symmetric matrices.

The potential efficiency improvements of the proposed approach are evaluated on an FPGA platform. While Moore’s law has continued to promote FPGAs to a level where it has become possible to provide substantial acceleration over microprocessors by directly implementing floating-point linear algebra kernels \cite{53,214,244,245}, floating-point operations remain expensive to implement, mainly because there is no hard support in the FPGA fabric to facilitate the normalisation and denormalisation operations required before and after every floating-point addition or subtraction. This observation has led to the development of tools aimed towards fusing entire floating-point datapaths, reducing this overhead \cite{43,118}. However, as described in Section 6.2.1, there is still a very large performance gap between fixed-point and floating-point implementations in FPGAs.

To exploit the architecture flexibility in an FPGA we present a parameterisable architecture generator where the user can tune the level of parallelisation and the data type of each signal. This generator is embedded in a design automation tool that selects the best architecture parameters to minimise latency, while satisfying the accuracy specifications of the application and the FPGA resources available. Using this tool we show that it is possible to get sustained FPGA performance very close to the peak theoretical GPGPU performance when solving a single Lanczos problem to equivalent accuracy. If there are multiple independent problems to solve simultaneously, as described in Chapter 7, it is possible to exceed the peak floating-point performance of a GPGPU. If one considers the power consumption of both devices, the fixed-point Lanczos solver on the FPGA is more than an order of magnitude more efficient than the peak GPGPU efficiency. The test data are obtained from a benchmark set of problems from the large airliner optimal controller presented in Chapter 5.

Outline

The chapter starts by describing the Lanczos algorithm in Section 8.1. Section 8.2 presents the scaling procedure and contains the analysis to guarantee the absence of overflow in the Lanczos process. In Section ?? these results are extended to the MINRES method. The numerical results showing that the numerical quality of the linear equation solution does not suffer by moving to fixed-point arithmetic are presented in Section 8.3. In Section 8.4 we introduce an FPGA design automation tool that generates minimum latency architectures given accuracy specifications and resource constraints. This tool is used to evaluate the potential relative performance improvement between fixed-point and floating-point FPGA implementations and perform an absolute performance comparison against the peak performance of a high-end GPGPU. Section 8.5 discusses the possibility of extending this methodology to other nonlinear recursive kernels based on matrix vector multiplication and Section 8.6 discusses open topics in this area.
Algorithm 7 Lanczos algorithm

Require: Initial iterate \( r_1 \) such that \( \| r_1 \|_2 = 1 \), \( q_0 := 0 \) and \( \beta_0 := 1 \).

1: for \( i = 1 \) to \( i_{\text{max}} \) do
2: \( q_i \leftarrow \frac{r_i}{\beta_{i-1}} \)
3: \( z_i \leftarrow Aq_i \)
4: \( \alpha_i \leftarrow q_i^T z_i \)
5: \( r_{i+1} \leftarrow z_i - \alpha_i q_i - \beta_{i-1} q_{i-1} \)
6: \( \beta_i \leftarrow \| r_{i+1} \|_2 \)
7: end for
8: return \( q_i, \alpha_i \) and \( \beta_i \)

8.1 The Lanczos algorithm

The Lanczos algorithm [117] transforms a symmetric matrix \( A \in \mathbb{R}^{N \times N} \) into a tridiagonal matrix \( T \) (only the diagonal and off-diagonals are non-zero) with similar spectral properties as \( A \) using an orthogonal transformation matrix \( Q \). The method is described in Algorithm 7, where \( q_i \) is the \( i^{th} \) column of matrix \( Q \). At every iteration the approximation is refined such that

\[
Q_i^T A Q_i = T_i =: \begin{bmatrix}
\alpha_1 & \beta_1 & 0 \\
\beta_1 & \alpha_2 & \ddots \\
& \ddots & \ddots & \beta_{i-1} \\
0 & & \beta_{i-1} & \alpha_i
\end{bmatrix}, \quad (8.1)
\]

where \( Q_i \in \mathbb{R}^{N \times i} \) and \( T_i \in \mathbb{R}^{i \times i} \). The tridiagonal matrix \( T_i \) is easier to operate on than the original matrix. It can be used to extract the eigenvalues and singular values of \( A \) [76], or to solve systems of linear equations of the form \( Ax = b \) using the conjugate gradient (CG) [95] method when \( A \) is positive definite or the MINRES [174] method when \( A \) is indefinite. The Arnoldi iteration, a generalisation of Lanczos for non-symmetric matrices, is used in the generalized minimum residual (GMRES) method for general matrices [198] and is discussed in Section 8.5.1. The Lanczos (and Arnoldi) algorithms account for the majority of the computation in these methods – they are the key building blocks in modern iterative algorithms for solving all formulations of linear systems appearing in optimization solvers for optimal control problems.

Methods involving the Lanczos iteration are typically used for large sparse problems arising in scientific computing where direct methods, such as LU and Cholesky factorization, cannot be used due to prohibitive memory requirements [77]. However, iterative methods have additional properties that also make them good candidates for small problems arising in real-time applications, since they allow one to trade-off accuracy for computation time [38].
8.2 Fixed-point analysis

There are several challenges that need to be addressed before implementing an application in fixed-point. Firstly, one should determine the worst-case peak values for every variable in order to avoid overflow errors. The dynamic range has to be small such that small numbers can also be represented with a good level of accuracy. In interior-point solvers for model predictive control, some elements and eigenvalues of the KKT matrix have a wide dynamic range during a single solve, due to some elements becoming large and others small as the current iteration approaches the constraints. This affects the dynamic range of all variables in the Lanczos method. If one were to directly implement the algorithm in fixed-point, one would have to allocate a very large number of bits for the integer part to capture large numbers and an equally large number of bits for the fractional part to capture small numbers. Furthermore, there will be no guarantees of the avoidance of overflow errors, since most of the expressions cannot be analytically bounded in the general case.

For LTI algorithms it is possible to use discrete-time system theory to put tight analytical bounds on worst-case peak values [151]. A linear algebra operation that meets such requirements is matrix-vector multiplication, where the input is a vector within a given range and the matrix does not change over time. For some nonlinear non-recursive algorithms interval arithmetic [153] can be used to propagate data ranges forward through the computation graph [14]. Often this approach can be overly pessimistic for non-trivial graphs because it cannot take into account the correlation between variables.

For algorithms that do not fall in either of these two categories the tools available have limited power. In this section we first acknowledge the limitations of current tools for handling the bounding problem for the Lanczos algorithm and we then propose an alternative procedure based on linear algebra.

8.2.1 Results with existing tools

Linear algebra kernels for solving systems of equations, finding eigenvalues or performing singular value decomposition are nonlinear and recursive. The Lanczos iteration belongs to this class. For this type of computation the bounds given by interval arithmetic quickly blow up, rendering useless information. Table 8.1 highlights the limitations of state-of-the-art bounding tool Gappa [149] – a tool based on interval arithmetic – for handling the bounding problem for one iteration of Algorithm 7. Even when only one iteration is considered, the bounds quickly become impractical as the problem size grows, because the tool cannot use any extra information in matrix $A$ beyond bounds on the individual coefficients. Other recent tools [23] that can take into account the correlation between input variables can help to tighten the single iteration bounds, but there is still a significant amount of conservatism. More complex tools [44] that can take into account additional prior information on the input variables can further improve the tightness of the bounds. However, as shown in Table 8.1, the complexity of the procedure limits its usefulness to...
very small problems. In addition, the bounds given by all these tools will grow further for more than one iteration. As a consequence, these types of algorithms are typically implemented using floating-point arithmetic because the absence of overflow errors cannot be guaranteed, in general, with a practical number of fixed-point bits for practical problems.

Despite the acknowledged difficulties there have been several fixed-point implementations of nonlinear recursive linear algebra algorithms. CG-like algorithms were implemented in [31, 93], whereas the Lanczos algorithm was implemented in [102]. Bounds on variables were established through simulation-based studies and adding a heuristic safety factor. In the targeted digital signal processing (DSP) applications, the types of problems that have to be processed do not change significantly over time, hence this approach might be satisfactory, especially if the application is not safety critical. In other applications, such as in optimization solvers for embedded automatic control, the range of linear algebra problems that need to be solved on the same hardware is so varied that it is not possible to assign word-lengths based on simulation in a practical manner. Besides, in safety-critical applications analytical guarantees are desirable, since overflow errors can lead to unpredictable behaviour and even failure of the system [133].

### 8.2.2 A scaling procedure for bounding variables

We propose the use of a diagonal scaling matrix $M$ to redefine the problem in a new co-ordinate system to allow us to control the bounds in all variables, such that the same fixed precision arithmetic can efficiently handle problems with a wide range of matrices. For example, if we want to solve the symmetric system of linear equations $Ax = b$, where $A = A^T$, we propose instead to solve the problem

\[ MAMy = Mb \]
\[ \Leftrightarrow Ay = \hat{b}, \]

where $\hat{b}$ is a scaled version of $b$.
where
\[ \hat{A} := MAM, \]
\[ \hat{b} := Mb, \]

and the elements of the diagonal matrix \( M \) are chosen as
\[ M_{kk} := \frac{1}{\sqrt{\sum_{j=1}^{N} |A_{kj}|}} \quad (8.2) \]
to ensure the absence of overflow in a fixed-point implementation. The solution to the original problem can be recovered easily through the transformation \( x = My \).

An important point is that the scaling procedure and the recovery of the solution still have to be computed using floating-point arithmetic, due to the potentially large dynamic range and unboundness in the problem data. However, since the scaling matrix is diagonal, the cost of these operations is comparable to the cost of one iteration of the Lanczos algorithm. Since many iterations are typically required, most of the computation is still carried out in fixed-point arithmetic.

In order to illustrate the need for the scaling procedure, Figure 8.1 shows the evolution of the range of values of \( \alpha_i \) (Line 4 in Algorithm 7) throughout the solution of one optimization problem from the benchmark set described in Section 8.3. Notice that a different Lanczos problem has to be solved at each iteration of the optimization solver. Since the range of Lanczos problems that have to be solved on the same hardware is so diverse, without using the scaling matrix (8.2) it is not possible to decide on a fixed data format that can represent numbers efficiently for all problems. Based on the simulation results, with no scaling one would need to allocate 22 bits for the integer part to be able to represent the largest value of \( \alpha_i \) occurring in this benchmark set. Furthermore, using this number of bits would not guarantee that overflow will not occur on a different set of problems. The situation is similar for all other variables in the algorithm.

Instead, when using the scaling matrix (8.2) we have the following results:

**Lemma 3.** The scaled matrix \( \hat{A} := MAM \) has, for any non-singular symmetric matrix \( A \), spectral radius \( \rho(\hat{A}) \leq 1. \)

**Proof.** Let \( R_k := \sum_{j \neq k} |A_{kj}| \) be the absolute sum of the off-diagonal elements in a row, and let \( D(A_{kk}, R_k) \) be a Gershgorin disc with centre \( A_{kk} \) and radius \( R_k \). Consider an alternative non-symmetric preconditioned matrix \( \tilde{A} := M^2A \). The absolute row sum is equal to 1 for every row of \( \tilde{A} \), hence the Gershgorin discs associated with this matrix are given by \( D(\tilde{A}_{kk}, 1-|\tilde{A}_{kk}|) \). It is straightforward to show that these discs always lie inside the interval between 1 and -1 when \( |\tilde{A}_{kk}| \leq 1 \), which is the case here. Hence, \( \rho(\tilde{A}) \leq 1 \) according to Gershgorin’s circle theorem [77, Theorem 7.2.1]. Now, for an arbitrary eigenvalue-eigenvector pair \( (\lambda, v) \),
\[ M^2Av = \lambda v \quad (8.3) \]
\begin{align}
\leftrightarrow MAv &= M^{-1}\lambda v \\
\leftrightarrow MAmu &= \lambda u,
\end{align}

where (8.5) is obtained by substituting $Mu$ for $v$. This shows that the eigenvalues of the non-symmetric preconditioned matrix $\tilde{A}$ and the symmetric preconditioned matrix $\tilde{A}$ are the same. The eigenvectors are different but this does not affect the bounds, which we derive next.

\textbf{Theorem 2.} Given the scaling matrix (8.2), the symmetric Lanczos algorithm applied to $\tilde{A}$, for any non-singular symmetric matrix $A$, has intermediate variables with the following bounds for all $i$, $j$ and $k$:

- $[q_i]_k \in [-1, 1]$
- $[\tilde{A}]_{kj} \in [-1, 1]$
- $[\tilde{A}q_i]_k \in [-1, 1]$
- $\alpha_i \in [-1, 1]$
- $[\beta_{i-1}q_{i-1}]_k \in [-1, 1]$
- $[\alpha_iq_i]_k \in [-1, 1]$
- $[\tilde{A}q_i - \beta_{i-1}q_{i-1}]_k \in [-2, 2]$
• \( [r_{i+1}]_k \in [-1, 1] \)
• \( r_{i+1}^T r_{i+1} \in [0, 1] \)
• \( \beta_i \in [0, 1] \)

where \( i \) denotes the iteration number and \( \llbracket k \rrbracket \) and \( \llbracket k_j \rrbracket \) denote the \( k^{th} \) component of a vector and \( k_j^{th} \) component of a matrix, respectively.

**Corollary 2.** For the integer part of a fixed-point 2’s complement representation we require, including the sign bit, two bits for \( q_i, \hat{A} q_i, \alpha_i, \beta_i q_{i-1}, \alpha q_i, r_{i+1}, \beta_i \) and \( r_{i+1}^T r_{i+1} \), and three bits for \( \hat{A} q_i - \beta_{i-1} q_{i-1} \). Observe that the elements of \( M \) can be reduced by an arbitrarily small amount to turn the closed intervals of Theorem 2 into open intervals, saving one bit for all variables except for \( q_i \).

**Proof of Theorem 2.** The normalisation step in Line 2 of Algorithm 7 ensures that the Lanczos vectors \( q_i \) have unit norm for all iterations, hence all the elements of \( q_i \) are in \([-1, 1]\).

We follow by bounding the elements of the coefficient matrix:

\[
|\hat{A}_{kj}| = M_{kk} M_{jj} |A_{kj}| \leq \frac{1}{\sqrt{|A_{kj}|}} \frac{1}{\sqrt{|A_{kj}|}} |A_{kj}| = 1, \tag{8.6}
\]

where (8.6) follows from the definition of \( M \).

Using Lemma 3 we can put bounds on the rest of the intermediate computations in the Lanczos iteration. We start with \( \hat{A} q_i \), which is used in Lines 3, 4 and 5 in Algorithm 7:

\[
\|\hat{A} q_i\|_\infty \leq \|\hat{A} q_i\|_2 \leq \|\hat{A}\|_2 = \rho(\hat{A}) \leq 1 \quad \forall i, \tag{8.7}
\]

where (8.7) follows from the properties of matrix norms and the fact that \( \|q_i\|_2 = 1 \). The equality follows from the 2-norm of a real symmetric matrix being equal to its largest absolute eigenvalue \([77, \text{Theorem 2.3.1}]\).

We continue by bounding \( \alpha_i \) and \( \beta_i \), which are used in Lines 2, 4, 5 and 6 of Algorithm 7 and represent the coefficients of the tridiagonal matrix described in (8.1). The eigenvalues of the tridiagonal approximation matrix (8.1) are contained within the eigenvalues of \( \hat{A} \), even throughout the intermediate iterations \([77, \S 9.1]\). Hence, one can use the following relationship \([77, \S 2.3.2]\)

\[
\max_{j,k} |[T_i]_{jk}| \leq \|T_i\|_2 = \rho(T_i) \leq \rho(\hat{A}) \leq 1 \quad \forall i \tag{8.8}
\]

to bound the coefficients of \( T_i \) in (8.1), i.e. \( |\alpha_i| \leq 1 \) and \( |\beta_i| \leq 1 \), for all iterations.

Interval arithmetic can be used to show that the elements of \( \alpha_i q_i \) and \( \beta_{i-1} q_{i-1} \) are also between 1 and -1 and the elements of \( \hat{A} q_i - \beta_{i-1} q_{i-1} \) are in \([-2, 2]\).

The following equality

\[
A q_i - \alpha_i q_i - \beta_{i-1} q_{i-1} = \beta_i q_{i+1} =: r_{i+1} \quad \forall i, \tag{8.9}
\]
which always holds in the Lanczos process \[77, \S 9\], can be used to bound the elements of the auxiliary vector \( r_{i+1} \) in \([-1, 1]\) via interval arithmetic on the expression \( \beta_i q_{i+1} \). We also know that \( \beta_i \) is non-negative so its bound derived from (8.8) can be refined.

Finally, we can also bound the intermediate computation in Line 6 of Algorithm 7 using
\[
\|r_{i+1}\|_2 = |\beta_i|\|q_{i+1}\|_2 = |\beta_i| \leq 1 \quad \forall i,
\]
hence \( r_{i+1}^T r_{i+1} \) lies in \([0, 1]\).

The following points should also be considered for a reliable fixed-point implementation of the Lanczos process:

- Division and square root operations are implemented as iterative procedures in digital architectures. The data types for the intermediate variables can be designed to prevent overflow errors. In this case, the fact that \( |r_{i+1}|_k \leq \beta_i \) and \( r_{i+1}^T r_{i+1} \leq 1 \) can be used to establish tight bounds on the intermediate results for any implementation. For instance, all the intermediate variables in a CORDIC square root implementation \[157\] can be upper bounded by one if the input is known to be smaller than one.

- A possible source of problems both for fixed-point and floating-point implementations is encountering \( \beta_i = 0 \). However, this would mean that we have already computed a perfect tridiagonal approximation to \( \hat{A} \), i.e. the roots of the characteristic polynomial of \( T_{i-1} \) are the same as those of the characteristic polynomial of \( T_i \), signalling completion of the Lanczos process.

- If an upper bound estimate for \( \rho(A) \) is available, it is possible to bound all variables analytically without using the scaling matrix (8.2). However, the bounds will lose uniformity, i.e. the elements of \( q_i \) would still be in \([-1, 1]\) but the elements of \( A q_i \) would be in \([-\rho(A), \rho(A)]\).

The scaling operation that has been suggested in this section is also known as diagonal preconditioning. However, the primary objective of the scaling procedure is not to accelerate the convergence of the iterative algorithm, the objective of standard preconditioning. Sophisticated preconditioners attempt to increase the clustering of eigenvalues. Our scaling procedure, which has the effect on normalising the 1-norm of the rows of the matrix, can be applied after a traditional accelerating preconditioner. However, since this will move the eigenvalues, it cannot be guaranteed that the scaling procedure will not have a negative effect on the convergence rate. In such cases, a better strategy could be to include the goal of normalising the 1-norm of the rows of the matrix in the design of the accelerating preconditioner.
8.2.3 Validity of the bounds under inexact computations

We now use Paige’s error analysis of the Lanczos process [172] to adapt the previously derived bounds in the presence of finite precision computations. We are interested in the worst-case error in any component. In the following, we will denote with \( \epsilon_x \) the deviation of variable \( x \) from its value under exact arithmetic.

Unlike with floating-point arithmetic, fixed-point addition and subtraction operations involve no round-off error, provided there is no overflow and the result has the same number of fraction bits as the operands [223], which will be assumed in this section. For multiplication, the exact product of two numbers with \( k \) fraction bits can be represented using \( 2^k \) fraction bits, hence a \( k \)-bit truncation of a 2’s complement number incurs a round-off error bounded from below by \(-2^{-k}\). Recall that in 2’s complement arithmetic, truncation incurs a negative error both for positive and negative numbers.

The maximum absolute component-wise error in the variables involved in Algorithm 7 is summarised in the following proposition:

**Proposition 5.** When using fixed-point arithmetic with \( k \) fraction bits and assuming no overflow errors, the maximum difference in the variables involved in the Lanczos process described by Algorithm 7 with respect to their exact arithmetic values can be bounded by:

\[
\| \epsilon_q_i \|_\infty \leq (N + 4)2^{-k+1}, \tag{8.10}
\]
\[
\| \epsilon_z_i \|_\infty \leq \rho(\hat{A})\|\epsilon_q_i\|_\infty + N2^{-k}, \tag{8.11}
\]
\[
\| \epsilon_{\alpha_i} \|_\infty \leq \|\epsilon_q\|_\infty + \|\epsilon_z_i\|_\infty + N2^{-k}, \tag{8.12}
\]
\[
\| \epsilon_{r_i} \|_\infty \leq \rho(\hat{A})(N + 7)2^{-k}, \tag{8.13}
\]
\[
\| \epsilon_{\beta_i} \|_\infty \leq 2\|\epsilon_{r_i}\|_\infty + N2^{-k}. \tag{8.14}
\]

for all iterations \( i \), where \( \hat{A} \in \mathbb{R}^{N \times N} \).

**Proof.** In the original analysis presented in [172], higher order error terms are ignored since every term is assumed to be significantly smaller than one for the analysis to be valid, hence, higher order terms have a negligible impact on the final results. We do the same here as it significantly clarifies the presentation.

According to [172], the departure from unit norm in the Lanczos vectors can be bounded by

\[
|q_i^T q_i - 1| \leq (N + 4)2^{-k} \tag{8.15}
\]

for all iterations \( i \). In the worst case, all the error can be attributed to the same element in \( q_i \), hence, neglecting higher-order terms we have

\[
\| 2\epsilon_{q_i} \|_\infty \leq (N + 4)2^{-k}
\]

leading to (8.10).
The error in Line 3 of Algorithm 7 can be written, using the properties of matrix and vector norms, as

$$
\|\epsilon_{z_i}\|_\infty \leq \|\hat{A}\|_2 \|\epsilon_{q_i}\|_2 + N2^{-k},
$$

where the last term represents the maximum component-wise error in matrix-vector multiplication. Using (8.15) one can infer that the bound on \(\|\epsilon_{q_i}\|_2\) is the same as the bound on \(\|\epsilon_{q_i}\|_\infty\) given by (8.10), leading to (8.11).

Neglecting higher order terms, the error in \(\alpha_i\) in Line 4 of Algorithm 7 can be obtained by forward error analysis as

$$
\|\epsilon_{\alpha_i}\|_\infty \leq \|z_i\|_\infty \|\epsilon_{q_i}\|_\infty + \|q_i\|_\infty \|\epsilon_{z_i}\|_\infty + N2^{-k},
$$

where the last term arises from the maximum round-off error in the dot-product computation. Using the bounds given by Theorem 2 one arrives at (8.12).

Going back to the original analysis in [172] one can use the fact that the 2-norm of the error in the relationship (8.9), i.e.

$$
\|Aq_i - \alpha_iq_i - \beta_i - \beta_{i-1}q_{i-1} - \beta_{i+1}q_{i+1}\|_2
$$

can be bounded from below by

$$
\rho(\hat{A})(N + 7)2^{-k}
$$

(8.16)

for all iterations \(i\). One can infer that the bound on \(\|\epsilon_{r_i}\|_2\) is the same as (8.16). Using the properties of vector norms leads to (8.13).

The error in Line 6 of Algorithm 7 can be written as

$$
\|\epsilon_{\beta_i}\|_\infty \leq 2\|r_i\|_\infty \|\epsilon_{r_i}\|_\infty + N2^{-k}.
$$

Using the bounds given by Theorem 2 yields (8.14).

The error bounds given by Proposition 5 enlarge the bounds given in Theorem 2. In order to prevent overflow in the presence of round-off errors the integer bitwidth for \(q_i\) has to increase by \(\lceil \log_2((N + 4)2^{-k-1}) \rceil\) bits, which will be one bit in all practical cases. For the remaining variables, which have bounds that depend on \(\rho(\hat{A})\), one has two possibilities – either use extra bits to represent the integer part according to the new larger bounds, or adjust the spectral radius \(\rho(\hat{A})\) through the scaling matrix (8.2) such that the original bounds still apply under finite precision effects.

The latter approach is likely to provide effectively tighter bounds. We now outline a procedure, described in the following lemma, for controlling \(\rho(\hat{A})\) and give an example showing how to make use of it.
Lemma 4. If each element of the scaling matrix (8.2) is multiplied by \((1 + \varepsilon)\), where \(\varepsilon\) is a small positive number, the scaled matrix \(\hat{A} := MAM\) has, for any non-singular symmetric matrix \(A\), spectral radius \(\rho(\hat{A}) \leq \frac{1}{1 + \varepsilon}\).

Proof. We now have \(|\hat{A}_{kk}| \leq \frac{1}{1 + \varepsilon}\). The new Gershgorin discs are given by \(D(\hat{A}_{kk}, \frac{1}{1 + \varepsilon} - |\hat{A}_{kk}|)\), which can be easily proved to lie inside the interval between \(-\frac{1}{1 + \varepsilon}\) and \(\frac{1}{1 + \varepsilon}\). \(\square\)

For instance, if one decides to use \(k = 20\) fraction bits on the benchmark problems described in Section 8.3 with dimension \(N = 229\), the worst error bounds would be given by

\[
\|\epsilon_\alpha\|_\infty \leq 4.4 \times 10^{-4} [\rho(\hat{A}) + 1] + 4.37 \times 10^{-4},
\]
\[
\|\epsilon_\beta\|_\infty \leq 4.5 \times 10^{-4} \rho(\hat{A}) + 2.18 \times 10^{-4}.
\]

The value of \(\varepsilon\) in Lemma 4 is chosen such that the following inequalities are satisfied

\[
\rho(\hat{A}) + \|\epsilon_\alpha\|_\infty \leq 1,
\]
\[
\rho(\hat{A}) + \|\epsilon_\beta\|_\infty \leq 1,
\]

which, for the given values, is satisfied by \(\varepsilon \geq 0.0013\).

8.3 Numerical results

In this section we show that even though these algorithms are known to be vulnerable to round-off errors [173] they can still be executed using fixed-point arithmetic reliably by using our proposed approach.

In order to evaluate the numerical behaviour of the Lanczos method, we examine the convergence of the Lanczos-based MINRES algorithm, described in Algorithm 8, which solves systems of linear equations by minimising the 2-norm of the residual \(\|\hat{A}y_i - \hat{b}\|_2\).

Notice that in order to bound the solution vector \(y\), one would need an upper bound on the spectral radius of \(\hat{A}^{-1}\), which depends on the minimum absolute eigenvalue of \(\hat{A}\). In general it is not possible to obtain a lower bound on this quantity, hence the solution update cannot be bounded and has to be computed using floating-point arithmetic. Since our primary objective is to evaluate the numerical behaviour of the computationally-intensive Lanczos kernel, the operations outside Lanczos are carried out in double precision floating point arithmetic.

Figure 8.2 shows the convergence behaviour of a single precision floating point implementation and several fixed-point implementations for a symmetric matrix from the University of Florida sparse matrix collection [42]. All implementations exhibit the same convergence rate. There is a difference in the final attainable accuracy due to the accumulation of round-off errors, which is dependent on the precision used. The figure also shows that the fixed-point implementations have a stable numerical behaviour, i.e. the
Algorithm 8 MINRES algorithm

Require: Initial values $\gamma_1 := 1$, $\gamma_0 := 1$, $\sigma_1 := 0$, $\sigma_0 := 0$, $\zeta = 1$, $w_0 := 0$, $w_{-1} := 0$ and $y := 0$. Given $q_i$, $\alpha_i$ and $\beta_i$ from Algorithm 7:

1: for $i = 1$ to $i_{\text{max}}$ do 
2: \[ \delta_i \leftarrow \gamma_i \alpha_i - \gamma_{i-1} \sigma_i \beta_{i-1} \]
3: \[ \rho_{i,1} \leftarrow \sqrt{\delta_i^2 + \beta_i^2} \]
4: \[ \rho_{i,2} \leftarrow \sigma_i \alpha_i + \gamma_{i-1} \gamma_i \beta_{i-1} \]
5: \[ \rho_{i,3} \leftarrow \sigma_{i-1} \beta_{i-1} \]
6: \[ \gamma_{i+1} \leftarrow \frac{\delta_i}{\rho_{i,1}} \]
7: \[ \sigma_{i+1} \leftarrow \frac{\beta_i}{\rho_{i,1}} \]
8: \[ w_i \leftarrow \frac{q_i - \rho_{i,3} w_{i-2} - \rho_{i,2} w_{i-1}}{\rho_{i,1}} \]
9: \[ y \leftarrow y + \gamma_{i+1} \zeta w_i \]
10: \[ \zeta \leftarrow -\sigma_{i+1} \zeta \]
11: end for
12: return $y$

accumulated round-off error converges to a finite value. The numerical behaviour is similar for all linear systems for which the MINRES algorithm converges to the solution in double precision floating-point arithmetic.

In order to investigate the variation in attainable accuracy for a larger set of problems we created a benchmark set of approximately 1000 linear systems coming from an optimal controller for the Boeing 747 aircraft model [86, 127] described in Section 5.6 under many different operating conditions. The linear systems are the same size ($N = 229$) but the condition numbers range from 50 to $2 \times 10^{10}$. The problems were solved using a 32-bit fixed-point implementation, and single and double precision floating-point implementations, and the attainable accuracy was recorded in each case. The results are shown in Figure 8.3. As expected, double precision floating-point with 64 bits achieves better accuracy than the fixed-point implementations. However, single precision floating-point with 32 bits consistently achieves less accurate solutions. Since single precision only has 23 mantissa bits, a fixed-point implementation with 32 bits can provide better accuracy that a floating-point implementation with 32 bits if the problems are formulated such that the full dynamic range offered by a fixed representation can be efficiently utilised across different problems. Figure 8.3 also shows that these problems are numerically challenging – if the scaling matrix (8.2) is not used, even the floating-point implementations fail to converge. This suggests that the proposed scaling procedure can also improve the numerical behaviour of floating-point iterative algorithms along with achieving its main goal for bounding variables. An example application supporting this claim is described in [86] and in Section 5.6.

In order to evaluate the effect of the proposed approach on an optimization solver we consider a mixed precision interior-point solver where the Lanczos iterations – the most computationally intensive part in an interior-point solver based on an iterative linear solver (MINRES in this case) – is computed in fixed-point, whereas the rest of the algorithm
Figure 8.2: Convergence results when solving a linear system using MINRES for benchmark problem sherman1 from [42] with $N = 1000$ and condition number $2.2 \times 10^4$. The solid line represents the single precision floating-point implementation (32 bits including 23 mantissa bits), whereas the dotted lines represent, from top to bottom, fixed-point implementations with $k = 23, 32, 41$ and 50 bits for the fractional part of signals, respectively.

Figure 8.3: Histogram showing the final log relative error $\log_2(\frac{\|Ax-b\|_2}{\|b\|_2})$ at termination for different linear solver implementations. From top to bottom, preconditioned 32-bit fixed-point, double precision floating-point and single precision floating-point implementations, and unpreconditioned single precision floating-point implementation.
is computed in double precision floating-point. The fixed-point behaviour is simulated using Matlab’s fixed-point toolbox [208], which allows specifying rounding and overflow modes. When using floor rounding and no saturation, the results were verified to match simulation results on a Xilinx FPGA with the same options for the arithmetic units. The closed-loop behaviour of the different precision controllers was evaluated with a simulation where the aircraft is at steady-state and the reference changes at $t = 5s$. This change in reference guarantees that the input constraints become active. 150 MINRES iterations and 20 interior-point iterations are used in all cases in order to have a fair comparison. Figure 8.4 shows the accumulated cost for different controller implementations. When using the scaling procedure, the quality of the control is practically the same with the 32-bit fixed-point controller as with the double precision floating-point controller. For the unscaled controller we used information about the maximum value that all signals in the Lanczos process took for the benchmark set to decide how many bits to allocate for the integer part of each signal. The total number of bits for each signal was kept constant with respect to the preconditioned controller implementation. Of course, this cannot guarantee the absence of overflow so we changed the overflow mode to saturation (this would incur an extra penalty in terms of hardware resources). Figure 8.4 shows that it is essential to apply the scaler in order to be able to implement the mixed precision controller and still maintain the control quality.

![Figure 8.4: Accumulated closed-loop cost for different mixed precision interior-point controller implementations. The dotted line represents the unpreconditioned 32-bit fixed-point controller, whereas the crossed and solid lines represent the preconditioned 32-bit fixed-point and double precision floating-point controllers, respectively.](image-url)
The final attainable accuracy $\| \hat{A}y - \hat{b} \|_2$, denoted by $e_k$, is determined by the machine unit round-off $u_k$. When using floor rounding $u_k := 2^{-k}$, where $k$ denotes the number of bits used to represent the fractional part of numbers. It is well-known [96] that when solving systems of linear equations these quantities are related by

$$e_k \leq O(\kappa(\hat{A})u_k),$$

(8.17)

where $\kappa(\hat{A})$ is the condition number of the coefficient matrix. We have observed that, for $k \geq 15$, this relationship holds with approximate equality for all tested problems, including the problem described in Figure 8.2. For smaller bitwidths, excessive round-off error leads to unpredictable behaviour of the algorithm for the described application.

The constant of proportionality, which captures the numerical difficulty of the problem, is different for each problem. This constant cannot be computed a priori, but relationship (8.17) allows one to calculate it after a single simulation run and then determine the attainable accuracy when using a different number of bits, i.e. we can predict the numerical behaviour when using $k$ bits by shifting the histogram for 32 fixed-point fraction bits.

### 8.4 Evaluation in FPGAs

This section evaluates the impact of the results derived in Section 8.2 on FPGAs. We first describe a parameterizable FPGA architecture for the Lanczos process and then present a design automation tool that selects the degree of parallelisation and the computing precision to meet accuracy specifications and the resource constraints for the target chip. The resulting performance on a high-end FPGA is compared to the peak performance of a high-end GPGPU, which is considered the highest achievable single-chip performance for scientific computing.

#### 8.4.1 Parameterizable architecture

The results derived in Section 8.2 can be used to implement Lanczos-based algorithms reliably in low cost and low power fixed-point architectures, such as fixed-point DSPs and embedded microcontrollers. In this section, we will evaluate the potential efficiency improvements in FPGAs. In these platforms, for addition and subtraction operations, fixed-point units consume one order of magnitude fewer resources and incur one order of magnitude less arithmetic delay than floating-point units providing the same number of mantissa bits [232]. These platforms also provide flexibility for synthesizing different computing architectures. We now describe our architecture generating tool, which takes as inputs the data type, number of bits, level of parallelization and the latencies of an adder/subtractor ($l_A$), multiplier ($l_M$), square root ($l_{SQ}$) and divider ($l_D$) and automatically generates an architecture described in VHDL.

The proposed compute architecture for implementing Algorithm 7 is shown in Figure 8.5.
Figure 8.5: Lanczos compute architecture. Dotted lines denote links carrying vectors whereas solid lines denote links carrying scalars. The two thick dotted lines going into the $x^T y$ block denote $N$ parallel vector links. The input to the circuit is $q_1$ going into the multiplexer and the matrix $\hat{A}$ being written into on-chip RAM. The output is $\alpha_i$ and $\beta_i$.

The most computationally intensive operation is the $\Theta(N^2)$ matrix-vector multiplication in Line 3 of Algorithm 7. This is implemented in the block labeled $x^T y$, which is a parallel pipelined dot-product unit consisting of a parallel array of $N$ multipliers followed by an adder reduction tree of depth $\lceil\log_2 N\rceil$, as described in Figure 6.2. The remaining operations of Algorithm 7 are all $\Theta(N)$ vector operations and $\Theta(1)$ scalar operations that use dedicated components.

The degree of parallelism in the circuit is parameterized by parameter $P$. For instance, if $P = 2$, there will be two $x^T y$ blocks operating in parallel, one operating on the odd rows of $A$, the other on the even. All links carrying vector signals will branch into two links carrying the even and odd components, respectively, and all arithmetic units operating on vector links will be replicated. Note that the square root and divider, which consume most resources, only operate on scalar values, hence there will only be one of each of these units regardless of the value of $P$. For the memory subsystem, instead of having $N$ independent memories each storing one column of $A$, there will be $2N$ independent memories, where half of the memories will store the even rows and the other half will store the odd rows of $A$.

The latency for one Lanczos iteration in terms of clock cycles is given by

$$L := \left\lceil \frac{N}{P} \right\rceil + l_A \lceil \log_2 N \rceil + 5l_M + l_A + l_{SQ} + l_D + 2 + 2l_{\text{red}},$$

where

$$l_{\text{red}} := \left\lceil \frac{N}{P} \right\rceil + l_A \lceil \log_2 P \rceil + 2l_A + l_A \lceil \log_2 l_A \rceil - 1$$

(8.19)

is the number of cycles it takes the reduction circuit, illustrated in Figure 8.6, to reduce the incoming $P$ streams to a single scalar value. This operation is necessary when computing $q_i^T A q_i$ and $r_{i+1}^T r_{i+1}$. Note in particular that for a fixed-point implementation where $l_A = 1$.
Figure 8.6: Reduction circuit. Uses $P + l_A - 1$ adders and a serial-to-parallel shift register of length $l_A$.

Table 8.2: Delays for arithmetic cores. The delay of the fixed-point divider varies non-linearly between 21 and 36 cycles from $k = 18$ to $k = 54$.

<table>
<thead>
<tr>
<th>Representation</th>
<th>$l_A$</th>
<th>$l_M$</th>
<th>$l_D$</th>
<th>$l_{SQ}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>fixed-point</td>
<td>1</td>
<td>2</td>
<td>-</td>
<td>$\lceil \frac{k+1}{2} \rceil + 1$</td>
</tr>
<tr>
<td>float</td>
<td>11</td>
<td>8</td>
<td>27</td>
<td>27</td>
</tr>
<tr>
<td>double</td>
<td>14</td>
<td>15</td>
<td>57</td>
<td>57</td>
</tr>
</tbody>
</table>

and $P = 1$, the reduction circuit is a single adder and $l_{red} = N$, as expected. Table 8.2 shows the latency of the arithmetic units under different number representations.

As described in Section 6.2.1 floating-point units incur significantly larger delays than fixed-point units on FPGAs. Figure 8.7 shows the latency of 32-bit fixed-point and single precision floating-point implementations of the Lanczos kernel. For a fixed-point implementation, smaller arithmetic latencies mean that the constant term in the latency expression (8.18) has less weight, hence the incremental benefit of adding more parallelism is greater as a consequence of Amdahl’s law. Furthermore, a fixed-point implementation allows one to move further down the parallelism axis due to fewer resources being needed for individual arithmetic units. Larger problems benefit more from extra parallelism in all cases.

8.4.2 Design automation tool

In order to evaluate the performance of our designs for a given target FPGA chip we created a design automation tool that generates optimum designs with respect to the following rule:

$$\min_{P,k} L(P, k)$$
subject to

$$\Pr(e_k \leq \eta) > 1 - \xi, \quad (8.20)$$

$$R(P, k) \leq \text{FPGA area}, \quad (8.21)$$

where $L(P, k)$ is defined in (8.18) with the explicit dependence of latency on the number of fraction bits $k$ noted. $\Pr(e_k \leq \eta)$ represents the probability that any problem chosen at random from the benchmark set meets the user-specified accuracy constraint $\eta$, and is used to model the fact that for any finite precision representation – fixed point or double precision floating point – there will be problem instances that fail to converge for numerical reasons. The user can specify $\eta$ – the tolerance on the error, and $\xi$ – the proportion of problems allowed to fail to converge to the desired accuracy. In the remainder of the paper, we set $\xi = 10\%$, which is reasonable for the application domain for the data used. $R(P, k)$ is a vector representing the utilization of the different FPGA resources: flip-flops (FFs), look-up tables (LUTs), embedded multipliers and embedded RAM, for the Lanczos architecture illustrated in Figure 8.5 with parallelism degree $P$ and a $k$-bit fixed point datapath.

Even though this is an integer optimization problem, it can be easily solved. First, determine the minimum number of fraction bits $k$ necessary to satisfy the accuracy requirements (8.20) by making use of the information in Figure 8.3 and (8.21). Once $k$ is fixed, find the maximum $P$ such that (8.21) remains satisfied using the information in Ta-
Table 8.3: Resource usage

<table>
<thead>
<tr>
<th>Type</th>
<th>Amount</th>
</tr>
</thead>
<tbody>
<tr>
<td>Adder/Subtractor</td>
<td>$P(N + 3) + 2l_A - 2$</td>
</tr>
<tr>
<td>Multiplier</td>
<td>$P(N + 5)$</td>
</tr>
<tr>
<td>Divider</td>
<td>1</td>
</tr>
<tr>
<td>Square root</td>
<td>1</td>
</tr>
<tr>
<td>Memory - $2\left\lceil \frac{N}{P} \right\rceil$ $k$-bits</td>
<td>$PN$</td>
</tr>
<tr>
<td>Memory - $N$ $k$-bits</td>
<td>$5P$</td>
</tr>
</tbody>
</table>

Table 8.3 and a model for the number of LUTs, flip-flops and embedded multipliers necessary for implementing each arithmetic unit for different number of bits and data representations [232]. If $P = 1$ is not able to satisfy (8.21), then the problem is infeasible and either the accuracy requirements have to be relaxed or a larger FPGA will be necessary. Note that the actual resource utilization of the generated designs can differ slightly from the model predictions. However, the possible modelling error is insignificant compared to the efficiency improvements that will be presented in Section 8.4.3.

Memory is typically the limiting factor for implementations with a small number of bits, whereas for larger numbers of bits embedded multipliers limit the degree of parallelisation. In the former case, storage of some of the columns of $\hat{A}$ is implemented using banks of registers so FFs become the limiting resource. In the latter case, some multipliers are implemented using LUTs so these become the limiting resource. Figure 8.8 shows the trade-off between latency and FFs offered by the floating-point Lanczos implementations and two fixed-point implementations that, when embedded inside a MINRES solver, meet the same accuracy requirements as the single and double precision floating-point implementations. The trade-off is similar for other resources. We can see that the fixed-point implementations make better utilization of the available resources to reduce latency while providing the same solution quality.

8.4.3 Performance evaluation

In this section we will evaluate the relative performance of the fixed-point and floating-point implementations under the resource constraint framework of Section 8.4.2 for a Virtex 7 XT 1140 FPGA [234]. Then we will evaluate the absolute performance and efficiency of the fixed-point implementations against a high-end GPGPU with a peak floating-point performance of 1 TFLOP/s.

The trade-off between latency (8.18) and accuracy requirements for our FPGA implementations is investigated in Figure 8.9. For high accuracy requirements a large number of bits are needed reducing the extractable parallelism and increasing the latency. As the accuracy requirements are relaxed it becomes possible to reduce latency by increasing parallelism. The figure shows that the fixed-point implementations also provide a better trade-off even when the accuracy of the calculation is considered.

The simple control structures in our design and the pipelined arithmetic units allow the
circuits to be clocked at frequencies up to 400MHz. Noting that each Lanczos iteration requires $2N^2 + 8N$ operations we plot the number of operations per second (OP/s) against accuracy requirements in Figure 8.10. For extremely high accuracy requirements, not attainable by double precision floating-point, a fixed-point implementation with 53 fraction bits still achieves approximately 100 GOP/s. Since double precision floating-point only has 52 mantissa bits, a 53-bit fixed-point arithmetic can provide more accuracy if the dynamic range is controlled. For accuracy requirements of $10^{-6}$ and $10^{-3}$ the fixed-point implementations can achieve approximately 200 and 300 GOP/s, respectively. Larger problems would benefit more from incremental parallelisation leading to greater performance improvements, especially for lower accuracy requirements.

The GPGPU curves are based on the NVIDIA C2050 [170], which has a peak single precision performance of 1.03 TFLOP/s and a peak double precision performance of 515 GFLOP/s. It should be emphasized that while the solid lines represent the peak GPGPU performance, the actual sustained performance can differ significantly [60]. In fact, [182] reported sustained performance well below 10% of the peak performance when implementing the Lanczos kernel on this GPGPU.

The trade-off between performance and accuracy requirements is important for the range of applications that we consider. For some HPC applications, high accuracy requirements,
even beyond double precision, can be a high priority. On the other hand, for some embedded applications that require the repeated solution of similar problems, accuracy can be sacrificed for the ability to apply actions fast and respond quickly to new events. In some of these applications, solution accuracy requirements of $10^{-3}$ can be perfectly reasonable.

The results presented so far have assumed that we are processing a single problem at a time. Using this approach the arithmetic units in our circuit are always idle for some fraction of the iteration time. In addition, because the constant term in (8.18) is relatively large, the effect of incremental parallelisation on latency reduction becomes small very quickly. In the situation when there are many independent problems available [108], it is possible to use the idle computational power by time-multiplexing multiple problems into the same circuit to hide the pipeline latency and keep arithmetic units busy [124] in a similar fashion as was described in Chapter 7. In this case, the number of problems needed to fill the pipeline is given by the following expression

$$\left\lceil \frac{L}{N^2} \right\rceil,$$  \hspace{1cm} (8.22)

If the extra storage needed does not hinder the achievable parallelism, it is possible to achieve much higher computing performance, exceeding the peak GPGPU performance for most accuracy requirements even for small problems, as shown in Figure 8.10 (b). Using this approach there is a more direct transfer between parallelisation and sustained
Figure 8.10: Sustained computing performance for fixed-point implementations on a Virtex 7 XT 1140 [234] for different accuracy requirements. The solid line represents the peak performance of a 1 TFLOP/s GPGPU. $P$ and $k$ are the degree of parallelisation and number of fraction bits, respectively.
performance. The sharp improvement in performance for low accuracy requirements is a consequence of a nonlinear reduction in the number of embedded multiplier blocks necessary for implementing multiplier units, allowing for a significant increase in the available resources for parallelisation.

For the Virtex 7 XT 1140 [234] FPGA from the performance-optimized Xilinx device family, Xilinx power estimator [236] was used to estimate the maximum power consumption at approximately 22 Watts. For the C2050 GPGPU [170], the power consumption is in the region of 100 Watts, while a host processor consuming extra power would still be needed for controlling the data transfer to and from the GPGPU. Hence, for problems with modest accuracy requirements, there will be more than one order of magnitude difference in power efficiency when measured in terms of operations per watt between the sustained fixed-point FPGA performance and the peak GPGPU floating-point performance.

8.5 Further extensions

In this section we discuss several extensions for the results derived in Section 8.2. First, it is shown how the same procedure can be applied to bound variables for other similar iterative linear algebra kernels. We then discuss the possibility of solving the linear systems arising in an interior-point method using fixed-point arithmetic without implementing a scaling procedure.

8.5.1 Other linear algebra kernels

It is expected that the same scaling procedure presented in Section 8.2 will also be useful for bounding variables in other iterative linear algebra algorithms based on matrix-vector multiplication.

The standard Arnoldi iteration [6], described in Algorithm 9, transforms a non-symmetric matrix \( A \in \mathbb{R}^{N \times N} \) into an upper Hessenberg matrix \( H \) (upper triangle and first lower diagonal are non-zero) with similar spectral properties as \( A \) using an orthogonal transformation matrix \( Q \). At every iteration the approximation is refined such that

\[
Q_i^T A Q_i = H_i = \begin{bmatrix}
  h_{1,1} & h_{1,2} & \cdots & \cdots & h_{1,k} \\
  h_{2,1} & h_{2,2} & & & \\
  0 & h_{3,2} & & & \\
  & \ddots & \ddots & \ddots & \\
  & & 0 & h_{k,k-1} & h_{k,k}
\end{bmatrix},
\]

where \( Q_i \in \mathbb{R}^{N \times i} \) and \( H_i \in \mathbb{R}^{i \times i} \).

Since the matrix \( A \) is not symmetric it is not necessary to apply a symmetric scaling procedure; hence, instead of solving \( Ax = b \), we solve

\[
M^2 Ax = M^2 b
\]
Algorithm 9 Arnoldi algorithm

**Require:** Initial iterate $q_1$ such that $\|q_1\|_2 = 1$ and $h_{1,0} := 1.$

1. for $i = 1$ to $i_{\text{max}}$ do
2. $q_i \leftarrow \frac{r_{i-1}}{h_{i,i-1}}$
3. $z \leftarrow Aq_i$
4. $r_i \leftarrow z$
5. for $k = 1$ to $i$ do
6. $h_{k,i} \leftarrow q_k^T z$
7. $r_i \leftarrow r_i - h_{k,i}q_k$
8. end for
9. $h_{i+1,i} \leftarrow \|r_i\|_2$
10. end for
11. return $h$

$\Leftrightarrow \hat{A}x = \hat{b}$

and the computed solution remains the same as the solution to the original problem. The following proposition summarises the variable bounds for the Arnoldi process:

**Proposition 6.** Given the scaling matrix (8.2), the Arnoldi iteration applied to $\hat{A}$, for any non-singular matrix $A$, has intermediate variables with the following bounds for all $i$, $j$ and $k$:

- $[q_i]_k \in [-1, 1]$
- $[\hat{A}q_i]_k \in [-1, 1]$
- $[\hat{A}]_{kj} \in [-1, 1]$
- $[H]_{kj} \in [-1, 1]$

where $i$ denotes the iteration number and $[.]_k$ and $[.]_{kj}$ denote the $k^{th}$ component of a vector and $k^j$th component of a matrix, respectively.

**Proof.** According to the proof of Lemma 3, the spectral radius of the non-symmetric scaled matrix is still bounded by $\rho(\hat{A}) \leq 1$. As with the Lanczos iteration, the eigenvalues of the approximate matrix $H_i$ are contained within the eigenvalues of $\hat{A}$ even throughout the intermediate iterations. One can use the relationship (8.8) to show that the coefficients of the Hessenberg matrix are bounded by $\rho(\hat{A})$. The bounds for the remaining expressions in the Arnoldi iteration are obtained in the same way as in Theorem 2.

It is expected that the same techniques could be applied to other related kernels such as the unsymmetric Lanczos process or the power iteration for computing maximal eigenvalues.

### 8.5.2 Bounding variables without online scaling

This chapter has proposed a scaling procedure for bounding variables in the Lanczos process – the most computationally intensive part of an interior-point solver based on an iterative linear solver. In order to bound variables without using the scaling procedure one
needs to establish bounds on the largest absolute eigenvalues of the KKT matrix. This bounds should vary as little as possible throughout the iterations to be able to efficiently represent numbers using a fixed-point data format. It is also desirable for the bounds to be close to one, as suggested by Theorem 2.

With the saddle-point (2.15) and normal equations (2.17) linear system formulations, which are used in all interior-point software packages, the bounds on the largest absolute eigenvalue of the KKT matrix grows at least as $O(\frac{1}{\mu})$ [197], where $\mu$ is a measure of sub-optimality. This means that the largest eigenvalue becomes unbounded as the method progresses towards the solution. With these linear system formulations the scaling procedure is essential for a reliable and efficient fixed-point realisation of the algorithm. However, with the (symmetrized) unreduced linear system formulation (2.12) one can obtain upper bounds that are independent of the duality gap, hence constant throughout the interior-point method, and are of the same order as the largest eigenvalue of the Hessian matrix [82], which can be scaled offline to be close to one. This approach could potentially allow solving the linear systems in the interior-point method using fixed-point arithmetic without any online scaling overhead.

### 8.6 Summary and open questions

Fixed-point computation is more efficient than floating-point from the digital circuit point of view. We have shown that fixed-point computation can also be suitable for problems that have traditionally been considered floating-point problems if enough care is taken to formulate these problems in a numerically favourable way. Even for algorithms known to be vulnerable to numerical round-off errors accuracy does not necessarily have to be compromised by moving to fixed-point arithmetic if the dynamic range can be controlled such that a fixed-point representation can represent numbers efficiently.

Implementing an algorithm using fixed-point arithmetic gives more responsibility to the designer since all variables need to be bounded in order to avoid overflow errors that can lead to unpredictable behaviour. We have proposed a scaling procedure that allows us to bound and control the dynamic range of all variables in the Lanczos method – the building block in iterative methods for solving the most important linear algebra problems, which are ubiquitous in engineering and science. The proposed methodology is simple to implement but uses linear algebra theorems to establish bounds, which is currently well beyond the capabilities of state-of-the-art automatic tools for solving the bounding problem.

The capability for implementing these algorithms using fixed-point arithmetic could have an impact both in the high performance and embedded computing domains. In the embedded domain, it has the potential to open up opportunities for implementing sophisticated functionality in low cost systems with limited computational capabilities. For high-performance scientific applications it could help in the effort to reach exascale levels of performance while keeping the power consumption costs at an affordable level.
For other applications there are substantial processing performance and efficiency gains to be realised.

Since the proposed approach suggests a hybrid precision interior-point solver for embedded MPC, it seems natural to explore the possibility of implementation on heterogeneous computing platforms. In these platforms, the custom logic will implement the fixed-point computations whereas the (single precision) floating-point operations will be implemented on an ARM processor embedded on the same chip. This approach should boost the performance of the interior-point architecture described in Chapter 5. Given the performance gap between floating-point and fixed-point arithmetic and the performance results presented in Section 5.6, the revised implementation of the architecture presented in Chapter 5 should significantly exceed the performance of current state-of-the-art embedded interior-point solvers.

In cases where more performance is needed or the cost of floating-point arithmetic support is beyond budget, a full fixed-point interior-point implementation would be necessary. The first obstacle is the need to bound the search direction, or the solution to the linear systems, which requires lower bounds on the minimum absolute eigenvalue of the KKT matrices. With the unreduced linear system formulation (2.12), even if one can prove that there will be no eigenvalues at zero, the best lower bounds for the absolute eigenvalues are still at zero [82], hence better bounds are needed to be able to bound the components of the solution to the linear systems. One approach to solve this problem could come from adding regularization terms to the optimization problem to influence the lower bounds on the eigenvalues of the KKT system [78, 200].

Further performance and reliability enhancements could come from theoretical precision analysis. Presently, the design automation tool described in Section 8.4.2 makes precision decisions to meet the accuracy specifications using empirically obtained data. Unlike with first-order methods in Chapter 6, with interior-point methods it is currently not possible to give any practical theoretical bounds on the solution error given the number of bits used, even for the linear system subproblems.
9 Conclusion

This thesis has proposed several techniques for improving the computational efficiency of optimization solvers with the objective of enabling optimal decision making on resource-constrained embedded systems. In this chapter we summarise the main contributions and discuss some remaining challenges and future work directions to improve on the results presented in this thesis.

Several parameterisable hardware designs have been proposed for implementation in custom hardware platforms, such as FPGAs. For interior-point solvers, design decisions were made to exploit the significant structure in optimization problems arising in control, including a custom storage technique that reduced memory requirements substantially and allowed to overcome I/O bandwidth bottlenecks. For certain types of MPC problems, first-order solvers were proposed for high-speed and low cost implementations because the algorithms have few sequential dependencies and can be fully implemented using fixed-point arithmetic. While these algorithm-specific designs can provide substantial performance improvements over software solvers, the description of the circuit design techniques that result in highly efficient implementations, such as how to partition computations for maximum hardware efficiency or how to make use of long pipelines, is transferable and can be used to design efficient hardware architectures for other optimization algorithms not considered in this thesis.

This thesis has also presented analysis to aid making precision-related decisions for the design of the hardware architectures. The precision used to represent data should always be questioned in efficient hardware design, since a reduction in the number of bits used lowers the cost of the implementation and can increase its performance. For the interior-point designs, numerical investigations showed that with a preconditioning procedure and the correct plant model scaling, only a small number of linear solver iterations is required to achieve sufficient control accuracy for a numerically challenging airliner case study while using single precision floating-point arithmetic. For the different first-order solver designs, a unified error analysis framework was used to obtain practical a priori estimates for the expected error in the minimiser given the computing precision. Several case studies demonstrated that the algorithms remain numerically reliable at very low bit-widths in fixed-point arithmetic.

Novel ways of posing optimization problems, new MPC-specific algorithms and modifications to existing algorithms have also been proposed to make the most efficient use of custom pipelined parallel platforms. A new structured formulation for linear-time invariant constrained control problems was introduced, where the computational effort grew
linearly in the horizon length with several additional advantages over other sparse formulations. The structure was introduced through a suitable change of variables that led to banded matrices. Several methods were proposed to improve the hardware utilisation by time-multiplexing more than one independent problems onto the same datapath to hide the pipeline latency. We showed how employing one of these new strategies, which breaks the original problem into smaller subproblems, allows one to save resources and achieve greater acceleration. In terms of modification to existing algorithms it was shown that fixed-point computation can also be suitable for problems that have traditionally been considered floating-point problems if enough care is taken to formulate these problems in a numerically favourable way such that a fixed-point representation can represent numbers efficiently. We proposed a simple to implement scaling procedure that allowed bounding all variables in the Lanczos method - the computational bottleneck in interior-point methods based on iterative linear solvers, enabling reliable low cost fixed-point implementations.

Prior to this thesis, there had been a significant amount of research that shared our goal of extending the use of complex optimal decision making by proposing several ways to overcome the computational burden. The main novelty in this thesis is a multidisciplinary approach that considers the development of optimization algorithms, the digital design of custom optimization solvers, and the use of control theory and numerical analysis to make algorithm design and implementation decisions. We believe that by jointly considering all the fields involved in the deployment of efficient optimal decision makers it is possible to achieve better results than by considering the different design challenges separately. For instance, hardware design for application acceleration often tries to replicate the functional behaviour of the preceding software implementation, but this approach hides many of the degrees of freedom available in custom hardware design. Besides, it is often unclear whether the accuracy given by the double precision constrained software implementation is appropriate for the given application. Designing applications and algorithms that can deal with inexact computation is a promising path towards highly efficient implementations. Most of the algorithms proposed for accelerating optimization solvers for embedded control are only tested on x86-based machines. Even when the test results are satisfactory, deployment of such algorithms on precision-, power- and memory-constrained embedded platforms can result in unpredictable behaviour. In addition, designing optimization algorithms assuming sequential execution can prove to be a severe performance limiter in future computing platforms. One of the goals of this thesis is to promote a more holistic approach for the research and implementation of embedded optimal decision makers.

Through several case studies we have shown how the different techniques developed in this thesis could be used to implement complex optimization-based functionality on very cheap devices while meeting the real-time requirements of the system. However, this required skills that are not common in the practitioner. In order to promote the industrial adoption of these complex technologies in new resource-constrained applications and in applications that currently employ simple PID controllers it is necessary to provide a set of design tools that simplify the task of deploying an optimization solver on an embedded
platform. These tools should make automatic design choices given the characteristics of the problem and specifications of the target platform and should not require, ideally, any deep understanding of computing hardware, numerical analysis, or even optimization.

Most of the techniques described in this thesis and other related works can be programmed to automatically make design decisions and synthesize custom hardware solvers based on the characteristics and requirements of the target application. Additional work is needed to extend these results so that they are applicable to a more general class of problems, such as those with non quadratic objectives or with different constraint sets; however, these goals seem attainable with modest effort. The successful development of such tools could enable the adoption of optimization-based decision making in a range of applications. For instance, a domain with relatively fast dynamics and tight resource constraints is automotive control, whereas in space control applications the sampling requirements are not as tight but the power constraints are extreme. Beyond industrial control, there are some promising consumer applications too. For instance, compressed sensing could, in principle, be used for capturing images and videos in portable devices with reduced power consumption and sensor cost and size. Decoding such images on the mobile device without exhausting the limited battery power requires extremely efficient embedded optimization solvers.

9.1 Future work

At the end of Chapters 4, 5, 6, 7 and 8 we have outlined several open questions that could be further explored to improve the capabilities of the mentioned tools. In this section we discuss in more detail the two future research directions that we consider most promising.

9.1.1 Low cost interior-point solvers

Chapter 6 proposed architectures based on first-order optimization solvers for low cost implementations. These methods are very well-suited for resource constrained applications because they can be implemented using fixed-point arithmetic only and their simplicity enables analysis that can provide theoretical guarantees on their behaviour under reduced precision computations.

Interior-point methods can handle a much broader range of optimization problems than first-order methods so it is desirable to have low cost implementations too. The main obstacle is that not all variables can be bounded, hence fixed-point arithmetic is not guaranteed to result in reliable computations. In addition, due to the complexity of the method it is not possible to perform a numerical analysis that can provide practical conclusions for choosing the computing precision given an error tolerance at the solution.

Additional investigation into the numerical precision necessary for interior-point methods to behave in a reliable way would allow to explore further the efficiency trade-offs that are possible in custom hardware. In terms of bounding variables, Chapter 8 made the first step to bound the variables in the most computationally intensive task of an interior-point
solver. However, additional work is needed to bound the remaining tasks, starting with
the solution to the linear systems, which requires lower bounds on the minimum absolute
eigenvalue of the KKT matrices. Adding regularizing terms to the optimization problem
could be a promising direction.

9.1.2 Considering the process’ dynamics in precision decisions

The focus of the theoretical numerical analysis in this thesis has been on establishing
round-off error bounds on the minimiser and consequently on the function value in order
to decide how many bits to use to represent data. In a real-time context, where actions
are being applied at regular intervals and there is feedback between the process and the
decision maker, studying the effect of suboptimality in the solution on the closed-loop
tracking error or the disturbance rejection capabilities is also necessary to be able to
satisfy higher level application performance requirements.

For some applications the performance will be very sensitive to the quality of the control
action whereas others might not be as vulnerable to suboptimal decisions. In this thesis,
this kind of investigation has been carried out in an empirical manner. It would be desirable
to perform a theoretical analysis that could characterise the dependence of the tracking
and disturbance rejection capabilities on the quality of the applied actions. Including the
sampling period in this analysis would also be useful for optimally tuning all the free
parameters in a real-time embedded implementation.
Bibliography


186


[175] G. Pannocchia and J. B. Rawlings. Disturbance models for offset-free model predic-

[176] P. Patrinos and A. Bemporad. An accelerated dual gradient-projection algorithm for
linear model predictive control. In *Proc. 51st IEEE Conf. on Decision and Control*,
Maui, HI, USA, Dec 2012.


[179] T. Poggi, M. Rubagotti, A. Bemporad, and M. Storace. High-speed piecewise affine
2012.


implementation of the lanczos method for the symmetric extremal eigenvalue prob-
2012.


[184] C. V. Rao, S. J. Wright, and J. B. Rawlings. Application of interior-point meth-
ods to model predictive control. *Journal of Optimization Theory and Applications*,


